EXPERT INSIGHT

Machine Learning for Time Series with Python

Python to forecast, predict, and monitor time series with modern ML and foundation models at scale

Second Edition



packt

Dr. Ben Auffarth

Machine Learning for Time Series with Python

Second Edition

Python to forecast, predict, and monitor time series with modern ML and foundation models at scale

Dr. Ben Auffarth



Machine Learning for Time Series with Python

Second Edition

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Sunith Shetty

Relationship Leads: Tushar Gupta, Apeksha Shetty

Project Manager: Shashank Desai

Content Engineer: Ayushi Bulani

Technical Editor: Seemanjay Ameriya

Indexer: Rekha Nair

Production Designer: Ajay Patule

Growth Lead: Ankur Mulasi

Second Edition: November 2026

First published: October 2021

Production reference: 1011026

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-83763-133-9

www.packtpub.com

Contributors

About the author

Dr. Ben Auffarth, is a fractional Head of AI, technical book author, and founder of Chelsea AI Ventures, specializing in technical services and architecting mission-critical AI systems for travel, HR tech, and insurance among other sectors. His background is a blend of research (a PhD in Computational Neuroscience from KTH, Sweden, with published work cited over 580 times) and high-impact delivery at major UK companies. He is the author of multiple bestsellers, including Generative AI with LangChain.

It has been my goal for a long time to overhaul and update the first edition of this book with the latest and best including foundation models and prediction intervals. I would like to thank my family for letting me carve out time from family and personal life to work on this book.

About the reviewers

Anudeep Katangoori is a Data Platform Architect at Swift Transportation with 14+ years of experience designing and deploying enterprise-scale big data, AI, and cloud solutions across transportation, retail, healthcare, finance, e-commerce, and telecom. He holds a B.Tech in Computer Science and Engineering from JNTU, India, an MS in Computer Science from UNC Greensboro, and an MBA in Project Management from the American College of Commerce and Technology. He is currently pursuing an Executive MBA at Duke University's Fuqua School of Business. Anudeep is a Senior IEEE Member, Fellow IETE Member, and holds certifications in Google Cloud Professional Cloud Architect, Microsoft Azure Solutions Architect, Azure DevOps, and PMP.

Andrey Kostenko is a Data Science Solutions Architect holding a PhD in Mathematics and Statistics (Time Series Forecasting) from Monash University, followed by 10+ years of international industry experience across Australia and Singapore, spanning InsurTech, RetailTech, WaterTech, and AdTech startups alongside large-scale enterprise environments. He combines deep expertise in the Python and R ecosystems with technical certifications across AWS, Snowflake, and Oracle. Currently expanding into full-stack Solutions Engineering, he bridges predictive and generative AI with enterprise cloud infrastructure and production ML lifecycles. His credentials include a granted US Patent for real-time fraud mitigation and a Scholarship of the President of the Russian Federation.

Contents

- Preface
  - Who this book is for
  - What this book covers
  - To get the most out of this book
  - Get in touch
  - Free benefits with your book
    - How to unlock
1. Towards Modern Forecasting
   - The evolution of forecasting methods
   - What is a time series?
     - Time series in the Python context
   - When do smart models fail?
   - Why algorithms aren’t enough
     - Why advanced models still fail
       - Failure mode 1: misleading validation metrics
       - Failure mode 2: integration breakdown
       - Failure mode 3: inability to adapt to change
     - The three fundamental gaps
       - Gap 1: temporal assumption violations
       - Gap 2: misalignment with business needs
       - Gap 3: the problem with fixed models
     - Building adaptive systems
       - Question 1: is this problem actually foreseeable?
       - Question 2: what decisions will this prediction enable?
       - Question 3: how will patterns change over time?
       - Question 4: how quickly can I adapt when I’m wrong?
   - Why choose Python?
     - Choosing the right tools for time series problems
       - Forecasting
       - Classification
       - Anomaly detection
       - Regression
     - Core principles for production-ready systems
   - Common pitfalls and how to avoid them

        - Pattern 1: static system thinking to dynamic system design
        - Pattern 2: optimization mindset to adaptation infrastructure
        - Pattern 3: point estimate fixation to uncertainty-aware decision-making
        - Pattern 4: independence assumptions to temporal structure respect
        - Pattern 5: technical metric obsession to business value alignment
        - Pattern 6: model artifact focus to production system design
    - Summary
    - Further Reading
    - Get this book's PDF version and more
2. Preparing and Visualizing Time Series Data
    - Technical requirements
    - Mathematical foundations for data understanding
        - Temporal dependencies and autocorrelation
            - Proving temporal structure exists
            - Interpreting autocorrelation patterns for modeling strategies
            - Tools for autocorrelation analysis
        - Stationarity testing and interpretation
            - Visualizing stationarity
            - Statistical stationarity tests
            - Achieving stationarity
            - Stabilizing variance with log and Box–Cox transformations
        - Signals from decomposition
            - Why the additive vs. multiplicative choice matters
            - Practical decomposition in Python
            - Choosing the right decomposition method
    - Building a time series data pipeline
        - Temporal indexing and frequency management
        - The strategic solution: robust conversion and validation
        - Quality checks and missing data
            - Missing data assessment
            - Detecting structural breaks
            - Quality threshold decision-making

        - Frequency conversion and resampling decisions
            - The resampling trap: Look-Ahead bias
            - Practical decision framework
    - Pattern discovery through visualization
        - Visual diagnostics for temporal structures
        - Autocorrelation and residual analysis
    - Forecastability assessment
        - Measuring predictability before modeling
            - Signal-to-noise proxy
            - The naive model
            - The seasonal naive benchmark
        - A framework for project viability
    - Summary
3. Classical Models and Validation
    - Why statistical methods matter
    - The role of baselines in forecasting
    - Classical baselines
        - Exponential smoothing
        - ARIMA
        - Theta Method
        - Overview of statistical forecasting methods
            - Case study: forecasting “remote work”
    - Validation and metrics
        - Why random splits fail
            - Expanding window
            - Rolling window
        - Advanced techniques
            - Walk-forward validation
            - Nested cross-validation
        - Common pitfalls
        - Performance metrics
            - Scale-dependent metrics
            - Scale-independent metrics
            - An experiment
        - Industry-specific validation strategies
    - Summary
    - Get this book’s PDF version and more

4. Forecasting with Machine Learning
    - Forecasting as a regression task
    - The ML forecasting workflow
        - Step 1: data preparation
        - Step 2: feature engineering
        - Step 3: model training
        - Step 4: prediction
        - Step 5: validation
    - The role of tooling
    - Tools for the job
        - Sktime
            - Darts
            - The Nixtla ecosystem
        - AutoML platforms
        - Direct scikit-learn integration
        - Choosing your tool: a decision framework
    - Creating an ML forecast
        - Step 1: setup and baseline
        - Step 2: manual feature engineering
        - Step 3: training the model
        - Step 4: prediction
        - Step 5: validation
    - Summary
5. Feature Engineering and Tree-Based Models
    - Technical requirements
    - Feature engineering
        - Calendar features
        - Event and holiday features
    - Advanced feature engineering
        - Lag features
        - Rolling window statistics
        - Encoding cyclical time
        - Domain-specific features
        - A practical guide to feature engineering
        - Building a reusable pipeline
    - Choosing a tree model
    - Hybrid modeling and decomposition

        - Pattern 1: Decompose-then-model-residuals
        - Pattern 2: Fourier features for seasonality
    - Automated feature engineering
        - Recipe 1: mlforecast and XGBoost
        - Recipe 2: tsfresh and Darts
        - Recipe 3: functime
    - Summary
    - Get this book's PDF version and more
6. Multivariate and Hierarchical Forecasting
    - Multivariate models
        - What are multivariate forecasts?
        - VAR and VARMAX
            - A simple VAR model
            - Granger causality
            - VARMAX
        - Global multi-series models
    - Hierarchical models
        - What are hierarchies?
        - Reconciliation approaches
            - The bottom-up approach
            - The top-down approach
            - The middle-out approach
            - Modern reconciliation: optimal combinations
    - Case study: Forecasting tourism in Australia
        - Evaluating and comparing results
    - Summary
    - Further Reading
7. Practical Deep Learning for Time Series
    - Technical requirements
    - Why deep learning?
    - Foundational models
        - Long short-term memory
        - Temporal Convolutional Networks
    - Linear and MLP-based models
        - DLinear
        - N-BEATS
        - TSMixer

- Transformer models
  - Temporal fusion transformer (TFT)
  - PatchTST
  - iTransformer
  - Crossformer
- 2D and decomposition models
- Model selection
- Practical tips for training deep learning models
- Further Reading
- Summary
- Get this book's PDF version and more

8. Quantifying Time Series Uncertainty with Conformal Prediction
   - The problem with uncertainty
   - The conformal prediction framework
     - The conformal guarantee
     - The split conformal algorithm
   - Fixed-width intervals with mapie
   - Adaptive intervals with conformalized quantile regression (CQR)
     - The failure of fixed-width intervals
     - Implementation with mapie
   - Recipes for the modern time series ecosystem
     - Recipe 1: nixtla/statsforecast
     - Recipe 2: sktime
     - Recipe 3: DARTS (native conformal models)
   - Conformal prediction for non-exchangeable time series
     - EnbPI in practice
     - ACI in practice
     - Multi-horizon coverage
   - Production-ready uncertainty
     - Evaluating interval quality
     - Monitoring for calibration drift
       - Building a rolling coverage dashboard
       - Recalibration strategy
     - Diagnosing conditional coverage
   - Modeling financial volatility
     - Identifying volatility clustering

        - A hybrid model: GARCH combined with conformal prediction
        - Validating exchangeability of standardized residuals
        - Production and deployment patterns
    - Choosing your uncertainty method
    - Summary
    - Further Reading
9. Foundation Models: Quantitative and Qualitative Forecasting
    - A taxonomy of foundation model architectures
    - The need for time series foundation models
        - A hands-on example
    - Quantitative foundation models
        - Google TimesFM
        - Amazon Chronos
        - Salesforce Moirai
    - Qualitative and multimodal forecasting
        - Introducing OpenTSLM
        - Prompt engineering for multimodal LLMs: Chain-of-Charts
        - Multimodal datasets
    - Critical challenges
        - Performance risks
        - Generalization risks
        - Memorization risks
        - Adversarial fragility
        - Probabilistic calibration
        - Long-horizon decay
        - Deploying foundation models in production
    - Summary
    - Get this book's PDF version and more
10. Production Workflows: Deployment, Monitoring, and Scaling
    - The production gap
    - Architectural patterns for time-series deployment
        - Batch processing architectures
            - The schedule-compute-cache pattern
            - Global context and robustness
        - Stream processing (real-time) architecture
            - Event-driven inference and state management

        - The training-serving skew challenge
    - Hybrid approaches: lambda and kappa architectures
- Data engineering
    - Point-in-time correctness and time travel
        - The Feature Store solution
    - Handling data gaps, irregularities, and censored data
    - Data validation with Pandera
- Scaling strategies
    - Local vs. global modeling paradigms
    - Distributed forecasting with ray
    - Hierarchical reconciliation at scale
- Deployment strategies
    - Shadow deployment
    - Model serialization and MLflow
- Monitoring and observability
    - Proxy metrics and drift detection
    - Performance estimation (CBPE)
- Advanced failure modes and mitigation
    - Censored data and the demand death spiral
    - Feedback loops and the bullwhip effect
- Summary

11. Beyond Forecasting: Specialized Applications
    - Time series classification and structural pattern recognition
        - Self-supervised representation
        - sktime vs. aeon
        - The human activity recognition problem
        - Diffusion models for time series
    - Managing nonstationarity
        - Drift detection methods
        - Implementation: adaptive windowing (ADWIN)
        - Graph and spatiotemporal modeling
        - Neural ODEs for irregular time series
    - Causal inference
        - Intervention analysis with CausalImpact
        - Benchmarks
    - Summary
    - Get this book’s PDF version and more

12. Intermittent Forecasting and Survival Analysis
    - Intermittent demand forecasting
        - The mathematics of sparsity and obsolescence
        - Hierarchical Bayesian formulations
        - Implementation of intermittent heuristics
        - Stacking and hurdle-based combinations
        - Deep learning architectures for zero-inflated data
    - Time-to-event and survival analysis
        - The challenge of censoring and truncation
        - Mathematical foundations: hazard and state-space transformations
        - Implementation of the Cox Proportional Hazards model
        - Multi-horizon forecasting and tree-based survival
        - Evaluating time-to-event calibration
        - Discrete-time hazard forecasting at scale
    - Applied survival and reconciliation
    - Summary
13. Unlock Your Exclusive Benefits
    - Unlock this Book’s Free Benefits in 3 Easy Steps

Other Books You May Enjoy

Index

Preface

In March 2020, Target's machine learning models, built by world-class data science teams, told the company to keep stocking jeans and party dresses. Customers were busy clearing the shelves of toilet paper, sweatpants, and cleaning supplies. Apparel comp sales fell more than 30 percent in the worst weeks of March; electronics and household essentials surged 20 to 45 percent in the other direction. At Amazon, the algorithms that had balanced millions of SKUs were overwhelmed: household essentials vanished from the site, and the company suspended inbound shipments of anything it classified as non-essential just to free up warehouse capacity. At Walmart, stockouts climbed across 4,700 U.S. stores faster than the planning systems could adjust, and U.S. inventory finished the quarter down nearly five percent year-over-year.

Within weeks, billions of dollars of forecasting infrastructure went from a competitive advantage to a source of systematic error.

Twenty years earlier, Nike had taken a $100 million hit to quarterly sales, and a 20 percent stock drop the day it was announced, after its newly installed i2 demand planner over-ordered Air Garnett sneakers and under-ordered Air Jordans. In December 2011, Best Buy canceled online holiday orders days before Christmas, weeks after it had accepted payment, blaming demand that had outrun stock. In August 2020, California's grid operators ordered rolling blackouts because their solar-generation forecasts had not learned

the negative correlation between sunset and air-conditioning load during a heatwave. Different industries, different decades, the same pattern: smart teams with state-of-the-art models, blindsided by the way time series actually behaves in production.

This book is about why that keeps happening and how to build systems that hold up when it does.

The short answer is that time series breaks the assumption almost every other machine learning method depends on, that observations are independent. Today's value is a function of yesterday's. Shuffle the rows and the signal disappears. Validate with a random split and you train on the future, test on the past, and ship a model that looks great in the notebook and fails the first time it meets a pattern it has not memorized. Add the fact that most production forecasting pipelines are 5 percent algorithm and 95 percent data engineering, calibration, monitoring, and drift handling (what *Chapter 1* calls the *Iceberg Problem*), and you have a discipline where the gap between a working prototype and a working system is wider than in almost any other corner of machine learning.

Across twelve chapters, we work through the full stack a modern forecasting practitioner needs: diagnostics, classical baselines, machine learning, deep learning, hierarchical, and multivariate models, conformal prediction intervals, time-series foundation models, production deployment, anomaly detection, and classification, and the awkward cases (intermittent demand, censored events) that most introductory books skip. The thread holding it together is the production gap. Every chapter asks two questions: *does the model fit?* and *will the model survive a Tuesday?*

Who this book is for

This book is for data scientists, machine learning engineers, and analysts who already know Python and the standard ML toolbox (pandas, scikit-learn, gradient boosting, a bit of PyTorch) and who now need to ship forecasting systems that work in production. You do not need a statistics PhD, but you should be comfortable with linear regression, basic probability, and the idea of cross-validation.

If you are a practitioner, every chapter is structured so you can read the conceptual sections, copy the code that fits your problem, and skip the deeper dives.

If you are coming from classical time series and want to understand the modern machine-learning, deep-learning, and foundation-model toolbox, the path from *Chapter 3* forward is for you.

If you are coming from ML and have never worked with a time series before, start with *Chapter 2* and pay close attention to validation in *Chapter 3*.

What this book covers

The book moves through four arcs. *Chapters 1–3* are foundations: what makes time series different from cross-sectional data, how to diagnose a series before modeling it, and the validation discipline (rolling-origin splits, scale-invariant metrics, beating a baseline) that catches data leakage before it ships. *Chapters 4–7* are the modern modeling toolkit: forecasting reframed as supervised regression, feature engineering with gradient-boosted trees, multivariate, and hierarchical methods, and deep learning. *Chapters 8 and 9* are about reliability and the current frontier: conformal prediction for uncertainty intervals you can defend in front of an inventory planner, and time-series foundation models for the zero-shot and cold-start cases. *Chapters 10–12* are about production and the awkward problems most introductions skip: deployment and monitoring on a real system, classification, and anomaly detection, and the sparse data (intermittent demand, right-censored events) that breaks regression-style thinking.

`Chapter 1`, *Towards Modern Forecasting*, opens with the case studies above and develops the two ideas the rest of the book leans on: the **Prediction Paradox** and the **Iceberg Problem**. It also surveys the 2026 toolbox: time-series foundation models, conformal prediction, global and hierarchical methods, and the platform-native endpoints that have appeared since the first edition of this book.

`Chapter 2`, *Preparing and Visualizing Time Series Data*, teaches you to diagnose your data before you model it. You compute autocorrelations, test for stationarity, build a temporal data pipeline

with proper indexing and resampling, surface seasonality and cycles through specialized plots, and assess whether a series has enough signal to be worth forecasting at all. The bike-sharing dataset is our running example.

`Chapter 3`, *Classical Models and Validation*, builds the baselines every advanced model must beat: Exponential Smoothing, ARIMA, and the Theta method. Equally important, it builds the validation framework you reuse in every later chapter.

`Chapter 4`, *Forecasting with Machine Learning*, makes the conceptual jump from *decomposing a signal* to *treating forecasting as supervised regression*. This single reframing opens up the entire ML catalog (gradient boosting, neural networks, and the rest) for time series, and the chapter walks through the workflow end to end with `skforecast`.

`Chapter 5`, *Feature Engineering and Tree-Based Models*, is where the gains live. Using the M5 Walmart dataset, you build lag features, rolling statistics, calendar encodings, automated feature pipelines, and gradient-boosted models that scale to thousands of series at once.

`Chapter 6`, *Multivariate and Hierarchical Forecasting*, handles the cases where univariate methods break: variables that drive each other and forecasts that must reconcile across a hierarchy. You use VAR and VARMAX, global cross-learning with `mlforecast`, and modern reconciliation methods like `MinT`, applied to the Australian tourism dataset.

Chapter 7, *Practical Deep Learning for Time Series*, covers the architectures that now drive production forecasting at scale: LSTMs and TCNs as foundations, DLinear and TSMixer for the simplicity backlash, and the Transformer family (TFT, PatchTST, iTransformer) for long horizons and many covariates. The chapter rotates through darts, neuralforecast, and sktime so you learn to read across the ecosystem rather than memorize one API.

Chapter 8, *Quantifying Time Series Uncertainty with Conformal Prediction*, replaces the brittle Gaussian intervals that classical models ship with by default. You build split-conformal intervals, adaptive intervals via Conformalized Quantile Regression, online recalibration for drift, and a hybrid GARCH-conformal model for financial volatility. The chapter opens with a decision tree so you can pick the right method from your residuals.

Chapter 9, *Foundation Models: Quantitative and Qualitative Forecasting*, covers time-series foundation models (Chronos, TimesFM, Moirai, Tiny Time Mixers) and when zero-shot inference beats per-series training. It also covers the qualitative frontier: combining a numerical forecaster with a language model so the system can read context and explain why a forecast moved.

Chapter 10, *Production Workflows: Deployment, Monitoring, and Scaling*, follows a fictional ride-hailing platform, Glide, from notebook to production: architectural patterns (online versus batch, Lambda versus Kappa), feature stores, monitoring, drift detection, and the failure modes peculiar to time series (the Bullwhip Effect, feedback loops, freshness SLAs).

Chapter 11, *Beyond Forecasting: Specialized Applications*, expands the toolkit past regression: time-series classification with ROCKET, MiniRocket, and HIVE-COTE; anomaly detection with foundation-model embeddings; generative models with diffusion; graph and spatio-temporal models; neural ODEs for irregularly sampled data; and causal inference for marketing-attribution-style questions.

Chapter 12, *Intermittent Forecasting and Survival Analysis*, handles the data the rest of the field tends to skip: long runs of zeros and right-censored events. You work through Croston, SBA, and TSB alongside Kaplan-Meier and Cox models, and connect both to modern probabilistic deep models.

To get the most out of this book

The code is tested against `Python 3.12` and the pinned versions in the companion repository's `pyproject.toml`. A working setup looks like this:

```
git clone https://github.com/PacktPublishing/Machine-Learning-for-Time
cd Machine-Learning-for-Time-Series-with-Python-Second-Edition
pip install -e .
```

Each chapter's notebooks live in their chapter-wise repositories and reuse a single environment, conventionally called `ml_timeseries`. *Chapter 7* and the foundation-model sections of *Chapter 9* benefit from a GPU; the rest is laptop-runnable. Datasets either ship with the libraries we use (AirPassengers, Australian tourism, bike-sharing, ETTh1) or come with a one-line download script (M5 from Zenodo).

The companion repository runs the notebooks under GitHub Actions on every push. If the code in the book drifts from the repository, the build goes red and we fix it. Open an issue on GitHub if you spot something out of sync, and we will treat it as a bug.

You can read the chapters in order, but the dependencies are loose. *Chapter 2* sets up the diagnostic vocabulary the whole book uses. *Chapter 3*'s validation framework reappears everywhere. After that, chapters are mostly self-contained, with explicit pointers when one builds on another.

Download the example code files

The code bundle for the book is hosted on GitHub at https://github.com/PacktPublishing/Machine-Learning-for-Time-Series-with-Python-Second-Edition. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "If you are new to ML forecasting, start with `skforecast`."

A block of code is set as follows:

```
import yfinance as yf
def fetch_aapl():
# Fetch Apple stock data
aapl = yf.download(
    'AAPL', start='2020-01-01',
    end='2023-12-31', progress=False
)
aapl['returns'] = aapl['Close'].pct_change() * 100
return aapl
```

Any command-line input or output is written as follows:

```
pip install pandas numpy matplotlib scipy statsmodels yfinance
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “To predict the sales on **Day 3** (which were **15**), we use the sales from **Day 2** (**12**) and **Day 1** (**10**).”



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at

`customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported it to us. Please visit http://www.packt.com/submit-errata, click **Submit Errata**, and fill in the form. If you find any issues with or have suggestions about the code, please go to https://github.com/PacktPublishing/Machine-Learning-for-Time-Series-with-Python-Second-Edition and file an issue.

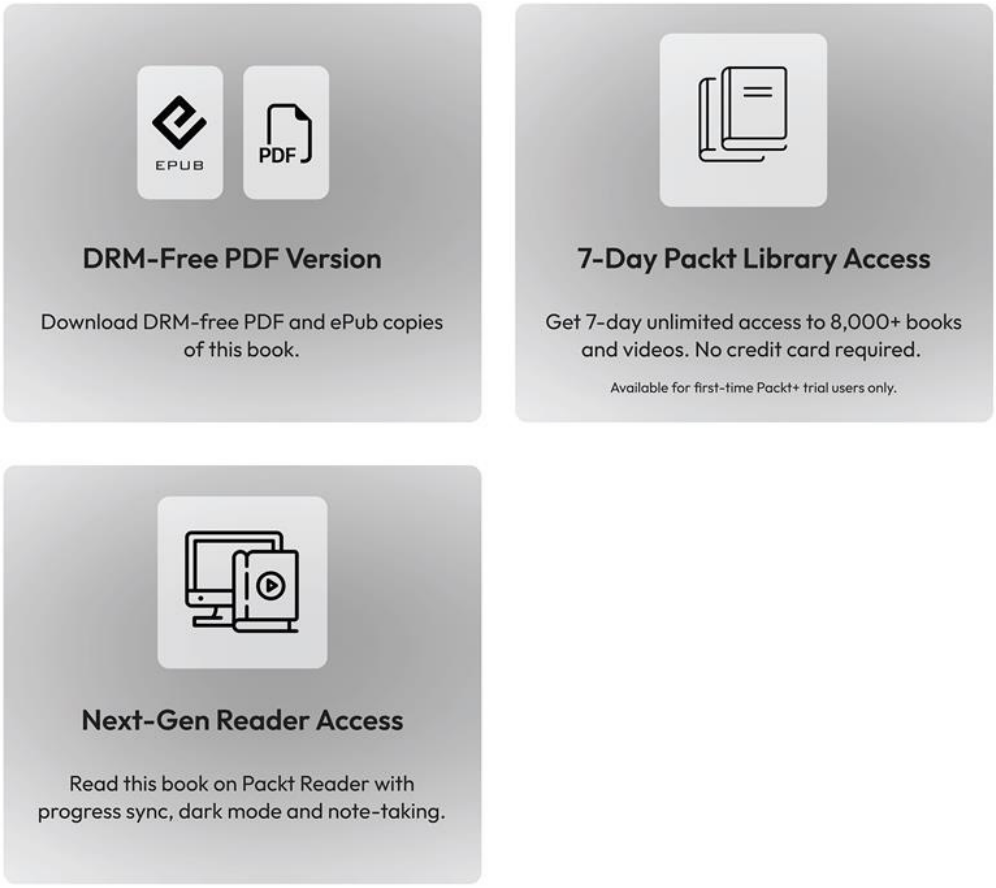
Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you provided us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit http://authors.packt.com/.

Free benefits with your book

This book comes with free benefits to support your learning. Activate them now for instant access (see the "*How to unlock*" section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



How to unlock

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

Share your thoughts

Once you've read *Machine Learning for Time Series with Python, Second Edition*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Towards Modern Forecasting

Time series problems are critical in industry and academia, with applications ranging from stock market analysis to predicting consumer behavior in retail. As **Machine Learning** (**ML**) for time series continues to evolve, practitioners are faced with a sprawling set of libraries and algorithms. However, building reliable forecasting systems requires the understanding of what makes time series fundamentally different from the cross-sectional data that most ML methods assume. Building such systems end to end is what this book teaches: modern models including time-series foundation models, global and hierarchical forecasting, conformal prediction intervals, drift detection, and the operational habits production demands.

In this chapter, we will explore why modern, technically advanced forecasting systems can fail spectacularly when faced with unexpected events, a phenomenon we call the *Prediction Paradox*. We will examine case studies from major retailers like Target and Amazon during the COVID-19 pandemic and the historical precedent set by Nike's supply chain disaster to understand this enduring challenge. We will also introduce the *Iceberg Problem*, which illustrates that the visible algorithm is only 5% of the work, while the hidden 95% (including data engineering, validation

strategy, and monitoring) ultimately determines production success. Both ideas are why we treat conformal prediction as foundational rather than an afterthought: paradox-prone models need coverage guarantees, and operating safely on the iceberg's hidden 95% means knowing when a forecast cannot be trusted.

After reading this chapter, you will understand what makes time series data unique, know the main types of time series problems, and have a framework for approaching any temporal analysis project. The sections are structured as follows:

- The evolution of forecasting methods
- When do smart models fail?
- What is a time series?
- Why algorithms aren't enough
- Why choose Python?
- Common pitfalls and how to avoid them

Let's have a look at how forecasting and time series methods have changed throughout history as we take the whirlwind quest from star charts to supercomputers.



Your purchase includes a free PDF copy + exclusive extras

Your purchase includes a DRM-free PDF copy of this book, 7-day trial to the Packt+ library (no credit card required), and additional exclusive extras. See the *Free benefits with your book section* in the *Preface* to unlock them instantly and maximize your learning.

The evolution of forecasting methods

Humanity's quest to forecast the future has always walked a fine line between reverence and fear. From merchants in medieval Europe who were accused of witchcraft for accurately predicting market trends to court astronomers in ancient China who faced execution for failing to predict solar eclipses, the stakes have always been high. For centuries, our methods were no more reliable than reading tea leaves. Today, we have replaced these arcane arts with time series analysis, a blend of statistics, mathematics, and ML that allows us to decode the patterns of our world.

This journey toward data-driven analysis has been long. In 1854, even though Vice Admiral Robert FitzRoy's early weather predictions were dismissed as quack prognostications, he was laying the groundwork for a more systematic approach to forecasting. The field took a mathematical leap forward with Francis Galton's work on regression in the 1880s, but the next true revolution arrived in the 1970s, when George Box and Gwilym Jenkins introduced their ARIMA models, providing a rigorous framework for time-dependent data that is still a cornerstone of the field today.

The digital age, however, catalyzed an explosion of innovation that brought both unprecedented power and new forms of peril. The 2010 Flash Crash, when the Dow Jones plunged nearly 1,000 points in minutes, was triggered by high-frequency trading algorithms reacting to time series data at millisecond intervals. This event was a stark demonstration of the potential pitfalls of modern forecasting,

spurring the development of more sophisticated models capable of detecting and responding to anomalies in real time.

The dawn of this computational era can be traced to a single, room-sized machine: the **Electronic Numerical Integrator and Computer** (**ENIAC**). Originally used for calculating artillery trajectories, this behemoth of 17,500 vacuum tubes was repurposed by programming pioneers like Betty Snyder to run the world's first numerical weather predictions.

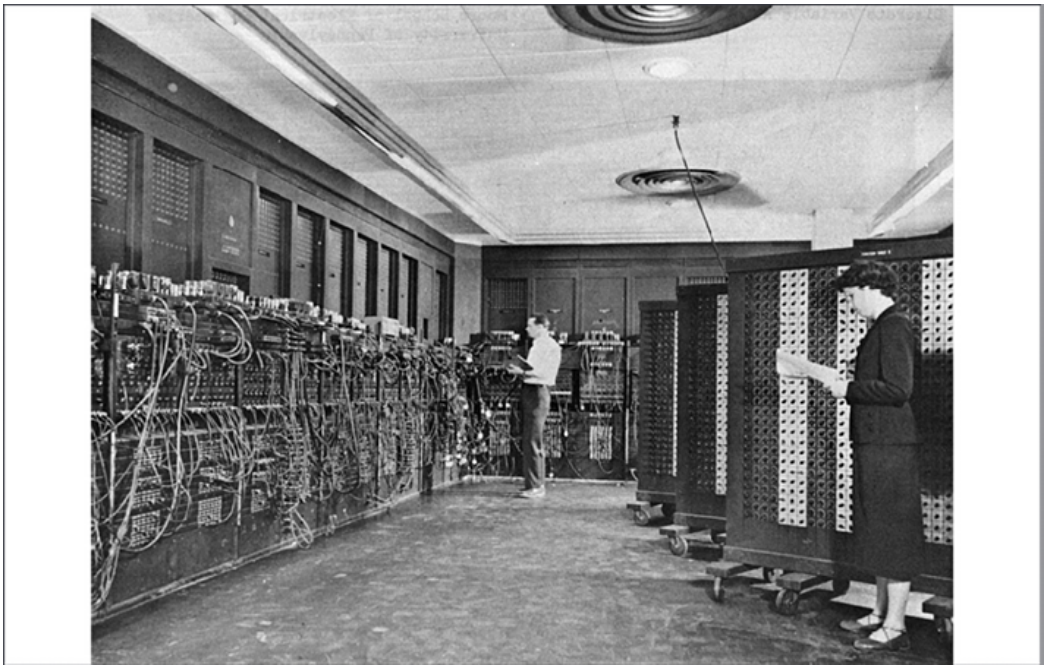


Figure 1.1: Electronic Numerical Integrator and Computer (ENIAC). Betty Snyder, one of the earliest programmers of the ENIAC, stands in front of the machine that ushered in the era of computational forecasting. (Source: Wikimedia Commons)

The computational path ENIAC forged leads directly to the massive, AI-driven systems of today, like Google's MetNet, which processes petabytes of data to predict weather with remarkable accuracy. Yet, our dependence on these complex data streams creates new vulnerabilities. During the COVID-19 pandemic, the grounding of

roughly 75% of commercial aircraft starkly revealed this fragility. The loss of atmospheric data collected by these planes (which can reduce forecast errors by up to 20%) led to a notable decrease in weather forecast accuracy.

Modern forecasting in 2026 has a wider toolbox still. It now includes **Time-Series Foundation Models** (**TSFMs**) like Amazon's Chronos-2 and Nixtla's TimeGPT, platform-native forecasting endpoints, conformal prediction intervals, and hierarchical or global models that share strength across thousands of related series. None of this retires the Prediction Paradox: a TSFM trained on 2019 demand patterns can still mistake a regime shift for noise.

This brings us to the Prediction Paradox in its modern form. We have journeyed from tea leaves to AI, building systems of incredible sophistication. Why, then, as we saw with retailers during that same pandemic, do these models still fail so catastrophically?

What is a time series?

A time series is a sequence of data points collected at successive time intervals, where the temporal ordering carries information that would be lost if we shuffled the observations.

From stock prices ticking minute by minute to seasonal sales patterns repeating yearly, this kind of data surrounds us and standard ML methods miss what the order alone tells us.

When Target's ML models treated March 2020 toilet paper purchases as independent observations that happened to occur in chronological order, they missed the fundamental temporal dependencies that

drive panic buying behavior. The models failed because they applied cross-sectional thinking to temporal data.

Before we can build systems that work reliably when patterns change, we need to establish the mathematical foundations that make time series fundamentally different from the static datasets most ML methods assume.

A **time series** is a sequence of data points measured at successive time intervals, where the temporal ordering contains crucial information about relationships and system behavior.

The temporal ordering here is the primary organizing principle, and it enables analytical techniques unavailable in cross-sectional analysis. Time series datasets describe how the world changes over time, often addressing the fundamental question of how the past influences the present and future.

More formally, we can think of a time series as observations from an underlying **data-generating process** that evolves over time:

- **Univariate time series**: $\{x_1, x_2, x_3, ..., x_t\}$ where each x_t represents a single measurement at time t
- **Multivariate time series**: Multiple related variables measured simultaneously over time

These are realizations of systematic processes where past values provide information about future values through temporal dependencies.

Formally, a time series represents a mathematical mapping from a time domain T to a measurable space S^k:

$$X: T \rightarrow S^k$$

Here, T is typically integer-valued ($T \subseteq \mathbb{N}$ for discrete time) and `k` represents the dimensionality of the variables being measured. When $k = 1$, we have a univariate time series tracking a single variable over time. When $k = 1$, we have a multivariate time series simultaneously tracking multiple related variables.

Time series data typically contains several key components that help us understand its structure:

- **Trend**: The long-term direction or movement in the data, such as gradual increase in global temperatures over decades
- **Seasonality**: Regular, predictable patterns that repeat at fixed intervals, like daily temperature cycles or weekly retail sales patterns
- **Cycles**: Longer-term fluctuations that don't have fixed periods, such as business cycles in economic data
- **Irregularity/noise**: Random fluctuations that can't be attributed to trend, seasonality, or cycles

Understanding these components helps determine the right modeling approach. For example, data with strong seasonality might benefit from explicit seasonal decomposition, while trending data might need differencing or trend-aware models.

The temporal structure enables analytical techniques. We can create lag features using past values to predict future outcomes, identify seasonal patterns that repeat at regular intervals, and detect structural breaks where underlying relationships fundamentally change. However, this temporal dependence also violates the

independence assumptions that make standard ML techniques reliable.

Time series in the Python context

Understanding time series through concrete examples reveals why these mathematical properties matter for practical analysis. Consider three representative datasets that illustrate different aspects of temporal data:

- **Google Trends Search Volume Data** provides an excellent example of multivariate time series with relative scaling challenges:

Month	**Python (Worldwide)**	**R (Worldwide)**	**Julia (Worldwide)**
2004-01-01	31	13	1
2004-02-01	29	13	1
2004-03-01	33	13	1
2004-04-01	31	14	1

2004-05-01	32	13	1

Table 1.1: Google Trends data for data science languages Python, Julia, and R

We'll go into detail about the structure of time series data in `Chapter 2`. For now, it should suffice to say that this is a multivariate time series, with columns for Python, R, and Julia. The first column is the index, a date column, and its period is the month. In cases where we have only a single variable, we speak of a univariate series. This dataset would be univariate if we had only one programming language instead of three.

Google Trends provides relative rather than absolute search volumes, normalized on a 0-100 scale where 100 represents peak popularity for the selected time period. The temporal dependencies become apparent when you consider that programming language popularity doesn't change randomly; community adoption, major releases, and educational trends create persistent patterns over months and years. This comparison has to be taken with a grain of salt because of how Google's categories are organized. You are invited to do your own comparisons.

- **Financial Time Series** illustrate the highest-stakes temporal dependencies in business applications. Daily stock prices exemplify how past values directly influence future probabilities. Unlike cross-sectional data where shuffling observations preserves information content, randomizing stock price sequences destroys the fundamental relationships that

make financial analysis possible. The closing price on Tuesday provides information about Wednesday's likely trading range, creating autocorrelation structures that sophisticated models can exploit.

- **Sensor and IoT Data** reveal the irregular timestamp challenges that separate real-world applications from academic datasets. Industrial sensors might record temperature readings every minute during normal operations but switch to continuous monitoring when anomalies are detected. Network traffic monitoring systems often record data-on-change rather than fixed intervals, creating sparse observations during quiet periods and dense measurements during peak usage.

Understanding these temporal properties reveals why the sophisticated failures we've examined aren't caused by inadequate algorithms—they result from a fundamental misunderstanding of what determines production success in temporal domains.

When do smart models fail?

Our forecasting engines had never been more advanced. Yet that very complexity created a dangerous paradox. In March 2020, as COVID-19 swept the globe, these state-of-the-art systems at major retailers began producing wildly inaccurate predictions.

At Target, the picture from the Q1 2020 earnings call was a forecasting system caught flat-footed by a regime change: apparel comp sales fell about 20 percent for the quarter (with the worst weeks in late March past 30 percent and early-April weeks past 40 percent), while electronics surged more than 45 percent, hardlines

more than 20 percent, and essentials/food/beverage roughly 20 percent in the other direction. At Amazon, the algorithms that had balanced millions of SKUs were overwhelmed: household essentials vanished from the site, and the company stopped accepting inbound shipments of anything it classified non-essential to free up warehouse capacity. These failures weren't the result of simple spreadsheet errors but of forecasting pipelines (some pure ML, others decomposition-based) that either omitted an explicit seasonality-and-anomaly step or relied on static, pre-pandemic settings. Instead of flagging the March–April 2020 spikes as one-off shocks, they folded them into their trend and seasonal baselines, yielding persistent over- and under-forecasts until the models were hurriedly re-tuned.

Within weeks, billions of dollars in forecasting infrastructure were rendered nearly useless. At Walmart, stockouts climbed across 4,700 U.S. stores faster than the planning systems could adjust, and U.S. inventory finished the quarter down nearly 5 percent year-over-year. CVS Health's prescription models, built on years of consistent patient behavior, missed the surge in anxiety-drug scripts and failed to anticipate the drop in routine refills as people skipped doctor visits. Across retail and healthcare, systems once lauded for their high validation scores and rigorous A/B tests became sources of systematic error instead of competitive advantage. In the next section, we'll examine the technical sophistication that made these failures so surprising and instructive.

These failures reveal why time series requires different thinking. The sophisticated feature engineering that works well for cross-sectional data can become actively harmful when underlying patterns change

faster than models can adapt. These organizations were deploying methods that represented the pinnacle of data science best practices circa 2020:

- **Advanced feature engineering**: Target's models incorporated over 200 engineered features, including rolling averages, seasonal decompositions, vacation effects, weather impacts, and complex interaction terms between promotional activities and product categories. These feature engineering techniques, properly implemented, are covered systematically in *Chapter 5*.
- **Ensemble methods**: Amazon's systems combined Random Forest, XGBoost, and neural network predictions using stacking approaches validated through extensive backtesting and A/B tests. Later chapters demonstrate how to implement these ensemble approaches using modern Python frameworks designed for production reliability.
- **Real-time processing**: Walmart's demand forecasting updated hourly, ingesting point-of-sale data, weather forecasts, social media sentiment, and economic indicators through streaming analytics pipelines. The monitoring and real-time adaptation strategies that could have detected these failures early are covered in *Chapter 10*.
- **Rigorous validation**: These weren't models hastily deployed without testing. They had been validated using time series cross-validation, stress-tested against historical anomalies like Black Friday and hurricane seasons, and deployed with appropriate prediction intervals and monitoring systems. However, they used validation approaches designed for stable

environments rather than the dynamic validation frameworks that we'll cover starting with *Chapter 3*.

Yet when consumer behavior shifted faster than their training pipelines could adapt, even the most advanced ML became a liability.

Understanding temporal dependencies matters because they create both opportunities and challenges. While past values can provide predictive information, they also mean that patterns can change over time: a phenomenon called **concept drift**, where the statistical relationship between inputs and the target shifts, so a model trained on yesterday's distribution gradually becomes wrong on tomorrow's. The same dependencies that make forecasting possible also make validation harder: the data-generating process itself moves with the calendar, so a holdout score collected last quarter cannot certify how the model will behave this quarter. A validation framework for time series therefore has to score both how well a model fits the past and how quickly that fit decays as the world drifts.

Before we can build systems that work reliably when patterns change, we need to establish what makes time series fundamentally different from the static datasets most ML methods assume.

Time series data violates a fundamental assumption of most ML methods: that *observations are independent and identically distributed*. When we treat temporal observations as independent data points that just happen to be ordered chronologically, we miss the dependencies between past and future values. This is why techniques that work well for cross-sectional data (like random

train/test splits or standard feature engineering) can fail with time series.

Why algorithms aren’t enough

This section reads through three lenses on the same problem: failure modes name the symptoms, gaps trace the root causes, and patterns offer the remedies.

When Virginia Tech researchers studied ML models for patient deterioration prediction in 2024, they discovered a shocking paradox: models with 94% validation accuracy failed to recognize 66% of critical patient deterioration cases in actual clinical settings.

Similar to our previous examples, this wasn’t a case of using inadequate algorithms. These were cutting-edge ensemble models combining **Long Short-Term Memory** (**LSTMs**), gradient boosting, and attention mechanisms, developed by teams with deep clinical expertise using rigorous validation protocols. Yet when deployed where lives depended on them, the most technically elaborate approaches became sources of systematic error rather than competitive advantage. The Iceberg Problem made the difference: the algorithm above the waterline was excellent, while the 95% below it (data quality, workflow integration, calibration to clinical decision thresholds) went unbuilt.

Why advanced models still fail

Understanding why technically sound models fail in time series applications helps you avoid common pitfalls in your own projects.

These failure modes show how development metrics can be misleading and what to focus on instead.

Failure mode 1: misleading validation metrics

The Promise: Healthcare AI systems achieved 94% accuracy on retrospective datasets, with excellent AUC scores and statistically significant improvements over baseline methods.

The Reality: In actual clinical deployment, these same models failed to generate adequate mortality risk scores for any synthesized emergency scenarios, remaining essentially blind to life-threatening conditions they were designed to detect.

The Root Cause: The models achieved high accuracy on historical data for statistical loss functions on clean, historical data, but clinical utility depends on entirely different factors: the ability to detect rare emergencies, integration with nursing workflows, and generating actionable insights rather than accurate predictions. This illustrates a key time series principle: validation accuracy doesn't guarantee real-world performance when temporal patterns change or when business requirements differ from statistical objectives.

Development Metrics	**Production Reality**
✓ 94 % accuracy	✓ 66% Failure Rate
✓ Excellent AUC	✓ Missed Critical Cases
✓ Statistical Significance	✓ Workflow Integration

✓ Clean Data	✓ Real-world Complexity

Table 1.2: Healthcare AI System – an example of a validation failure

This pattern repeats across industries. Financial models with impressive backtests fail during market volatility. Energy forecasting systems with high `R²` scores cannot handle renewable energy integration. The sophistication that enables excellent validation metrics often creates brittleness under real operational conditions.

Failure mode 2: integration breakdown

The Promise: Advanced enterprise software deployed with proper implementation planning.

The Reality: Nike's $100 million Q3 FY2001 loss (with a 20 percent same-day stock drop) when its newly installed i2 demand planner over-ordered Air Garnett sneakers and under-ordered Air Jordans, while the wider three-system rollout (SCM, ERP, CRM) buckled under business-process integration failures

The Root Cause: Nike attempted to implement three massive enterprise systems (SCM, ERP, and CRM) simultaneously across their global operations. The algorithms assumed they would operate within stable, well-integrated business processes, and Nike deployed them without testing against real production data. This mirrors what happens when data scientists build models in clean notebooks but don't account for production data quality issues. The legacy-system incompatibilities, data-quality issues, and organizational resistance Nike ran into could not be overcome by algorithmic sophistication alone.

Event	Outcome
2000: Deploy advanced forecasting	Months later: wildly wrong orders
Legacy and process gaps	$100M+ inventory disaster
Lesson:	Algorithm ≠ System

Table 1.3: Example for an integration breakdown

Twenty years later, the same pattern repeated at COVID-era retailers, this time with far more capable ML infrastructure behind them.

Failure mode 3: inability to adapt to change

The Promise: Models trained on years of stable historical data.

The Reality: COVID-19 rendered billions of dollars in forecasting infrastructure nearly useless within weeks as consumption patterns shifted faster than retraining pipelines could adapt.

The Root Cause: When seasonality disappeared overnight, the same feature engineering and ensemble stacking that handled normal patterns amplified errors instead of smoothing them, because every component model was wrong in the same direction at the same time.

Period	Model Behavior	Result

Before	Complex ML leads to success	Stable performance
During	Same ML becomes blind to changes	Systematic failure
After	Pattern shifts cause obsolescence	Model retraining can't keep up

Table 1.4: Example for a failure to adapt

The three fundamental gaps

These failures reveal three systematic gaps that no amount of algorithmic sophistication can bridge.

Gap 1: temporal assumption violations

Traditional ML assumes training data represents future environments accurately enough for reliable extrapolation. Time series data systematically violates this assumption through concept drift, structural breaks, and regime changes that render historical patterns misleading rather than predictive.

The Algorithm Perspective: Optimize for historical accuracy using sophisticated ensembles and feature engineering.

The Reality: Historical patterns become actively harmful when underlying systems change faster than retraining pipelines can adapt.

Gap 2: misalignment with business needs

Algorithms optimize for mathematical objectives that may have little relationship to actual business value. Minimizing **Root Mean Squared Error** (**RMSE**) or maximizing **Area Under the ROC Curve** (**AUC**) scores doesn't automatically improve inventory decisions, clinical outcomes, or operational efficiency.

The Algorithm Perspective: Achieve impressive validation metrics using state-of-the-art architectures.

The Reality: Statistical accuracy measures often inversely correlate with business utility when operational constraints and human workflows are ignored.

Gap 3: the problem with fixed models

Static optimization approaches assume that finding the best model architecture and hyperparameters solves the forecasting problem permanently. Time series applications require continuous adaptation as patterns evolve.

The Algorithm Perspective: Build sophisticated models that capture complex patterns in historical data.

The Reality: The ability to adapt quickly when patterns change matters more than the sophistication of pattern recognition within stable environments.

The pattern of hype-driven tool selection exemplifies the broader challenge. Facebook's Prophet library was promoted very broadly to teams who lacked the expertise to understand its limitations and became a default choice. The result is disastrously inaccurate

forecasts that shattered stakeholder trust—exactly the systematic failure mode that sophisticated algorithms alone cannot prevent.

Building adaptive systems

The sophisticated failures across healthcare, retail, and supply chain reveal a fundamental truth: your algorithm represents roughly 5% of what determines production success. Google's experience building ML at scale confirms this insight—as their engineering teams discovered, *most of the problems you will face are, in fact, engineering problems, not machine learning problems.*

The remaining 95% consists of problem formulation, data engineering, validation strategy, monitoring systems, and business integration. This isn't just Google's perspective—it's documented in the influential *Hidden Technical Debt in Machine Learning Systems* paper, which shows ML code as a tiny box surrounded by massive infrastructure requirements for configuration, data collection, feature extraction, monitoring, and serving.

Instead of starting with *What algorithm should I use?* successful practitioners ask four diagnostic questions that prevent the failure patterns we've examined. Each question leads to specific steps that guide technical decisions, forming a natural workflow (see Table 1.5).

Step	**Purpose**

Step	Purpose
Assess forecastability	Determines if modeling is worthwhile
Clarify decisions	Guides problem formulation and technical requirements
Plan for change	Shapes validation strategy and monitoring design
Build adaptation capability	Ensures long-term system reliability

Table 1.5: The natural workflow for building adaptive forecasting systems

Question 1: is this problem actually foreseeable?

Calculate signal-to-noise ratios before building complex models. Some series are inherently random and won't benefit from heavy algorithmic guns.

In order to check if a problem is forecastable, try the following steps:

1. **Plot your data first**: A visual pass catches obvious trends, gaps, and outliers, though many real signals only surface once a model fits them
2. **Build naive baselines**: Last value, seasonal naive, simple averages

3. **Assess improvement potential**: Can any method beat these baselines?
4. **Set realistic expectations**: Communicate inherent limitations to stakeholders
5. **Quick diagnostic**: If random walk performs as well as fancier methods, invest effort elsewhere

Question 2: what decisions will this prediction enable?

Connect technical outputs to business workflows. Ensure your model enhances rather than complicates human decision-making.

Ask these before building your model:

1. **Define the decision context**: Inventory planning? Capacity allocation? Resource scheduling?
2. **Choose the right problem type**: Forecasting, classification, anomaly detection, or regression?
3. **Understand decision timing**: Real-time responses or batch processing?
4. **Clarify interpretability needs**: Black box acceptable or explanations required?
5. **Reality check**: Models optimized for statistical accuracy often ignore operational constraints and fail despite impressive validation scores

Question 3: how will patterns change over time?

Design for concept drift detection from day one. Build systems that adapt when relationships break down rather than assuming historical stability.

Here's how to prepare for pattern shifts:

1. **Assess temporal stability**: Are patterns stationary or evolving?
2. **Design proper validation**: Use temporal cross-validation, never random splits (covered in *Chapter 3*)
3. **Plan monitoring systems**: How will you detect when models degrade?
4. **Build fallback strategies**: What simplified approaches maintain basic functionality?
5. **Key insight**: The ability to adapt quickly when patterns change matters more than initial algorithmic sophistication

Question 4: how quickly can I adapt when I'm wrong?

Prioritize retraining infrastructure over marginal accuracy improvements. Build systems that survive pattern changes rather than just optimizing for stable conditions.

Here are steps you can take to stay responsive:

1. **Design retraining pipelines**: Automated or manual model updates?
2. **Implement uncertainty quantification**: Provide prediction intervals, not just point estimates
3. **Establish performance thresholds**: When do you trigger model updates?

4. **Plan deployment workflows**: How fast can you deploy fixes when patterns shift?
5. **Success indicator**: You can deploy model updates within days of detecting performance degradation, not months

This diagnostic approach transforms potential disasters into systematic capabilities. You'll build systems that maintain stakeholder trust through honest uncertainty communication rather than false precision that inevitably disappoints.

The technical details of validation strategies, metric selection, and performance evaluation are covered systematically in *Chapter 3*.

This progression transforms the disaster patterns we've examined into systematic capabilities. You'll learn temporal validation that prevents the data leakage that makes models look good in development but fail in production. You'll master uncertainty quantification that enables risk-aware decision making rather than false precision. You'll build monitoring systems that detect concept drift before it destroys business value.

You will develop the professional judgment to communicate limitations transparently instead of overselling capabilities, earning stakeholder trust through reliable uncertainty estimates rather than impressive point forecasts that inevitably disappoint.

The journey begins with understanding your data's temporal properties and continues through to building production systems that adapt gracefully when reality violates your assumptions. In *Chapter 2*, we'll be building these capabilities systematically.

The failures at Target, Nike, and healthcare systems were preventable with the right approach to tool selection and system design. Python's mature ecosystem provides unique advantages for building such adaptive systems.

Why choose Python?

When organizations evaluate their analytics capabilities, they don't think about crystal balls or fortune telling; they think about dashboards, monitoring systems, and automated decision-making platforms. Time series analysis has evolved far beyond statistical forecasting to become the foundation for intelligent systems that power competitive advantage across every major industry. The question isn't whether to invest in temporal analytics, but how to build systems that work reliably when business decisions depend on them.

Python offers comprehensive coverage of time series techniques, from classical statistical methods to modern ML approaches, with libraries optimized for different types of problems and performance requirements. While other environments lock you into single approaches, Python enables strategic navigation across complementary libraries, each optimized for different aspects of production time series challenges.

Choosing the right tools for time series problems

Inside Python, the time series toolkit fans out across four problem types: forecasting, classification, anomaly detection, and regression.

These four overlap at the edges. A sensor classifier needs an anomaly detector for its unlabeled tail; a regression model gives you the levers a forecast needs to steer; an anomaly that recurs becomes a classifier candidate. Even so, each one foregrounds a distinct question. Forecasting asks *what comes next*. Classification asks *what kind of pattern this* is. Anomaly detection asks *does this look like anything we've seen* before. Regression asks *what drives the target and by how much*. Naming the question first is the cheapest decision in the project, and the one that determines which corner of the ecosystem the rest of the work lives in. The four sections below walk each problem type in turn, naming the production systems already running and the Python libraries the rest of this book draws on.

Forecasting

Forecasting predicts future values to drive planning. Mount Sinai Health System learned the value during COVID-19, when its demand models projected ventilator and ICU-bed needs three weeks ahead and let hospitals stage supplies before the peak hit. Pacific Gas and Electric runs the same pattern on the demand side, forecasting consumer electricity usage to set dynamic prices that shave peak load away.

The pattern repeats wherever a decision today depends on a number tomorrow. NextEra Energy fuses weather forecasts with historical generation to dispatch wind and solar hours in advance. Uber surge pricing scores thousands of concurrent series at sub-second latency.

Amazon stages inventory across fulfillment centers, Netflix places content close to viewing demand, YouTube anticipates per-user engagement, Google rides temporal query patterns to keep search relevant as topics trend, and Tesla forecasts Supercharger usage to plan network expansion.

Python's forecasting stack reflects this breadth. `statsmodels`, `sktime`, and `darts` cover the classical baselines (ARIMA, exponential smoothing, state-space models) every credible project should benchmark against. Nixtla's `statsforecast`, `neuralforecast`, and `mlforecast` suite scale those ideas to thousands of series with global and hierarchical models, which is what teams reach for once one model has to serve every store, region, or SKU. AWS `GluonTS` adds probabilistic deep learning when point predictions stop being enough.

The 2026 frontier sits in foundation models like Amazon's Chronos-2 and Nixtla's `TimeGPT`, both introduced earlier in this chapter; they often beat handcrafted baselines without any retraining at all. Prophet still earns a place when monthly retail seasonality dominates, though reaching for it by default is the failure mode the *Why advanced models still fail* section warns against.

Classification

Classification labels temporal patterns so the right action fires automatically. The catch is that the label depends on the *shape* of the recent history, which is what separates time-series classification from a snapshot classifier looking at one row.

Johns Hopkins Hospital's Early Warning System reads multivariate vital signs to flag patients whose trajectories match septic-shock precursors, alerting clinicians hours before symptoms become obvious. On factory floors, 3M classifies temperature, pressure, and vibration patterns from production sensors to stop quality defects before they reach packaging, and General Electric monitors aircraft engines, wind turbines, and power plants the same way to schedule maintenance before components fail. Meta classifies temporal activity patterns in account behavior to identify fake accounts and coordinated abuse. Financial institutions classify transaction sequences to flag fraud across millions of daily payments without trapping legitimate customers in a wall of false declines.

`sktime` and `aeon` are the entry points for time-series classifiers across the major families (shapelet, dictionary-based, interval, deep). `tslearn` covers distance-based methods like dynamic time warping for cases where samples arrive on uneven timestamps.

`tsfresh` sits one layer up, turning a raw series into hundreds of summary statistics that feed scikit-learn or gradient-boosting pipelines when the goal is interpretable importance scores rather than end-to-end deep learning.

Anomaly detection

Anomaly detection flags observations whose pattern deviates enough from baseline to warrant attention, which is what separates a working monitor from one that drowns its operators in alerts.

Cybersecurity teams use it to spot network intrusions inside enterprise traffic; cloud providers use it to predict server failures

from system-metric drift before customer impact; energy grid operators use it to catch equipment anomalies before they cascade into outages. The same lens applies to high-frequency trading, where signal extraction at the microsecond scale matters more than long-horizon planning. It also underwrites predictive maintenance whenever a defect signature has not yet been cataloged for a classifier.

The open-set problem, where the unknowns dominate the knowns, is exactly the regime where 3M's quality monitors and GE's industrial-IoT platforms earn their keep.

Python tooling runs through PyOD for the broad catalog of detectors (isolation forest, autoencoder, copula-based, distance-based) and Merlion and Kats for production-monitoring framings that bundle detection with thresholding and root-cause hooks.

`river` handles online detection where points arrive one at a time and the model never sees the whole stream at once, and darts' anomaly-detection module fits in when a team is already invested in darts for forecasting and wants to share a single residual signal between the two.

Regression

Regression explores how external factors drive a target series, which is the right framing when the question is causal or counterfactual rather than *what comes next*.

Banks use it for **Value-at-Risk** models that project potential losses against millions of transactions while leaving the audit trail regulators demand for capital allocation. Marketing teams trace how

campaigns move sales across channels and time horizons; urban planners measure how transit changes ripple into traffic and economic activity; central banks study how policy moves employment, inflation, and growth over time. Tesla's Autopilot is regression at its most demanding: cameras, radar, and ultrasonic sensors stream into models that predict how nearby vehicles will move next, and that is multivariate temporal regression even when the network underneath is a deep one.

Python here leans on scikit-learn and statsmodels for the parametric core that gives you coefficients, standard errors, and a story to tell a regulator.

`causalimpact` and `DoWhy` step in when the question is whether one series caused a change in another, providing the counterfactual machinery (synthetic controls, propensity weighting, instrumental variables) to make the answer defensible rather than a correlation dressed in temporal clothing.

In production, the four rarely live alone. GE runs predictive-maintenance pipelines that classify known failure signatures, then run anomaly detection on what is left and feed a forecasting model that schedules technician visits, all from the same telemetry stream. Uber's surge-pricing forecast feeds a classification model that decides which drivers to nudge into a busy zone. Mount Sinai's COVID forecasts drove a regression model on staff capacity, asking which interventions would change the trajectory most. Later chapters chain libraries along those seams. Python's advantage lives in multi-model systems no library was designed to own end-to-end.

Core principles for production-ready systems

Across forecasting, classification, anomaly detection, and regression a set of shared principles separates successful production systems from demos.

- **Adaptation and monitoring**: Success comes from systems that adapt as patterns evolve. Monitoring concept drift and performance degradation often matters more than the complexity of the underlying models, and Tesla, for instance, constantly updates its Autopilot models as it encounters new driving scenarios.
- **Business integration**: The most effective time series systems augment human decision-making. Doctors use early warning systems to focus their attention rather than to automate a diagnosis; traders use price predictions to inform strategy rather than to execute trades blindly. The goal is to extend expert judgment.
- **Rigorous uncertainty quantification**: Moving beyond single point predictions to provide statistically sound confidence intervals is essential for critical decisions. Financial risk models must quantify potential losses, energy trading systems must incorporate volatility forecasts, and healthcare alerts must be calibrated based on prediction uncertainty to be trustworthy.

Meeting these multifaceted requirements is where Python's ecosystem comes into its own. The ability to combine specialized libraries (statistical frameworks for interpretability, production libraries for efficiency, deep learning for complex patterns, and

uncertainty tools for reliability) is what allows organizations to build temporal intelligence that drives real-world value.

Chapter 4 translates these principles into actionable criteria. The goal is to create adaptive systems that continue working reliably when temporal patterns inevitably change. Choosing the right tool requires understanding what is available and recognizing the systematic failure patterns that derail even complex implementations. Six critical patterns emerge consistently across industries and decades.

Common pitfalls and how to avoid them

The COVID-19 retail disasters, healthcare AI blindness, and energy forecasting failures share systematic patterns that emerge when organizations apply traditional data science thinking to temporal challenges. After analyzing hundreds of production time series failures across industries, six critical failure patterns consistently emerge. Each pattern has a corresponding solution framework that successful organizations use to build adaptive systems rather than brittle technical demonstrations.

Understanding these patterns and their solutions provides a diagnostic framework for recognizing when your time series initiatives are heading toward failure, and more importantly, actionable strategies for building systems that work reliably when business decisions depend on them.

Pattern 1: static system thinking to dynamic system design

As we've seen from the retail and healthcare examples, one of the most common failure modes occurs when teams treat time series data as if it were static. Let's examine this pattern in detail.

The failure pattern: Many practitioners treat time series data as collections of independent observations that happen to be ordered chronologically. They assume patterns learned from historical data will continue indefinitely and view forecasting as extrapolating stable trends into the future. This leads to models that work well in development but fail when deployed.

The fix: *Design for regime change detection from day one.* Build systems designed to earn and maintain stakeholder trust through transparent adaptation when fundamental patterns shift, rather than waiting for performance degradation to become obvious. Implement automated concept drift detection with predefined fallback strategies when historical relationships break down.

Implementation path: *Chapter 3* will show you how to implement temporal validation that prevents data leakage and tests model robustness when patterns shift. *Chapter 10*'s monitoring frameworks provide real-time concept drift detection that triggers alerts before business impact occurs, enabling rapid response through simplified models or human intervention.

Success indicator: Your system detects and responds to pattern changes within days rather than months, maintaining reliable

performance during market disruptions while competitors' models fail systematically.

Pattern 2: optimization mindset to adaptation infrastructure

The failure pattern: Teams focus on finding the single best model with highest accuracy on historical validation data, treating model development as a one-time optimization problem. They spend weeks fine-tuning hyperparameters to achieve marginal accuracy improvements while building no infrastructure for ongoing adaptation. This mindset created financial risk management failures where tightly optimized Value-at-Risk models worked well historically but failed systematically during market volatility.

The fix: *Build retraining pipelines before optimizing algorithms***.** Invest in infrastructure that enables rapid model iteration and deployment within hours rather than pursuing algorithmic perfection that becomes obsolete when patterns change. Design automated retraining workflows with performance monitoring that can deploy updated models faster than manual processes.

Implementation path: *Chapter 5*'s scalable ML frameworks prioritize rapid experimentation and automated retraining over complex architectures. *Chapter 10*'s production workflows provide MLOps pipelines that can retrain and deploy models automatically when performance thresholds are exceeded, maintaining system reliability without manual intervention.

Success indicator: You can deploy model updates within 24-48 hours of detecting performance degradation, rather than requiring months of redevelopment when historical patterns become obsolete.

Pattern 3: point estimate fixation to uncertainty-aware decision-making

The failure pattern: Organizations provide single-point predictions that appear precise and confident, treating uncertainty as secondary consideration or post-processing step. Stakeholders make high-stakes decisions based on false precision, leading to catastrophic failures when reality falls outside narrow prediction bands. The energy grid operators who failed during extreme weather relied on point forecasts of renewable generation without accounting for prediction uncertainty during volatile conditions.

The fix: *Use conformal prediction for distribution-free uncertainty quantification.* Replace traditional Gaussian-assumption prediction intervals that rely on stable statistical properties with conformal prediction methods that provide coverage guarantees regardless of model choice or data distribution. This approach offers mathematically rigorous uncertainty estimates that remain valid even when underlying patterns change, exactly the reliability guarantee that the failures at Target, Nike, and healthcare systems lacked. Unlike traditional parametric prediction intervals that assume stable statistical properties, conformal prediction provides valid coverage even when patterns change, making it the technical cornerstone for trustworthy production systems.

Implementation path: *Chapter 8* covers conformal prediction techniques using MAPIE that can wrap any trained model to generate statistically guaranteed prediction intervals. These uncertainty estimates enable risk-aware inventory planning, capacity management, and resource allocation decisions that hold up during volatile periods.

Success indicator: Business stakeholders make explicitly risk-adjusted decisions using 90% prediction intervals with formal coverage guarantees, leading to better operational outcomes when forecasts are inevitably wrong rather than treating predictions as precise facts.

Pattern 4: independence assumptions to temporal structure respect

The Failure pattern: Teams apply standard ML validation techniques like random train-test splits and k-fold cross-validation to time series data, assuming observations are independent. This creates subtle data leakage, where models appear accurate during development but completely fail in production because they've learned to predict using information that won't be available at prediction time.

The fix: *Validate with respect to temporal order, never random splits* (the i.i.d. argument from *What is a time series?* applies). Implement time series cross-validation that strictly respects temporal structure and tests models under realistic prediction scenarios. Design validation frameworks that mirror production constraints, ensuring development metrics actually predict deployment performance.

Implementation path: *Chapter 3*'s validation frameworks provide systematic temporal cross-validation techniques that prevent data leakage while testing model robustness under realistic conditions. These approaches ensure that impressive development results translate to reliable production performance rather than providing falsely optimistic estimates.

Success indicator: Your development validation metrics accurately predict production performance within narrow margins, rather than discovering 20-40% performance drops after deployment that indicate temporal data leakage.

Pattern 5: technical metric obsession to business value alignment

The failure pattern: Organizations optimize for statistical accuracy measures like RMSE or **Mean Absolute Percentage Error** (**MAPE**) that are easy to compute and compare across models, assuming that improving technical metrics automatically improves business outcomes. This leads to investing modeling effort in improvements that have minimal business impact while ignoring the factors that actually drive operational value.

The fix: *Learn to align metrics with actual decision-making*. Instead of defaulting to RMSE or **Mean Absolute Error** (**MAE**), develop evaluation metrics that directly measure business impact (inventory holding costs, stockout penalties, capacity utilization rates) rather than generic statistical measures. Focus modeling effort on the prediction errors that cause the highest business costs. *Chapter 3*

shows you how to translate business requirements into appropriate loss functions.

Implementation path: The evaluation strategies we'll introduce in *Chapter 3* demonstrate how to translate business decisions into appropriate loss functions and validation approaches. These business-aligned metrics tie model improvements to operational outcomes that really matter.

Success indicator: A 10% improvement in your custom business metrics translates to measurable cost savings or revenue increases, while improvements in statistical metrics correlate strongly with operational outcomes.

Pattern 6: model artifact focus to production system design

The failure pattern: Teams view model development as complete when they have a trained algorithm that performs well on validation data, treating deployment, monitoring, and maintenance as separate concerns handled by other teams. Models that work excellently in notebooks fail when deployed because they can't handle production data quality issues, latency requirements, or integration constraints.

The fix: *Plan for deployment from the start.* Thinking about how your model will be used helps you make better design decisions. This book teaches deployment-friendly practices throughout, culminating in *Chapter 10*'s hands-on MLOps workflows.

Implementation path: *Chapter 10*'s production workflows provide complete MLOps frameworks designed specifically for time series

applications, including model registries, automated deployment pipelines, and monitoring systems that detect operational issues before they impact business decisions.

Success Indicator: Your models maintain stable performance in production environments with real data quality issues, latency constraints, and integration requirements, rather than only functioning in controlled development environments.

Summary

The failures at Target, Amazon, and Nike were preventable with the right understanding of how to build adaptive systems for temporal data. The six failure patterns we've identified and their corresponding solutions provide both the conceptual foundation and practical framework needed to build systems that work reliably when patterns change.

These common mistakes provide a learning roadmap: you'll master temporal validation that respects time structure, develop models that adapt when patterns change, implement uncertainty quantification for better decisions, align metrics with business goals, and build deployment-ready systems from the start.

The next chapter begins building these capabilities with the practical foundations of data engineering and visualization, turning the conceptual framework into working systems that handle the temporal challenges that break standard ML approaches.

Let's begin.

Further Reading

- Target Q1 2020 earnings discussion at https://corporate.target.com/getmedia/56e9c6ca-8a04-4a31-983f-e95fdbb70b32/TGT-USQ_Transcript_2020-05-20.pdf, with the call transcript at https://www.fool.com/earnings/call-transcripts/2020/05/20/target-corp-tgt-q1-2020-earnings-call-transcript.aspx
- Bezos's 2020 letter to shareholders at https://www.aboutamazon.com/news/company-news/2020-letter-to-shareholders describes the operational pivot
- The 2021 letter at https://www.aboutamazon.com/news/company-news/2021-letter-to-shareholders documents the inventory placement problems that followed
- Walmart Q1 FY21 earnings release filed with the SEC at https://www.sec.gov/Archives/edgar/data/104169/000010416920000016/earningsrelease-4302020.htm, with stockout coverage in Supply Chain Dive at https://www.supplychaindive.com/news/Walmart-coronavirus-stockouts-e-commerce-earnings/583689/
- CVS Health Q1 2020 results press release at https://cvshealth.com/news-and-insights/press-releases/cvs-health-reports-results-2020-q1 details the surge in 90-day prescriptions and early refills, and the operational response

- Gama et al. 2014, *A Survey on Concept Drift Adaptation*, ACM Computing Surveys, https://doi.org/10.1145/2523813
- Computerworld's contemporary coverage at https://www.computerworld.com/article/1356579/nike-blames-financial-snag-on-supply-chain-project.html
- CIO's later retrospective at https://www.cio.com/article/264637/enterprise-resource-planning-nike-rebounds-how-nike-recovered-from-its-supply-chain-disaster.html documents the i2 deployment, the missed earnings, and the recovery

Get this book's PDF version and more

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

2

Preparing and Visualizing Time Series Data

Before building a reliable forecast, you must first learn to diagnose your data. The defining characteristic of a time series is **temporal dependence**: the value of an observation today is often a function of its value yesterday. We've discussed in *Chapter 1* that many standard machine learning models are built on the assumption that data points are independent. Applying them to time series often leads to failure because this core assumption is fundamentally broken; shuffling the rows of a time series destroys the very patterns that make forecasting possible.

The chapter begins by exploring the mathematical foundations of time series, teaching you to probe your data for core properties like autocorrelation and to test for stationarity—a critical prerequisite for many classical models. From there, we will equip you with the skills to build a reliable data pipeline, mastering temporal indexing, quality checks for missing data, and intelligent resampling decisions. With clean, structured data, you will then learn how to discover hidden patterns like seasonality and cycles using a suite of specialized visualizations. Finally, you will learn how to perform a crucial forecastability assessment to determine if a series has enough

predictable signal to justify the modeling effort or if it is primarily random noise.

In this chapter we will look at the following topics:

- Mathematical foundations for data understanding
- Building a time series data pipeline
- Pattern discovery through visualization
- Forecastability assessment

We begin with the core principles that make time series analysis a unique discipline.

Technical requirements

The dataset used in this chapter can be accessed here: https://archive.ics.uci.edu/dataset/275/bike+sharing+dataset.

The chapter's code uses `pandas`, `numpy`, `matplotlib`, `scipy`, `statsmodels`, `yfinance` (apart from standard library modules). Install the third-party packages once with:

```
pip install pandas numpy matplotlib scipy statsmodels yfinance
```

The code targets `pandas 2.0+` and `statsmodels 0.14+`. We stick with `pandas` throughout the chapter, but readers working with very large time series may want to look at Polars, which has gained traction for its lazy evaluation and faster *groupby* on hourly or sub-hourly data; the diagnostics in this chapter port over without changing in spirit.

If you follow the instructions on the book's companion repository on GitHub, you can set up an environment that has everything installed: https://github.com/PacktPublishing/Machine-Learning-for-Time-Series-with-Python-Second-Edition.

Mathematical foundations for data understanding

Why does time series require its own toolkit? Because the core assumption of many standard models—that data points are independent—is fundamentally broken. A time series is a sequence of dependent observations. The value today is often a function of the value yesterday. This section proves why that matters, using hands-on diagnostics to reveal the hidden structure in your data.

Temporal dependencies and autocorrelation

Time series data breaks the fundamental assumption underlying most machine learning: that observations are independent. Unlike cross-sectional datasets where you can shuffle rows without losing information, time series observations are connected through **temporal dependence**. Today's values influence tomorrow's, creating both forecasting opportunities and systematic vulnerabilities when patterns shift.

This temporal dependence distinguishes time series from traditional **Machine Learning** (**ML**) and explains why standard approaches

often fail. Understanding this concept is fundamental before we explore classical models in *Chapter 3*, navigate the modern ecosystem in *Chapter 4*, or engineer temporal features in *Chapter 5*.

That was about temporal dependence, but what is autocorrelation? Autocorrelation measures how correlated a time series is with its own past values. To calculate the correlation between two variables, you use this formula:

$$\rho_k = Corr(Y_t, Y_{t-k}) = \frac{\gamma_k}{\gamma_0}$$

Where:

- ρ_k is the autocorrelation at lag k
- ρ_k is the autocovariance at lag k
- γ_0 is the variance of the series

When $\rho_1 = 0.8$, today's value strongly relates to yesterday's—excellent for forecasting. When $\rho_0 \approx 0$ for all lags $k>0$, you have white noise—mathematically unpredictable and not worth modeling.

Proving temporal structure exists

Let's demonstrate temporal dependence using a concrete example applied to the Apple stock. Let's first fetch data using the Yahoo Finance library `yfinance`:

```
import yfinance as yf
def fetch_aapl():
    # Fetch Apple stock data
```

```
    aapl = yf.download(
        'AAPL', start='2020-01-01', end='2023-12-31', progress=False)
    aapl['returns'] = aapl['Close'].pct_change() * 100
    return aapl
```

In this code, the returns are the percentage change in price from one day to the next.

Next, we'll compare Apple stock returns with their randomly shuffled version. The contrast of structure reveals the hidden structure that enables financial forecasting:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf
aapl = fetch_aapl()
aapl_returns = aapl['returns'].dropna() # drop leading NaN at the fir
# Calculate autocorrelations for original and shuffled data
original_autocorr = aapl_returns.autocorr(lag=1)
shuffled_returns = aapl_returns.sample(
    frac=1, random_state=42).reset_index(drop=True)
shuffled_autocorr = shuffled_returns.autocorr(lag=1)
print(f"Original lag-1 autocorrelation: {original_autocorr:.4f}")
print(f"Shuffled lag-1 autocorrelation: {shuffled_autocorr:.4f}")
print(f"Structure destroyed by shuffling: {abs(original_autocorr - sh
# Visual comparison
fig, axes = plt.subplots(1, 2, figsize=(14, 5))
plot_acf(aapl_returns, lags=20, ax=axes[0],
         title='ACF: Original Apple Returns')
plot_acf(shuffled_returns, lags=20, ax=axes[1],
         title='ACF: Shuffled Returns (Structure Lost)')
plt.tight_layout()
plt.show()
```

You should see this output:

```
Original lag-1 autocorrelation: -0.1332
Shuffled lag-1 autocorrelation: 0.0002
Structure destroyed by shuffling: 0.1335
```

We also visualized the comparison using the code:

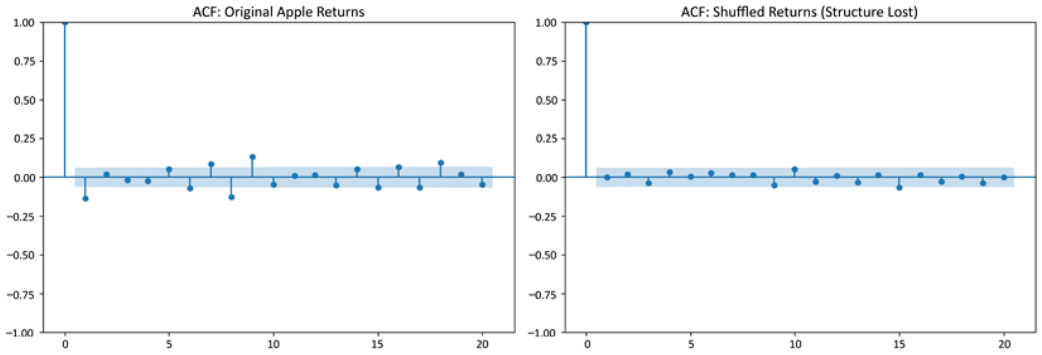


Figure 2.1: Two side-by-side autocorrelation function plots

The left panel shows original Apple returns with visible pattern/structure in autocorrelations while the right shows shuffled returns with autocorrelations near zero (no structure).

You'll notice the original returns show small but statistically significant autocorrelation patterns, while the shuffled data's autocorrelation drops to nearly zero. The ACF plots confirm this visually: the left plot shows structure outside the confidence bands, while the right plot shows only noise. We've statistically proven that temporal order contains predictive information.

This structure is the statistical relationship between an observation and its predecessors and is what enables financial modeling. Forecasting models are designed to learn this very relationship. By shuffling the data, we break the temporal link, making it impossible for a model to learn how `Y(t-1)` influences `Y(t)`.

When we shuffle the data, we destroy the very relationships that make forecasting possible—this is why time series requires its own specialized approaches, which we'll explore systematically in *Chapters 3* through *5*.

Interpreting autocorrelation patterns for modeling strategies

Different autocorrelation patterns tell you which modeling approaches will work best. Understanding these patterns before diving into implementation saves weeks of trial and error. To make these insights actionable, we can automate the interpretation of autocorrelation patterns. The following utility scans a series, highlights significant lags, and suggests suitable modeling strategies based on the detected structure:

```
def analyze_autocorrelation_patterns(
    series, series_name, max_lags=20, alpha=0.05):
    """Simple ACF scan + text recommendations."""
    n = len(series)
    threshold = 1.96 / np.sqrt(n)
    acfs = [series.autocorr(lag=k) for k in range(1, max_lags+1)]
    sig = [(k, acf) for k,
           acf in enumerate(acfs, 1) if abs(acf) > threshold]
    # Build a clean summary
    summary = {
        "name": series_name,
        "n_lags_tested": max_lags,
        "n_significant": len(sig),
        "top_lags": sig[:3]
    }
    # Map to clear interpretations
    interpretations = []
    for lag, corr in summary["top_lags"]:
        if lag == 1:
```

```
            text = "Lag-1 momentum → include lag-1 as feature"
        elif lag in (5, 7):
            text = f"Lag-{lag} seasonality → try weekly/seasonal mode
        else:
            text = f"Lag-{lag} cycle → craft custom lag-{lag} feature
        interpretations.append(f"  • Lag {lag}: ρ={corr:.3f} → {text}
    # Decide strategy (no overlap of cases)
    if summary["n_significant"] == 0:
        strategy = "No clear autocorrelation →
                    focus on external/exogenous predictors."
    elif summary["n_significant"] <= 3:
        strategy = "Moderate autocorrelation →
                    a classical ARIMA model is probably sufficient."
    else:
        strategy = "Rich autocorrelation →
                    consider more flexible models or ML approaches."
    # Print out neatly
    print(f"\nAutocorrelation Analysis: {summary['name']}")
    print(f"  Tested lags 1–{summary['n_lags_tested']}, "
          f" found {summary['n_significant']} significant.")
    print("\nKey patterns:")
    print("\n".join(interpretations) if interpretations else "  • Non
    print(f"\nRecommended strategy:\n  → {strategy}\n")
    return summary, interpretations, strategy
# Apply to Apple returns
apple_patterns = analyze_autocorrelation_patterns(
    aapl_returns, "Apple Daily Returns")
```

This analysis directly informs your modeling strategy. Strong `lag-1` correlations suggest including lagged features when we build ML models in *Chapter 5*. Weekly patterns indicate seasonal models from *Chapter 3* might be effective. No significant patterns suggest focusing on external variables and uncertainty quantification.

We should see this output:

```
►Lag-1 ACF: Original = -0.1332, Shuffled = 0.0002 (Δ=0.1335)
Autocorrelation Analysis: Apple Daily Returns
```

```
  Tested lags 1–20, found 9 significant.
Key patterns:
  • Lag 1: ρ=-0.133 → Lag-1 momentum → include lag-1 as feature
  • Lag 6: ρ=-0.071 → Lag-6 cycle → craft custom lag-6 features
  • Lag 7: ρ=0.087 → Lag-7 seasonality → try weekly/seasonal model
Recommended strategy:
  → Rich autocorrelation → consider more flexible models or ML approac
```

Keep in mind that this is only a first pass for exploratory analysis. Be mindful that it's a screening tool, not a substitute for the richer identification and diagnostic steps in a full time-series workflow.



Author's insight: Remember that a simple lag-by-lag ACF check is just a first glance; scanning 20 lags at 5% gives you one false signal on average, so consider a Bonferroni correction or a single Ljung–Box Q-test instead. Also, ACF spikes mix direct and indirect effects. And if your series still has trend, level shifts or seasonality, those will warp every autocorrelation, so always difference or detrend (and strip out any clear seasonal cycles) before you run your ACF scan. Finally, if you see slowly decaying correlations or volatility clusters, a basic ARMA won't cut it. In this case, you'll need long-memory, GARCH or even machine-learning approaches for those rich patterns. Don't worry! We have you covered.

Tools for autocorrelation analysis

Your autocorrelation analysis approach depends on context and requirements. For quick exploration, pandas' `autocorr()` method provides immediate lag-specific insights. When you need statistical rigor for stakeholder presentations, statsmodels' `plot_acf()` includes confidence intervals and significance testing. For production monitoring systems, `numpy`'s optimized calculations enable real-time analysis of streaming data. *Chapter 4* will provide a comprehensive framework for choosing between these and other ecosystem options.

Understanding temporal dependence prevented some organizations from costly failures during COVID-19. Retail systems that could detect when established autocorrelation patterns suddenly broke were able to adapt. Those that couldn't—systems built assuming yesterday's patterns would continue indefinitely—failed catastrophically when consumer behavior shifted overnight.

This isn't just mathematical theory. Temporal dependence is the foundation that determines whether your forecasting system will be a competitive advantage or an expensive liability. Master this diagnostic skill, and you'll approach time series problems with the strategic mindset needed to build systems that work reliably when business decisions depend on them.

The next section explores another critical mathematical requirement: stationarity, the property that makes learning from historical data mathematically valid.

Stationarity testing and interpretation

Many time series models operate under a critical assumption: that the statistical properties of the series do not change over time. This

property is called **stationarity**. A stationary series has a *constant mean, constant variance, and constant autocorrelation*.

This assumption is foundational for classical linear models like ARIMA, which we will explore in *Chapter 3*. These models estimate fixed coefficients based on historical data, and those estimates only make sense if the underlying process isn't changing. While many modern machine learning models (*Chapter 5*) don't strictly require stationarity, transforming the data to be stationary often leads to better and more stable performance.

In mathematical terms, a time series Y_t is weakly stationary when:

- $E[Y_t] = \mu$ (constant mean)
- $Var(Y_t) = \sigma^2$ (constant variance),
- $\text{Cov}(Y_t, Y_{t-k}) = \gamma_k$ (autocovariance depends only on lag k)

These conditions enable learning from historical data. In simple terms, the series is stable and predictable in its behavior, even if it's noisy.



Sometimes, a series may appear to have a constant mean, but its volatility changes over time. You can see hints of this in the rolling standard deviation plots, where the line is not perfectly flat. This phenomenon is called heteroscedasticity.

While a simple visual check is a good start, many financial time series (like stock returns) exhibit periods of high and low volatility even when their mean is stable around zero. This pattern of time-varying

variance violates the stationarity assumption and can't be captured by standard models. To handle it, specialized models like ARCH and GARCH are used to model the variance itself, a more advanced topic we revisit in *Chapter 8* when we combine GARCH with conformal prediction for adaptive prediction intervals. For now, recognizing that volatility can be nonconstant is a key diagnostic step.

Why does this stationarity matter? Models rely on learning patterns from the past to predict the future. If the fundamental properties of the series are constantly changing, what is learned from the past becomes irrelevant for the future. A model trained on a non-stationary series is building on shifting sand. Its assumptions will break, and its forecasts will be unreliable.

It's crucial to understand that stationarity doesn't guarantee a series is forecastable. A perfectly stationary series, like white noise, has no discernible patterns to model. Instead, think of stationarity as a necessary property that makes the patterns you do find reliable over time. If the statistical properties are stable, the patterns learned from the past are more likely to apply to the future, making your model's assumptions valid.

Understanding stationarity is fundamental before exploring classical models in *Chapter 3*, choosing appropriate libraries in *Chapter 4*, or engineering temporal features in *Chapter 5*.

Visualizing stationarity

Understanding stationarity is easier when you can see it, so let's make this concept visual. We'll compare Apple's raw stock price (a non-stationary series) with its daily returns (which we suspect are stationary). The difference is immediately obvious.

```
import pandas as pd
import matplotlib.pyplot as plt
# Use the same Apple data from the previous section
aapl_df = fetch_aapl()
aapl_df.dropna(inplace=True)
def plot_rolling_stats(series, title_suffix):
    """Plots a series with its rolling mean and standard deviation."""
    rolling_mean = series.rolling(window=30).mean()
    rolling_std = series.rolling(window=30).std()
    plt.figure(figsize=(12, 6))
    plt.plot(series, color='blue', label='Original Series')
    plt.plot(rolling_mean, color='red', label='Rolling Mean')
    plt.plot(rolling_std, color='black', label='Rolling Std Dev')
    plt.legend(loc='best')
    plt.title(f'Rolling Mean & Standard Deviation: {title_suffix}')
    plt.show()
# Plot for non-stationary price level
plot_rolling_stats(aapl_df['Close'], 'Apple Stock Price')
# Plot for (likely) stationary returns
plot_rolling_stats(aapl_df['returns'], 'Apple Stock Returns')
```

We should see a plot of daily closing prices over time:

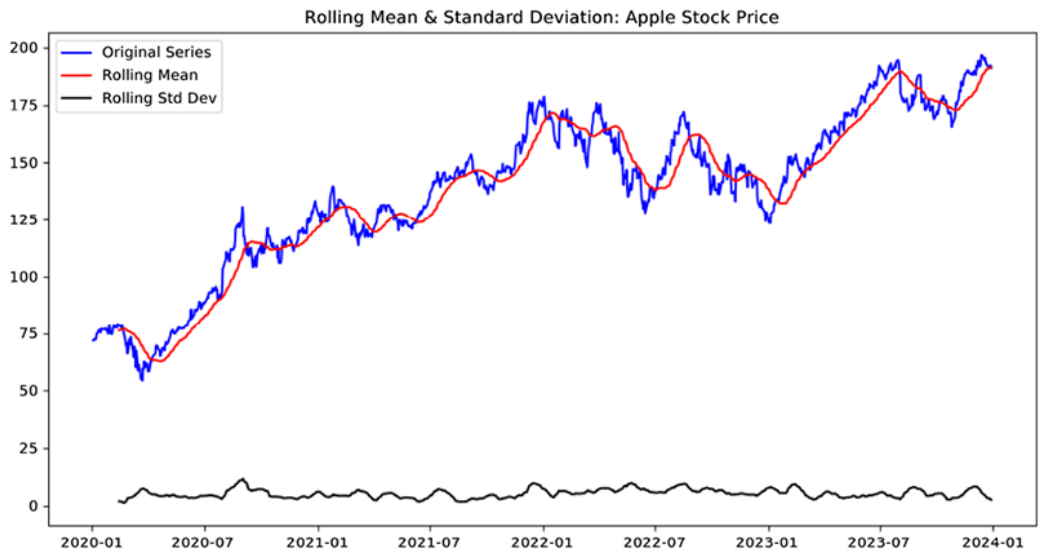


Figure 2.2: Rolling mean and standard deviation of daily closing prices over time

The price chart shows a clear upward trend. Its mean value is not constant; it's higher in 2023 than in 2020. The volatility (variance) also changes over time.

The second plot is of the daily returns.

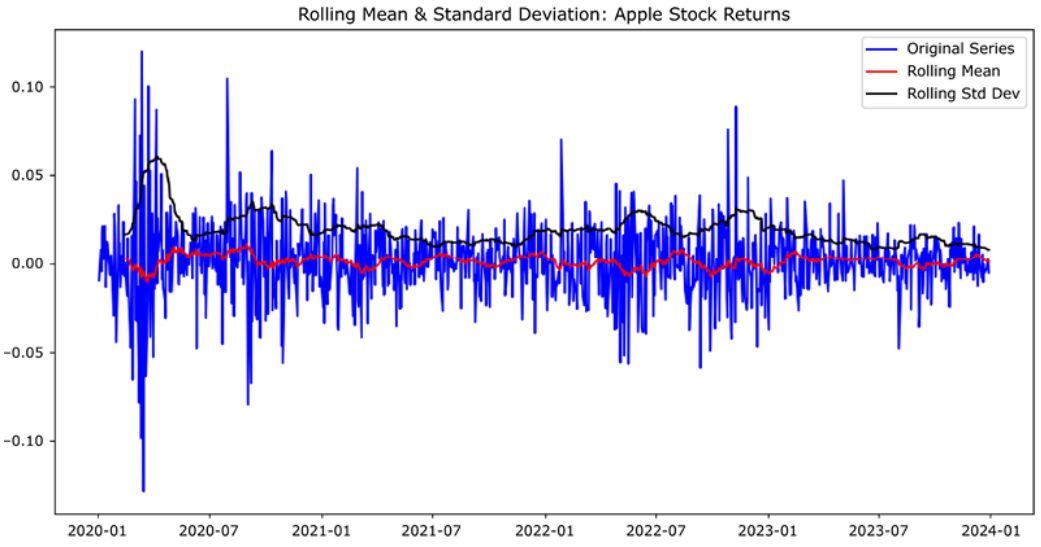


Figure 2.3: Rolling mean and standard deviation of daily returns

In contrast to the closing price, the returns chart appears much more stable. It hovers around a mean of zero and its variance, while not perfectly constant, is far more consistent than that of the price series.

Statistical stationarity tests

We will now formalize this visual intuition. First, we'll plot the data with a rolling mean and standard deviation. This helps visualize the changing statistics. Then, we will use two formal statistical tests:

- **Augmented Dickey-Fuller (ADF) Test**: The null hypothesis is that the series is non-stationary. A low p-value (e.g., < 0.05) allows us to reject this null hypothesis and conclude the series is stationary.
- **Kwiatkowski-Phillips-Schmidt-Shin (KPSS) Test**: The null hypothesis is that the series is stationary. A low p-value here suggests the series is not stationary.

Using both tests gives us a more confident diagnosis. Their nulls are flipped, so the decisions you make depend on whether they agree:

ADF says	**KPSS says**	**Read it as**	**What to do**
Stationary	Stationary	Stationary	Model the series as is.
Nonstationary	Non-stationary	Has a unit root	Difference once, re-test.
Stationary	Non-	Trend-	Detrend or run KPSS

	stationary	stationary	with regression='ct' (constant + trend).
Non-stationary	Stationary	Heavy persistence with stable mean	Inspect for heteroscedasticity; consider a log or Box–Cox transform before differencing.

Table 2.1: Comparing the ADF and KPSS tests

That last row is also where the KPSS *regression* parameter earns its keep. The default *regression='c'* tests stationarity around a constant mean; *regression='ct'* tests stationarity around a deterministic trend. If your series is clearly trending but you suspect the deviations from the trend are well-behaved, run KPSS with *regression='ct'* before reaching for differencing.

Let's look at the code for the process we outlined here:

```
from statsmodels.tsa.stattools import adfuller, kpss
def run_stationarity_tests(series, series_name=""):
    """Runs and interprets ADF and KPSS tests."""
    print(f"--- Stationarity Tests for {series_name} ---")
    # ADF Test
    adf_result = adfuller(series)
    print(f'ADF Statistic: {adf_result[0]:.4f}')
    print(f'p-value: {adf_result[1]:.4f}')
    print("ADF Conclusion: The series is likely NON-STATIONARY."
          if adf_result[1] > 0.05
          else "ADF Conclusion: The series is likely STATIONARY.")
    print("\n")
    # KPSS Test
    # Note: The KPSS test returns a p-value and the test statistic.
    # The interpretation is reversed from ADF.
```

```
    kpss_result = kpss(series, regression='c', nlags="auto")
    print(f'KPSS Statistic: {kpss_result[0]:.4f}')
    print(f'p-value: {kpss_result[1]:.4f}')
    # The critical values are for 5% and 1% levels.
    # We check if the stat exceeds them.
    print("KPSS Conclusion: The series is likely NON-STATIONARY."
          if kpss_result[1] < 0.05
          else "KPSS Conclusion: The series is likely STATIONARY.")
    print("-" * 30 + "\n")
# Test the price series (non-stationary)
run_stationarity_tests(aapl_df['Close'], "Apple Stock Price")
# Test the returns series (stationary)
run_stationarity_tests(aapl_df['returns'], "Apple Stock Returns")
```

We should see an output like this:

```
--- Stationarity Tests for Apple Stock Price ---
ADF Statistic: -1.5120
p-value: 0.5276
ADF Conclusion: The series is likely NON-STATIONARY.
KPSS Statistic: 3.9359
p-value: 0.0100
KPSS Conclusion: The series is likely NON-STATIONARY.
------------------------------
--- Stationarity Tests for Apple Stock Returns ---
ADF Statistic: -10.1632
p-value: 0.0000
ADF Conclusion: The series is likely STATIONARY.
KPSS Statistic: 0.1490
p-value: 0.1000
KPSS Conclusion: The series is likely STATIONARY.
```

For quick visual assessment, pandas provides rolling statistics, while statsmodels offers comprehensive statistical tests for formal stationarity validation.

It’s also important for us to interpret the test results; let’s see that here.

- **For Apple Stock Price:** The rolling mean clearly trends upward. The ADF test will yield a high p-value (e.g., > 0.05), failing to reject non-stationarity. The KPSS test will yield a low p-value (< 0.05), rejecting its null hypothesis of stationarity. Both tests agree: the price series is non-stationary.
- **For Apple Stock Returns:** The rolling mean and standard deviation are much more stable. The ADF test will give a very low p-value (< 0.01), confirming stationarity. The KPSS test will give a high p-value (> 0.10), failing to reject stationarity. Again, the tests agree.

Achieving stationarity

The natural approach here would be to work with the returns, especially since they are already stationary, but for the sake of the example, let's illustrate how to make the closing price stationary. The most common way to make a non-stationary series stationary is **differencing**. This technique computes the difference between consecutive observations. It is an effective way to remove trends and stabilize the mean. Pandas makes this trivial with the `.diff()` method. In fact, calculating returns is a form of differencing.

Why does differencing work? It trades a *level-dependent* series (where each value carries the trend forward) for a *change-dependent* series (where each value is the increment between adjacent points). The increments are far more likely to have a constant mean and variance than the levels, which is exactly the property the stationarity tests are looking for.

Let's apply a first-order difference to the price series and retest.

```
# Apply first-order differencing
aapl_df['price_diff'] = aapl_df['Close'].diff()
# diff() creates one NaN; preserving the original
aapl_df['price_diff'] = aapl_df['price_diff'].dropna()
# Visualize the differenced series
plot_rolling_stats(price_diff, 'Differenced Apple Stock Price')
# Re-run the stationarity tests on the differenced data
run_stationarity_tests(price_diff, "Differenced Apple Stock Price")
```

We should get this plot:

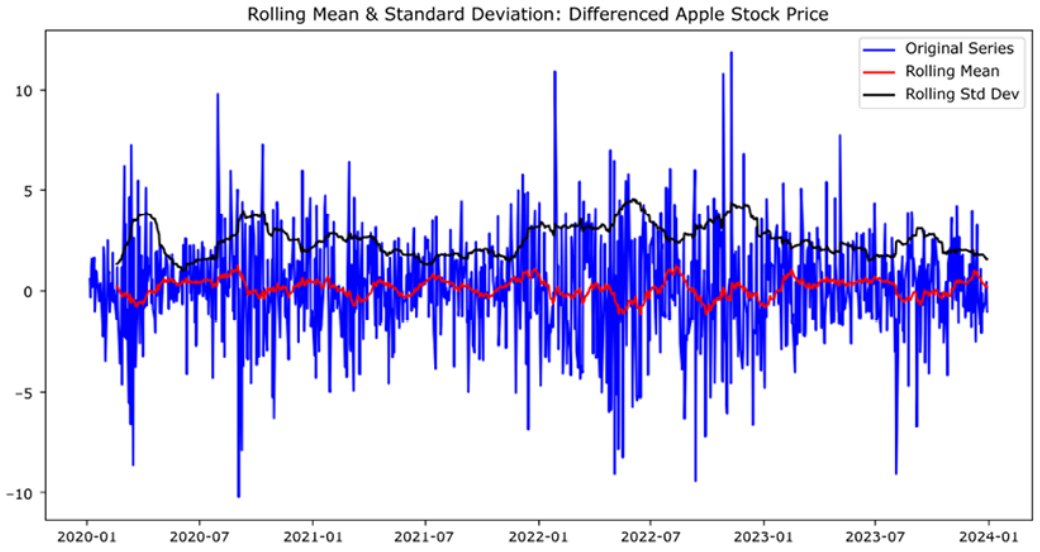


Figure 2.4: Rolling Mean and Standard Deviation: Differenced Apple Stock Price

Please note that log returns are often preferable (or sometimes percentage changes), so you could change this to:

```
aapl_df['log_return'] = np.log(aapl_df['Close']).diff()
log_returns = aapl_df['log_return'].dropna()
```

The result is clear. The differenced price series looks visually identical to the returns series. The statistical tests will now confirm

that this differenced series is stationary.

We should see this output in fact:

```
--- Stationarity Tests for Differenced Apple Stock Price ---
ADF Statistic: -32.9903
p-value: 0.0000
ADF Conclusion: The series is likely STATIONARY.
KPSS Statistic: 0.0539
p-value: 0.1000
KPSS Conclusion: The series is likely STATIONARY.
```

Both tests now agree, in the right direction: ADF rejects the unit-root null and KPSS fails to reject the stationarity null. That is the configuration the decision matrix calls *Stationary; model the series as is.*

We have successfully transformed our data into a state that is suitable for modeling. The key takeaway is not just that returns are stationary, but why we model them. We don't model the price directly with classical methods because its mean and variance are not constant. We model the stationary returns because we are interested in the price changes, which exhibit more stable statistical properties.

Stationarity is not an abstract goal. It is a practical check to ensure your model's assumptions are not violated from the start. If your model requires stationary data, you must test for it and, if necessary, induce it. Ignoring this step is like building a house on an unstable foundation. The structure may look fine at first, but it is destined to fail.

Stabilizing variance with log and Box-Cox transformations

Differencing fixes a moving mean, not a fanning variance. If your series grows exponentially or its volatility scales with its level — think weekly retail demand that swings ±50 units in January and ±5,000 in December, or a stock price that wiggles in dollars at $10 and in tens of dollars at $200 — you have *heteroscedasticity*, and differencing alone leaves it untouched. The fix is a variance-stabilizing transform. The log transform is the simple option for strictly positive data: it pulls the right tail in and turns multiplicative growth into additive growth. The Box–Cox transform generalizes this with a single shape parameter λ that the data picks for you ($k = 1$ reduces to log, $\lambda = 1$ leaves the data alone).

```
import numpy as np
from scipy import stats
# Log transform: handles strictly positive series with multiplicative
log_close = np.log(aapl_df['Close'])
# Box-Cox: lets the data choose the shape parameter.
boxcox_close, lam = stats.boxcox(aapl_df['Close'])
print(f"Box-Cox lambda: {lam:.3f}")  # AAPL 2020-2023 lands around 1.
# Now difference the transformed series to address mean and variance
log_returns = log_close.diff().dropna()
```

The pattern is: transform first to stabilize variance, then difference to stabilize the mean. Re-run the ADF + KPSS pair on the result; if your earlier diagnosis sat in the awkward *ADF says non-stationary, KPSS says stationary* cell of the decision matrix, this is usually the move that resolves it.

Signals from decomposition

A time series is often a composite of several underlying patterns working in concert. While a raw plot can hint at these patterns, manually identifying, and quantifying them is difficult and imprecise. Time series decomposition is a foundational technique that addresses this by separating a series into its constituent components, allowing us to study them in isolation.

This process is not merely a diagnostic step; it is a critical part of the modeling workflow that provides several key advantages:

- **Enhanced understanding:** By isolating components, you gain a clearer picture of the data's underlying structure—its long-term direction, cyclical patterns, and random variations.
- **Improved forecasting accuracy:** Each component can be modeled and forecasted separately. Combining these individual forecasts often yields more precise and reliable predictions than modeling the raw series directly.
- **Anomaly detection:** By removing the predictable trend and seasonal patterns, anomalies and outliers become much easier to spot within the residual component.
- **Seasonal adjustment:** In many fields, like economics, removing seasonal effects is necessary to analyze the true underlying trend. Decomposition is the primary method for this seasonal adjustment.

Typically, a time series is broken down into three parts:

- **Trend (γ_0):** The long-term progression or underlying direction of the series. It captures sustained increases, decreases, or

stability over a long duration.

- **Seasonality (Y_0):** A repeating, cyclical pattern of a fixed and known frequency. Examples include daily temperature cycles, weekly retail sales patterns, or yearly electricity demand.
- **Residuals (R_t):** Also known as noise or the irregular component, this is what remains after removing the trend and seasonality. In a good decomposition, the residuals should be random, with no discernible pattern.

Why the additive vs. multiplicative choice matters

These components can be combined in two primary ways:

- **Additive Model:** Assumes `Series = Trend + Seasonality + Residuals` ($Y_t = T_t + S_t + R_t$). This model is appropriate when the magnitude of the seasonal fluctuations is roughly constant over time, regardless of the level of the trend.
- **Multiplicative Model:** Assumes `Series = Trend * Seasonality * Residuals` ($Y_t = T_t * S_t * R_t$). This is suitable when the seasonal fluctuations grow or shrink in proportion to the trend level—a common pattern in sales data where vacation demand scales with overall growth.

Choosing the right model is critical. If you use an additive model on a series that is truly multiplicative, the residuals will be systematically biased. During high-trend periods, the seasonal effect will be underestimated, leading to large positive residuals that could be mistaken for anomalies. This corrupts the very component used for outlier detection, making your analysis unreliable. When in

doubt, visualizing the series can often reveal whether seasonal swings are stable or growing with the trend.

As discussed in *Chapter 1*, this static view of patterns was a key reason many forecasting models at major retailers failed during the massive demand shifts of the COVID-19 pandemic. When a structural break occurs, historical trends and seasonalities can become instantly obsolete.

To build more resilient systems, decomposition must be treated as a dynamic process. Advanced pipelines often involve on-the-fly re-estimation, such as rerunning STL on a sliding window of recent data to adapt to evolving patterns. Furthermore, the residuals themselves become a valuable feature; feeding the magnitude of the residual ($|R_t|$) or an "anomaly flag" back into a machine learning model allows it to learn how to react differently when shocks occur, making the entire system more adaptive to unforeseen events.

Practical decomposition in Python

To explore decomposition, we need a dataset with strong seasonal patterns. We will use a public dataset of hourly bike rentals from the UCI Machine Learning Repository. We expect to see clear daily (commute times) and weekly patterns.

We will explore two primary decomposition methods from the statsmodels library:

- `seasonal_decompose`: A classic, straightforward method based on moving averages. It's a great starting point but can be rigid, as it assumes a fixed, unchanging seasonal pattern.

- **Seasonal and Trend decomposition using Loess (STL)**: A more modern and robust method that uses local regression (LOESS) to perform the smoothing. Its key advantage is that it allows the seasonal component to change over time, making it far more flexible and less sensitive to outliers.

Let's apply these methods to our bike-sharing data. But before we do, let's define another data function:

```
import io
import zipfile
import requests
import pandas as pd
def fetch_bike_sharing():
    """Download Bike Sharing Data."""
    url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/
    r = requests.get(url)
    z = zipfile.ZipFile(io.BytesIO(r.content))
    df_bike = pd.read_csv(z.open('hour.csv'))
    df_bike['datetime'] = pd.to_datetime(
        df_bike['dteday'] + ' ' + df_bike['hr'].astype(str) + ':00:00
    df_bike.set_index('datetime', inplace=True)
    return df_bike
```

Let's fetch the bike data and decompose it:

```
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose, STL
# 1. Load & prep
df = fetch_bike_sharing()
bike = df['cnt'].asfreq('h').interpolate()   # fill any gaps
slice_ = bike['2011-01-01':'2011-03-31']
# 2. Compute decompositions
add = seasonal_decompose(slice_, model='additive',    period=24)
mult = seasonal_decompose(slice_, model='multiplicative', period=24)
stl = STL(slice_, period=24).fit()
```

```
# 3. Build the "comparison" plot
fig, axes = plt.subplots(3, 1, figsize=(12, 10), sharex=True)
# Trend comparison
axes[0].plot(add.trend,   label='Additive')
axes[0].plot(mult.trend,  label='Multiplicative')
axes[0].plot(stl.trend,   label='STL')
axes[0].set_ylabel('Trend')
axes[0].legend(loc='upper left')
axes[0].set_title('Trend Components')
# Seasonal comparison
axes[1].plot(add.seasonal,  label='Additive')
axes[1].plot(mult.seasonal, label='Multiplicative')
axes[1].plot(stl.seasonal,  label='STL')
axes[1].set_ylabel('Seasonal')
axes[1].legend(loc='upper left')
axes[1].set_title('Seasonal Components')
# Residual comparison
axes[2].plot(add.resid,   label='Additive')
axes[2].plot(mult.resid,  label='Multiplicative')
axes[2].plot(stl.resid,   label='STL')
axes[2].set_ylabel('Residual')
axes[2].legend(loc='upper left')
axes[2].set_title('Residual Components')
plt.tight_layout()
plt.show()
```

This is the figure we should be getting:

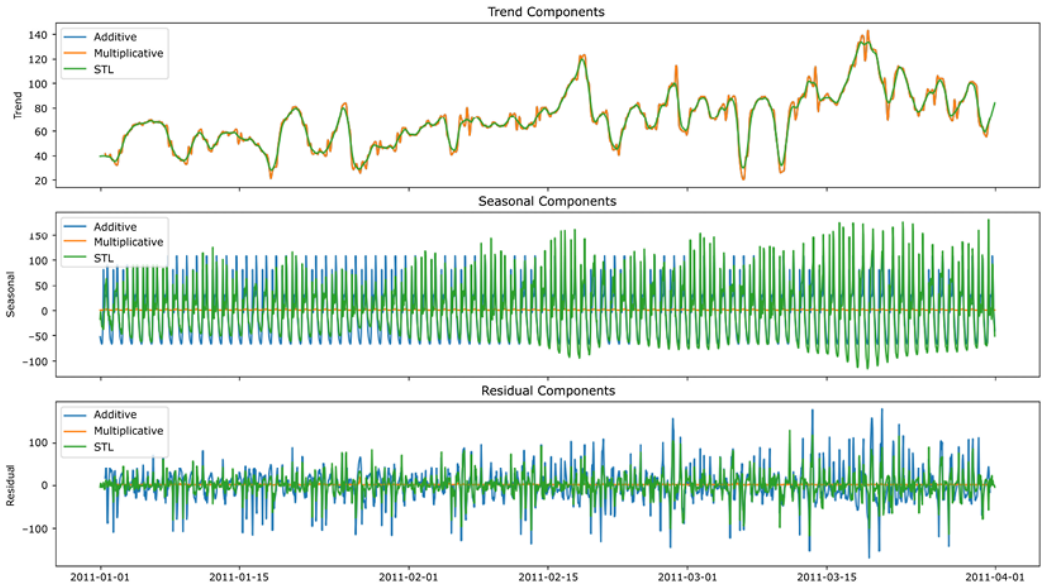


Figure 2.5: Comparison Between Additive, Multiplicative, and STL Decomposition

The plots generated by both `seasonal_decompose` and STL show three panels:

- **Trend:** The extracted long-term trend. In the bike data, you'll see a gradual increase as the weather warms up. The STL trend is often smoother than the one from the classical method.
- **Seasonal:** The repeating 24-hour cycle (morning and evening commute peaks).
- **Residual:** What's left once trend and seasonality are removed, ideally white noise. Any visible structure here means the decomposition missed some systematic pattern.

Each line color represents a different decomposition approach:

- **Additive**: assumes the daily cycle has constant amplitude
- **Multiplicative**: cycle amplitude grows when overall rentals are higher

- **STL**: can adapt its seasonal pattern over time, capturing subtle day-to-day shifts

Look at the seasonal panel of *Figure 2.5* as the trend climbs from January through March (the arm-up out of winter), the multiplicative seasonal component's peak grows with it, while the additive season stays flat. The multiplicative form often fits real-world data better when seasonal effects scale with the overall level of the series. STL sits in between, giving you a seasonality that slowly evolves with the data.

That hour-of-day signal is also a direct cue for feature engineering. When you move on to a tree-based model such as XGBoost or LightGBM, this seasonality is most cleanly captured with cyclical features: $hour_{sin} = np.sin(2 * np.pi * hour/24)$ and $hour_{cos} = np.cos(2 * np.pi * hour/24)$. Encoding hour as a raw integer makes the model treat 23 and 0 as far apart when they are in fact one hour apart. We cover cyclical encoding in detail in *Chapter 5*.

Choosing the right decomposition method

`seasonal_decompose` and STL are just two of many available methods. The best choice depends on the characteristics of your data, such as the complexity and stability of its seasonal patterns. To understand how to make the choice between the methods, look at this table:

Method	**Best For**	**Key Characteristics**
Classical	Simple, stable	Based on moving averages.

Decomposition (`seasonal_decompose`)	series with a single fixed seasonality.	Simple and fast, but assumes a constant seasonal pattern and can be sensitive to outliers.
STL Decomposition	Series with evolving seasonality or outliers.	Uses LOESS smoothing, allowing the seasonal pattern to change over time. Robust to outliers.
MSTL (Multiple STL)	Series with multiple seasonal patterns (e.g., daily and weekly).	An extension of STL that decomposes multiple seasonalities simultaneously, which is crucial for high-frequency data.
Advanced Methods (e.g., Wavelets, EMD)	Highly complex, non-stationary, or non-linear series.	Data-driven methods that can adapt to local changes in frequency and capture time-varying patterns without prior assumptions.

Table 2.2: A guide to choosing a timeseries decomposition method

Decomposition is a powerful diagnostic tool that directly informs your modeling strategy. By isolating the seasonal component in the bike-sharing data, we have confirmed a strong, predictable daily pattern. This discovery is not just an interesting insight; it's an actionable roadmap for building a forecasting model:

1. **Feature Engineering**: We now know that "hour of the day" and "day of the week" are critical features. We must engineer these from the timestamp and include them in our models. We'll explore this more in *Chapter 5*.
2. **Model Selection**: Our model must be capable of handling strong seasonality. A simple linear regression on the raw data would fail. Instead, we should consider:
    - Statistical models designed for seasonality, like SARIMA
    - Models with built-in seasonality handling, like Prophet
    - Tree-based machine learning models (**XGBoost**, **LightGBM**) that can effectively use our engineered time-based features (e.g., `sin_hour`, `cos_hour`, `is_weekend`)

You haven't built a forecast yet, but through decomposition, you already know what one of the most important ingredients must be. That is the power of pattern recognition.

Building a time series data pipeline

Real-world data is messy. Before we can apply the diagnostic tools from the previous section, we must often clean and structure our data first. This section provides the practical data engineering skills needed to build a reliable time series pipeline. We will cover handling messy timestamps, managing different time frequencies, and checking for data quality issues.

This section teaches you to think systematically about temporal data challenges. You'll learn to recognize the warning signs of corrupted temporal structure, choose appropriate strategies for different

missing data patterns, and make informed decisions about frequency conversion. The goal isn't to memorize pandas syntax—it's to develop professional judgment about temporal data quality.

A pipeline also has to decide how the series fit together. Production forecasting problems are rarely a single series in isolation: a retailer wants forecasts at the SKU, store, region, and chain level that all add up; a power utility wants meter-level and grid-level forecasts that stay coherent. That is the territory of *hierarchical forecasting*, where reconciliation methods like Bottom-Up, Top-Down, and MinT (Minimum Trace) adjust independent forecasts so the lower levels sum to the upper ones. Open-source options on the Python side include *hierarchicalforecast* (Nixtla) and *scikit-hts*. We treat the methods in depth in `Chapter 6`; for now, the pipeline-level point is that your data structure (long-format with explicit hierarchy keys) should be set up to support reconciliation later, not retrofitted to it.

Temporal indexing and frequency management

The index is the backbone of any time series. A proper time series index is not just a list of dates; it is a structured object that gives our data temporal context. It enables powerful operations like resampling, windowing, and alignment. Getting the index right is the first and most critical step in data preparation.

But temporal indexing goes beyond format conversion. Unlike regular data where you can shuffle rows without losing information, time series data creates meaning through sequence. The value at 2:30 PM depends on what happened at 2:29 PM. Sorting errors destroy

this fundamental relationship and can make predictable patterns appear random.

Many datasets arrive with dates stored as simple strings or separate columns for year, month, and day. Marketing systems might export "2023-12-25 14:30:00" while IoT sensors transmit "25/12/2023 2:30 PM" and financial APIs return "2023-12-25T14:30:00Z". Our first job is to convert these into a proper `DatetimeIndex` in pandas. This unlocks the full suite of time series functionality.

Your analysis pipeline must reconcile these differences and establish a single, authoritative temporal order. Does a data point arrive every hour? Every day? Every 5 minutes? Answering this question and enforcing a regular frequency prevent subtle bugs and make analysis much more straightforward.

Another common pitfall is time zone management. Data can arrive from servers in UTC, be stored in a local time zone, or have no time zone information at all. It's critical to standardize to a single, consistent time zone (often UTC) early in your pipeline using `pandas.Series.dt.tz_localize` and `pandas.Series.dt.tz_convert` to avoid silent errors.

Consider a temperature monitoring system where three different sensors report to the same database. Each uses a different timestamp format because they were installed years apart by different vendors:

```
# Messy real-world data - exactly what you'll encounter
raw_timestamps = [
    "2023-01-01 08:00:00",    # Sensor A: ISO format
    "01/01/23 9:00 AM",       # Sensor B: US format with AM/PM
    "2023-01-01T10:00:00",    # Sensor C: ISO format, no seconds
```

```
    "1/1/2023 11:00:00"       # Sensor D: US format, different style
]
```

These inconsistencies—multiple timestamp formats, unsorted records, and mixed time zones—are more than minor annoyances; they're structural risks. Before any resampling or modeling, we need a systematic way to standardize, order, and validate our timestamps. That's where a robust conversion and validation strategy comes in.

The strategic solution: robust conversion and validation

The solution isn't just technical—it's systematic. First, standardize all timestamps to pandas `DatetimeIndex` format. Second, sort chronologically (critical for time series). Third, validate the result to catch problems early.

```
import pandas as pd
# Convert mixed formats to DatetimeIndex
df['timestamp'] = pd.to_datetime(
    df['timestamp'],
    format='mixed'
)
df = df.set_index('timestamp').sort_index()
# Validate temporal structure
print(f"Date range: {df.index.min()} to {df.index.max()}")
print(f"Frequency: {pd.infer_freq(df.index) or 'Irregular'}")
if df.index.duplicated().any():
    print(f"Warning: {df.index.duplicated().sum()} duplicate timestam
```

Passing `format='mixed'` lets `to_datetime` parse rows with inconsistent timestamp formats in one pass; pandas 2.0+ infers per-row formats

by default, so you can also drop the keyword if every row follows the same template. Setting the timestamp as the index enables pandas' time series functionality, sorting ensures chronological order, and the validation step catches frequency issues and duplicates that would otherwise surface as silent forecasting bugs.



Professional Insight: Always validate your temporal structure immediately after creation. Frequency inference helps distinguish regular sensor data (hourly, daily) from irregular event data. Duplicate timestamps are common when merging data from multiple sources and must be resolved explicitly.

Quality checks and missing data

A time series with a perfect index can still hide problems. Gaps in the data or sudden, fundamental shifts in its behavior can silently sabotage a forecasting model. In this section, we will learn how to diagnose and treat these issues.

Missing data assessment

Missing data is a common problem. Sensors fail, data streams drop, or manual entry is forgotten. Simply ignoring these gaps, or filling them with a naive value like zero, can distort the statistical properties of your series. We must handle them intelligently.

Time series missing data follows predictable patterns. **Random gaps** suggests sensor failures or network issues—usually acceptable to

interpolate. **Systematic gaps** might indicate maintenance windows or business closures—interpolation could create misleading patterns. **Clustered gaps** often signals major events like power outages or system migrations.

A good first step is to count the missing values. Then, we can choose an appropriate strategy. For time series, **interpolation** is often a powerful technique. It estimates the missing value based on the values before and after it, preserving the temporal flow of the data much better than a simple forward-fill or mean-filling.

Detecting structural breaks

More subtle than missing data is a **structural break**. This is a point in time where the underlying statistical properties of the series change permanently. A new government policy, a major market crash, or a pandemic can all cause structural breaks. A model trained on data from before and after a break, without accounting for it, is learning from two different "regimes" at once. Its performance will suffer.

Picture two distinct point clusters on a scatterplot and a single straight line trying to pass through both. The line ends up too high for one cluster and too low for the other, so it fits neither. A forecasting model learning from data on either side of a structural break is in the same position: it averages the pre-break and post-break dynamics into a single set of coefficients, and the resulting forecasts run hottest right around the boundary, where the regime actually flipped. The remedy is to detect the break, then either restrict training to the post-break regime or add a regime indicator the model can condition on.

Visual inspection using rolling statistics is an effective way to spot potential breaks. A sudden jump in the rolling mean or a sustained change in the rolling standard deviation are clear warning signs.

The choice of gap-filling method should match your data's business context:

- For continuous physical measurements (temperature, pressure, vibration): Use interpolation. Physical systems change gradually, so linear interpolation between known points provides realistic estimates.
- For discrete state variables (machine status, account balances): Use forward-fill. These values persist until explicitly changed—carrying the last known state forward is usually accurate.
- For count data with business meaning (sales, web traffic): Consider the cause. If the system was down, use interpolation. If the business was closed, use zeros.

Let's look at these methods here:

```
# Strategy 1: Interpolation for continuous data
temp_filled = temperature_series.interpolate(method='linear')
# Strategy 2: Forward-fill for state data
status_filled = machine_status.ffill()
# Strategy 3: Business-aware filling
# Fill small gaps with interpolation, large gaps with business logic
sales_filled = sales_data.interpolate(limit=3)  # Max 3-hour gaps
sales_filled = sales_filled.fillna(0)  # Assume closed business = 0 s
```

The `limit` parameter in interpolation prevents filling unreasonably large gaps. A 3-hour gap in hourly temperature data can reasonably

be interpolated. A 3-day gap suggests systematic missing data that interpolation shouldn't bridge.

Quality threshold decision-making

Not all time series with missing data are worth analyzing. To deal with that establish clear thresholds. For example, if more than 20% of data is missing, question the dataset's reliability. If gaps are longer than your forecast horizon, consider whether historical patterns are still relevant. The following quick diagnostic helps you evaluate both the extent and structure of missing values in your series:

```
# Quick quality assessment
missing_pct = series.isnull().sum() / len(series) * 100
max_gap = series.isnull().astype(int).groupby((~series.isnull()).cums
print(f"Missing data: {missing_pct:.1f}%")
print(f"Largest gap: {max_gap} periods")
if missing_pct > 20:
    print("WARNING: High missing data percentage - consider data qual
if max_gap > 24:  # For hourly data
    print("WARNING: Large gaps may affect model reliability")
```

Frequency conversion and resampling decisions

Frequency conversion isn't just a technical operation—it's an analytical decision that affects your results. Converting minute-by-minute sensor data to hourly averages smooths out noise but loses information about short-term spikes. Aggregating daily sales to

weekly totals reveals business cycles but obscures day-of-week effects.

The key insight: **aggregation is irreversible**. Once you've averaged minute data to hours, you cannot recover the minute-level variations. Choose your target frequency based on your analytical goals, not computational convenience. When making frequency conversion decisions, consider the following:

- **Downsampling:** When reducing frequency, the aggregation method should match the data's meaning. Here's how you decide which method to pick:
    - Use `mean()` for intensive variables: Temperature, pressure, prices—values that represent a state
    - Use `sum()` for extensive variables: Sales counts, energy consumption, web traffic—values that accumulate
    - Use `last()` for balances: Account balances, inventory levels—values that represent end-of-period states

  Let's take a quick look at some code here to illustrate these methods:

```
# Intensive variables: mean preserves the underlying level
hourly_temp = minute_temp.resample('h').mean()
# Extensive variables: sum preserves the total quantity
hourly_sales = minute_sales.resample('h').sum()
# Balance variables: last value represents the period-end state
daily_inventory = hourly_inventory.resample('D').last()
```

- Upsampling: When increasing frequency, you're estimating values between known points. Your method should reflect how

the variable actually behaves:

- Forward-fill for step functions: Status indicators, policy rates—values that stay constant until changed
- Interpolation for smooth processes: Continuous measurements that change gradually
- Business logic for scheduled events: Known patterns like store hours or market sessions

The resampling trap: Look-Ahead bias

The most dangerous resampling error is using future information to fill past values. This creates artificially good model performance that collapses in production. To illustrate this, the example below contrasts two common resampling strategies. The first (`bfill`) mistakenly uses future data to fill past gaps, introducing lookahead bias. The second (`ffill`) correctly relies only on past information, preserving the causal structure of the time series.

```
# WRONG: This uses future data to fill past values
bad_fill = daily_data.resample('h').bfill()
# RIGHT: This only uses past information
good_fill = daily_data.resample('h').ffill()
```

In a professional workflow, always resample before creating train/test splits. Never use information from your test period to fill gaps in your training period. Document your resampling decisions—they become part of your model's assumptions.

Practical decision framework

Before resampling, ask three questions:

- What's my analysis timeframe? Hourly models need hourly data. Daily forecasts can use daily aggregates.
- What does this variable represent? Physical measurements behave differently from business counts.
- Am I introducing bias? Ensure your method doesn't leak future information into past periods.

These decisions seem technical, but they fundamentally shape your analysis results. A temperature model trained on hourly averages will miss short-term fluctuations. A sales model using interpolated weekend data might miss the fact that stores are closed.

The goal is developing professional judgment about temporal data quality—understanding not just how to transform data, but when and why different transformations are appropriate for your analytical objectives.

While performing the steps individually is effective for exploring a dataset, a more robust, professional, and reproducible approach is to encapsulate this logic into a standardized pipeline. Doing so ensures that the same quality checks and transformations are applied consistently across all your projects. We will return to this critical concept in *Chapter 10*, where we will build a complete, production-ready pipeline template that combines all of these techniques.

Pattern discovery through visualization

A simple line plot is the first step in analyzing a time series, but it is rarely the last. To truly understand the patterns that drive a series,

we need more specialized tools. This section teaches you how to use visualization not for aesthetics, but as a diagnostic instrument. We will create plots specifically designed to answer questions about seasonality, cycles, and autocorrelation. These visual diagnostics are often the fastest way to gain intuition about your data and formulate a hypothesis for your modeling approach.

Visual diagnostics for temporal structures

How does demand on a Monday differ from a Saturday? How does the pattern in January compare to July? These are questions about grouped or nested seasonality. A standard line plot makes these comparisons difficult. We need to slice and rearrange the data visually to make these patterns pop out. We will focus on three plots: the seasonal plot, the daily profile plot, and the lag plot.

- Seasonal Plots: A seasonal plot displays the data from each season overlaid on top of one another. For our bike-sharing data, a "season" could be a day. By plotting each day's hourly data on the same chart, we can instantly see the typical daily pattern and identify any days that deviate from the norm.
- Daily Profile Plots: A daily profile plot averages the hourly pattern within each day-of-week and overlays the seven curves on a single hour-of-day axis. It surfaces how the weekday commute pattern differs from the weekend leisure pattern at a glance. (Hyndman & Athanasopoulos's FPP3 reserves the term "seasonal subseries plot" for the layout that puts each season in

its own panel; the variant we use here trades that one-panel-per-season clarity for direct visual comparison across days.)

- Lag Plots: A lag plot is a simple scatter plot of a time series against its own lagged value (e.g., value(t) vs value(t-1)). The structure of this plot reveals the nature of the autocorrelation. If the points cluster tightly along a diagonal line, it indicates strong positive autocorrelation. A random cloud of points suggests no autocorrelation.

Let's build these plots using our bike-sharing data. We will use `matplotlib` and `pandas` to create these targeted visualizations.

```
import pandas as pd
import matplotlib.pyplot as plt
from pandas.plotting import lag_plot
from data import fetch_bike_sharing  # was defined earlier in the cha
# 1) Load, fill, slice
bike = (
    fetch_bike_sharing()['cnt']
    .asfreq('h')
    .interpolate()
    .loc['2011-01-01':'2011-06-30']
)
# 2) Prepare the two "wide" DataFrames
# --- Seasonal pivot: hours × days-of-year
seasonal = (
    bike
    .groupby([bike.index.hour, bike.index.dayofyear])
    .mean()
    .unstack(level=1)
)
# --- Daily profile: hours × day-of-week (reindexed Mon–Sun)
order = ['Monday','Tuesday','Wednesday','Thursday','Friday','Saturday
subseries = (
    bike
    .groupby([bike.index.hour, bike.index.day_name()])
    .mean()
```

```
    .unstack(level=1)                  # now: index=hour, columns=day_
    .reindex(columns=order)            # reorder the day_name columns
)
# 3) One Figure, Three Axes
fig, axes = plt.subplots(3, 1, figsize=(14, 20), constrained_layout=T
# 3.1 Seasonal Plot
seasonal.plot(
    ax=axes[0],
    legend=False,
    alpha=0.2
)
axes[0].set_title('Seasonal Plot: Hourly Rentals for Each Day')
axes[0].set_xlabel('')
axes[0].set_ylabel('Rentals')
# 3.2 Daily profile Plot (avg by day-of-week)
subseries.plot(
    ax=axes[1],
    legend=True,
    linewidth=1
)
axes[1].set_title('Daily profile Plot: Avg Hourly Rentals by Day of W
axes[1].set_xlabel('')
axes[1].set_ylabel('Avg Rentals')
# Move legend inside the plot
axes[1].legend(
    title='Day',
    loc='upper right',      # position inside the panel
    frameon=False,          # no box around it
    ncol=1,                 # one column of labels
    fontsize='small',       # shrink text so it doesn't overlap
    bbox_to_anchor=(0.18, 0.98)  # fine-tune inside placement
)
# 3.3 Lag Plot
lag_plot(bike, lag=1, ax=axes[2])
axes[2].set_title('Lag Plot (t vs. t-1)')
axes[2].set_xlabel('Rentals at Hour t-1')
axes[2].set_ylabel('Rentals at Hour t')
plt.show()
```

Here's how our plots look:

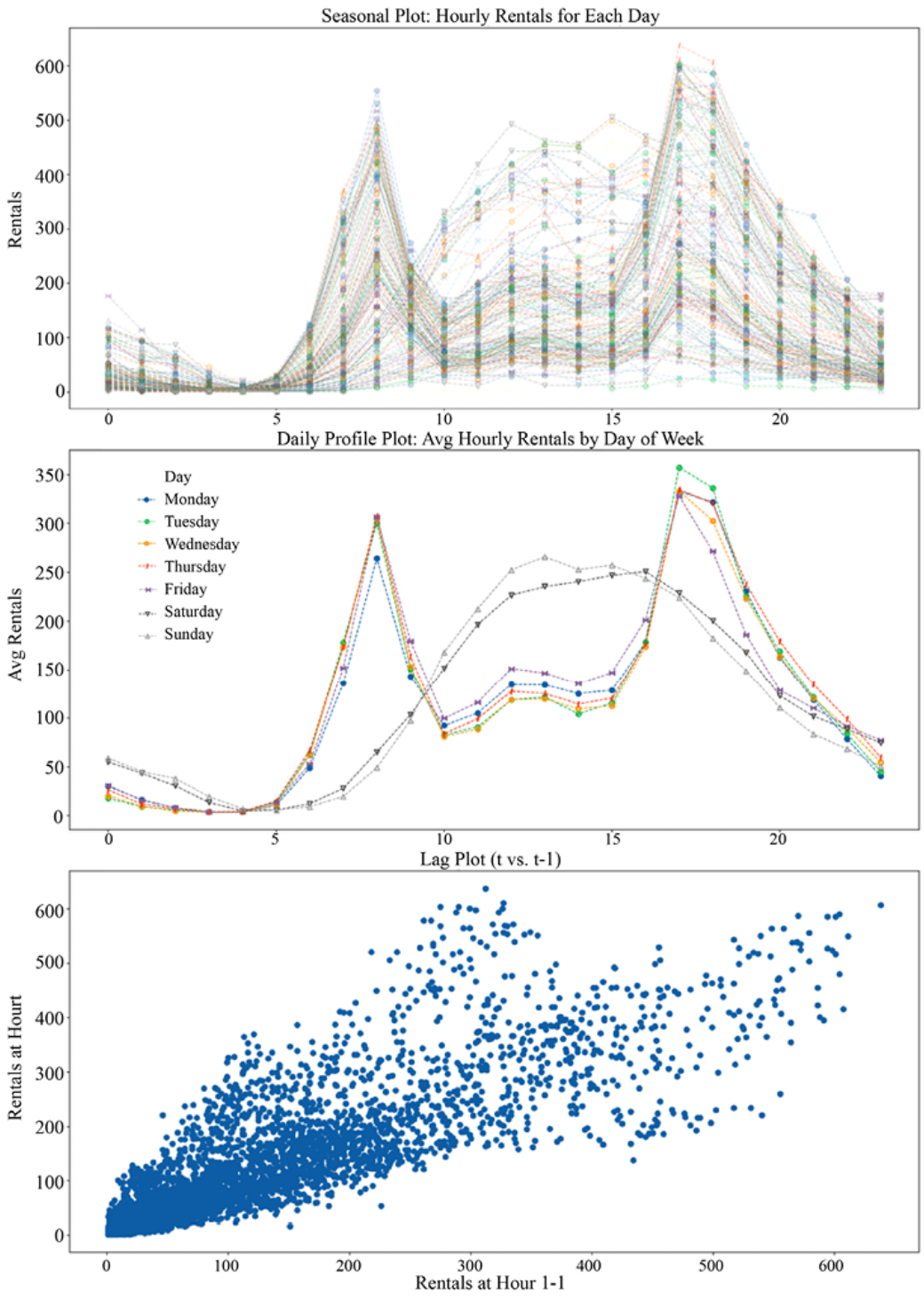


Figure 2.6: Diagnostic Plots

These plots provide immediate modeling clues:

- **Seasonal Plot:** The dense overlay of lines confirms a strong, repeatable daily pattern. The two main humps show the morning and evening commute peaks.
- **Daily Profile Plot:** This plot clearly separates the weekday and weekend patterns. The two-hump commute pattern is strong Monday-Friday, but it morphs into a broader, single-hump "leisure" pattern on Saturday and Sunday. This tells us that "day of the week" is a crucial feature. A model that treats all days the same will perform poorly.
- **Lag Plot:** The points form a distinct, tight cluster along the diagonal. This indicates a very strong positive autocorrelation at lag 1. The number of rentals in one hour is an excellent predictor of the number of rentals in the next hour.

By using these targeted visualizations, we have moved beyond a simple "the data goes up and down" to a specific, actionable insight: our model must account for hour-of-day and day-of-week effects, and it should leverage the strong short-term autocorrelation.

Autocorrelation and residual analysis

In the section *Temporal dependencies and autocorrelation*, we introduced autocorrelation as a measure of how a series correlates with its past values. The lag plot in the previous section gave us a visual hint of this relationship. Now, we will use two standard and more powerful tools to quantify this structure across many lags: the **Autocorrelation Function** (**ACF**) plot and the **Partial Autocorrelation Function** (**PACF**) plot.

- **ACF Plot:** Shows the correlation of the series with its lags. The correlation at lag 2 is also influenced by the correlation at lag +1, so the ACF does not isolate these effects.
- **PACF Plot:** Shows the *direct* correlation between the series and a specific lag, removing the indirect correlations from shorter lags. For example, the PACF at lag 2 shows only the correlation that lag 2 adds, independent of lag 1's effect.

A useful way to keep these straight: the ACF at lag k is the *total* correlation, including everything that bleeds through from shorter lags; the PACF at lag k is the *new* correlation that survives after the shorter lags are accounted for. If lag 1 alone explains why an observation looks like its lag-2 neighbor, the ACF at lag 2 will still be sizable but the PACF at lag 2 will collapse toward zero.

These plots are fundamental for identifying the type of autocorrelation present in a series, which is a key input for traditional forecasting models like ARIMA.

The ultimate goal of a forecasting model is to capture all the predictable patterns in the data, leaving behind only unpredictable, random noise. This means the model's **residuals** (the errors between the predicted values and the actual values) should be free of any remaining autocorrelation.

If we find significant autocorrelation in our model's residuals, it is a clear sign that the model has failed. It has left a predictable pattern on the table. Therefore, plotting the ACF of the residuals is one of the most important diagnostic checks you can perform.

The most honest way to evaluate your model is to inspect what it leaves behind. The residuals are what your model could not explain. If there is still a clear, predictable pattern in the residuals (as seen in the ACF plot), your model has failed to capture all the available information. A good model leaves behind only random, unpredictable noise; its residuals should look like white noise on an ACF plot.

Let's use the `statsmodels` library to create these plots for our bike-sharing data. We will:

1. Plot the ACF and PACF of the original series to see its rich structure
2. Apply differencing to the series and re-plot the ACF to show how this operation removes the trend-driven autocorrelation
3. Fit a simple dummy model (a daily mean) and plot the ACF of its residuals to demonstrate a failed model

```
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
# --- 1. Fetch and Prepare Data ---
df = fetch_bike_sharing()
bike_series = df['cnt'].asfreq('h')['2011-01-01':'2011-03-31'] # Use
bike_series.interpolate(inplace=True) # Fill any minor gaps
# --- 2. ACF and PACF of the Original Series ---
fig, axes = plt.subplots(2, 1, figsize=(12, 8))
plot_acf(bike_series, lags=48, ax=axes[0],
         title='ACF of Hourly Bike Rentals')
plot_pacf(bike_series, lags=48, ax=axes[1],
          title='PACF of Hourly Bike Rentals')
plt.tight_layout()
plt.show()
# --- 3. Effect of Differencing on Autocorrelation ---
# Apply a seasonal difference (24 hours) to remove the daily pattern
differenced_series = bike_series.diff(24).dropna()
```

```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))
plot_acf(bike_series, lags=48, ax=axes[0],
         title='ACF Before Differencing')
plot_acf(differenced_series, lags=48, ax=axes[1],
         title='ACF After 24-Hour Differencing')
plt.show()
# --- 4. Residual Autocorrelation from a Dummy Model ---
# Create a naive forecast: predict each day's value using that day's
daily_mean = bike_series.groupby(bike_series.index.date).transform('m
residuals = bike_series - daily_mean
# Plot the residuals and their ACF
fig, axes = plt.subplots(2, 1, figsize=(12, 8))
residuals.plot(ax=axes[0], title='Residuals from Daily Mean Model')
axes[0].set_ylabel('Error')
plot_acf(residuals, lags=48, ax=axes[1], title='ACF of Residuals')
plt.tight_layout()
plt.show()
```

You should see this plot:

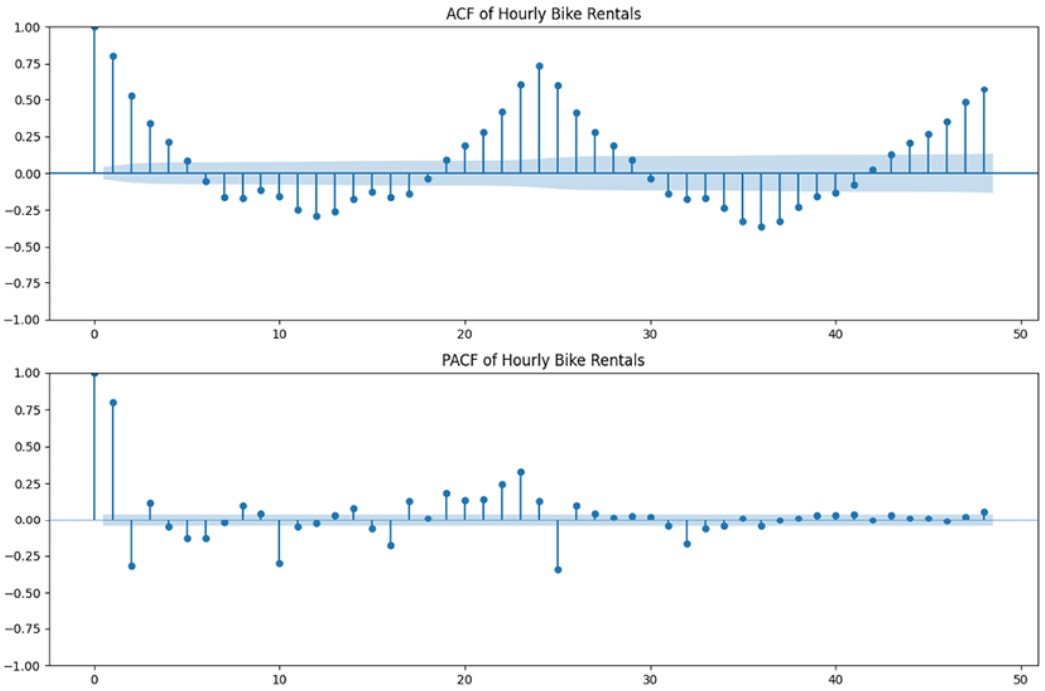


Figure 2.7: ACF and PACF of Hourly Bike Rentals

What does this show?

- **Original ACF/PACF:** The ACF plot for the original series shows that the correlation between an observation and its past values remains high for many lags. This slow decay is typical of a series with a strong trend and seasonality. The PACF plot has significant spikes at lags 1, 2, and around 24, indicating strong direct relationships at those lags.
- **Differenced ACF:** After applying a 24-hour difference, the new ACF plot looks very different. The long, slow decay is gone. We have successfully removed the primary daily seasonal pattern, and the series is now much closer to stationary.
- **Residual ACF:** This is the most important plot. The ACF of our dummy model's residuals shows massive, significant spikes. The wavy pattern clearly indicates that our simple model completely failed to capture the within-day autocorrelation. There is a predictable structure in the errors, which means our model is inadequate.

This exercise proves that diagnostic plots are not optional. They are the primary tool for validating whether your model has learned what it was supposed to learn. A "good" model is one that leaves no patterns behind in its residuals.

Please note that in some series, a slow decay is an indicator of long memory, which means that the influence of observations from the distant past lingers for a long time. While differencing helps remove the primary seasonal effect, true long-memory processes often require more advanced techniques like fractional differencing to be made stationary.

Static plots are powerful, but they have a fundamental limitation: they provide only one view of the data. What if you want to inspect a surprising spike more closely? Or compare the patterns from the first week of January to the first week of February? With static plots, this requires writing new code to filter the data and generate a new image for each question.

Interactive visualization solves this problem. By embedding plots that allow for zooming, panning, and hovering, we can empower ourselves to explore the data fluidly and intuitively. This "conversation" with the data often leads to discoveries that would be missed in a static analysis. A single interactive chart can replace dozens of static ones.

In the repository for this book on GitHub we're providing an example for a visualization with the plotly library of our bike-sharing data to create an interactive line plot. You can visually identify the overall upward trend as the year progresses, spot the dip in rentals during colder months, and zoom into any specific day to confirm the two-hump commute pattern we identified earlier.

Forecastability assessment

We now have the tools to find patterns and build a clean data pipeline. It is tempting to jump directly to building a sophisticated model. But this skips a vital strategic step: assessing **forecastability**. Just because you *can* fit a model to a time series does not mean the resulting forecast will have any practical value.

Not every time series is worth forecasting. Some contain so much noise that predicting them is futile. Others have patterns so weak that simple heuristics perform as well as complex models. Before investing weeks in sophisticated modeling, you must assess whether your series has enough predictable signal to justify the effort.

This section teaches systematic approaches to measure predictability and establish performance benchmarks. You'll learn to calculate signal-to-noise ratios, implement baseline forecasting models, and make data-driven decisions about project viability. This analytical discipline prevents costly failures and sets realistic expectations for forecasting projects.

Measuring predictability before modeling

The hardest lesson in time series work is knowing when to walk away. Consider these real scenarios:

- Scenario 1: Daily Stock Returns A financial firm spends months building neural networks to predict daily stock price movements. Despite sophisticated architectures and massive datasets, their model barely outperforms random guessing. The problem? Daily stock returns are dominated by noise, making short-term prediction essentially impossible.
- Scenario 2: Weekly Sales with Clear Patterns: A retailer applies the same modeling approach to weekly sales data. Their simple seasonal models achieve 15% MAPE, enabling better inventory decisions and reducing stockouts by 30%. The difference? Sales data contain strong, recurring patterns that models can learn.



The Strategic Insight: Predictability varies dramatically across time series. Pattern strength, not model sophistication, determines forecasting success.

So how do you know if you're going in the right direction? You can ask yourself the following questions:

- Does this series contain enough signal to predict?
- Can I beat simple baseline methods?
- Are the patterns stable enough to rely on?
- Will the forecast accuracy justify the modeling investment?

Therefore, before writing a single line of modeling code, we can get a strong sense of a series' predictability. We will use two simple but effective techniques: calculating a signal-to-noise proxy and establishing a naive forecast baseline.

Signal-to-noise proxy

Predictability comes from signal (patterns) relative to noise (randomness). High signal-to-noise ratios suggest excellent forecasting potential. Low ratios indicate that sophisticated models will struggle to beat simple heuristics.

A simple proxy for forecastability is to compare the variation in the original series (the signal) to the variation in its period-to-period changes (the noise). A series with strong patterns will have a much larger overall standard deviation than the standard deviation of its differences.

Note that this is a practical heuristic, not a formal definition of a signal-to-noise ratio. It serves as a quick diagnostic to gauge whether a series is dominated by its own internal variation (signal) or by random, period-to-period fluctuations (noise). A low value is a strong warning sign.

The naive model

Simple baseline models establish the minimum performance threshold for any sophisticated approach. The simplest possible forecast is the **naive forecast**. It states that the forecast for tomorrow is simply the value from today: `Forecast(t+1) = Value(t)`.

This basic model provides a powerful reality check. Any sophisticated model you build must significantly outperform this naive baseline. If it cannot, your complex model is not adding value. These baselines also provide interpretable benchmarks for stakeholders.

Let's apply these ideas to our predictable bike-sharing data and the less predictable daily stock returns of Apple (AAPL).

```
import pandas as pd
import numpy as np
from sklearn.metrics import mean_absolute_error
df_bike = fetch_bike_sharing()
bike_series = df_bike['cnt'].asfreq('h').interpolate()
aapl_df = fetch_aapl()
aapl_returns = aapl_df['Close'].pct_change().dropna() * 100
def calculate_snr_proxy(series):
    """A simple proxy for signal-to-noise ratio."""
    signal_std = series.std()
    noise_std = series.diff().std()
    snr = signal_std / noise_std if not np.isclose(noise_std, 0) \
```

```
    return snr.item() if isinstance(snr, pd.Series) else snr
def get_naive_forecast_mae(series):
    """Calculates the Mean Absolute Error of a naive forecast."""
    y_true = series.iloc[1:]
    y_pred = series.iloc[:-1]
    return float(mean_absolute_error(y_true, y_pred))
print("--- Forecastability Analysis ---")
print(f"Bike Rentals SNR Proxy: {calculate_snr_proxy(bike_series):.2f
print(f"Apple Returns SNR Proxy: {calculate_snr_proxy(aapl_returns):.
mae_bike = get_naive_forecast_mae(bike_series)
mae_aapl = get_naive_forecast_mae(aapl_returns)
print(f"Bike Rentals Naive MAE: {mae_bike:.2f} (vs. Mean of {bike_ser
print(f"Apple Returns Naive MAE: {mae_aapl:.2f} (vs. Mean of {aapl_re
```

You should see this output:

```
--- Forecastability Analysis ---
Bike Rentals SNR Proxy: 1.80
Apple Returns SNR Proxy: 0.66
Bike Rentals Naive MAE: 64.31 (vs. Mean of 187.89)
Apple Returns Naive MAE: 2.23 (vs. Mean of 0.12)
```

This shows that the bike data has a much higher SNR proxy, indicating strong patterns relative to noise. Its naive forecast MAE is also relatively small compared to the average number of rentals. The opposite is true for stock returns. This five-minute analysis tells us to proceed with confidence for the bike data but be highly skeptical about forecasting daily stock returns.

Beating a naive forecast is a low bar. For seasonal data, we must use a tougher benchmark. We must also be vigilant against common errors that create false confidence.

The seasonal naive benchmark

For data with strong seasonality, the **seasonal naive model** is a more appropriate benchmark. It predicts the value for a given time point using the value from the *previous season*. For our hourly data with a daily pattern, the forecast for 10 AM on Friday is simply the value from 10 AM last Friday.

```
Forecast(t) = Value(t - season_length)
```

This model inherently captures the main seasonal pattern. Any complex model we build must outperform this simple, intuitive benchmark.

The catch is that hourly bike-sharing data does not have *one* season; it has at least two: a daily commute pattern (period 24) and a weekly weekday/weekend pattern (period 168 = 24 × 7). A baseline that only honors the daily cycle leaves the weekly information on the table. The honest move is to ladder a few candidate periods and report the best, not to pick one and call it the baseline.

```
# Use one year of bike data
bike_series_2012 = df_bike['cnt'].asfreq('h').interpolate()['2012']
def get_seasonal_naive_mae(series, season_length):
    """Calculates MAE for a seasonal naive forecast."""
    y_true = series.iloc[season_length:]
    y_pred = series.iloc[:-season_length]
    return mean_absolute_error(y_true, y_pred)
# Ladder a few plausible periods.
for period, label in [
    (1, 'Naive (lag 1)'), (24, 'Daily (24h)'), (168, 'Weekly (168h)')
]:
    mae = get_seasonal_naive_mae(bike_series_2012, period)
    print(f"{label:20s} MAE: {mae:.2f}")
# For hourly data with daily seasonality, the period is 24
mae_seasonal_naive = get_seasonal_naive_mae(bike_series_2012, 24)
```

```
mae_simple_naive = get_naive_forecast_mae(bike_series_2012)
print(f"Simple Naive MAE: {mae_simple_naive:.2f}")
```

We should see this output:

```
 Naive (lag 1)        MAE: 79.73
Daily (24h)           MAE: 76.87
Weekly (168h)         MAE: 61.75
```

Two things stand out. First, the daily seasonal naive only nudges the simple naive (79.73 → 76.87, a 3.6% improvement); using period 24 alone would barely register as a baseline. Second, switching to period 168 cuts the error to 61.75, a 22.5% improvement, because that period encodes both the daily commute pattern and the weekday/weekend split. The weekly seasonal naive is the baseline the rest of the chapter compares against, and any ML model worth its training cost has to clear that bar, not the easier 24-hour bar.

A framework for project viability

Instead of a rigid script, use this professional checklist to decide whether a forecasting project should proceed.

1. Data Quality Assessment
    - **Sufficient History?** Do you have enough data to capture the patterns you want to model? (e.g., several years for yearly seasonality).
    - **Missing Data:** Is more than 20% of the data missing? Are there large, continuous gaps that might hide a change in the underlying system?

- **Data Integrity:** Are timestamps correct? Are the values plausible?

2. Predictability Assessment
    - **Visual Patterns:** Do visual tools (seasonal plots, ACF) show clear, repeating patterns, or does the series look like random noise?
    - **Signal-to-Noise:** Does a simple SNR proxy suggest the signal is much stronger than the noise?
    - **Stationarity:** Is the series stationary, or can it be made stationary easily through differencing?
3. Baseline Performance
    - **Beating Naive**: Can a simple model (like seasonal naive) produce a reasonably low error? A very high baseline error suggests the series is highly volatile and difficult to predict.
    - **Margin for Improvement**: Is the baseline error low enough that a more complex model has room to improve, but not so low that further effort is pointless? The ideal scenario is a baseline that is decent, which indicates predictable patterns exist, but there's also enough unexplained variance for a more sophisticated model to provide meaningful improvement.
4. Business Alignment
    - **Accuracy Needs:** What level of accuracy is actually required to make a better business decision? Can your baselines already achieve this?
    - **Forecast Horizon:** How far into the future do you need to predict? Predictability decays over longer horizons.

- **Decision Impact:** What is the cost of a wrong forecast? High-stakes decisions require more confidence and rigorous validation.

Based on this framework, our bike rental project is a clear **"GO"**: the data quality is high, patterns are strong, baselines are good but beatable, and the forecast can directly inform operational decisions.

Summary

This chapter provided the strategic foundation for all future modeling. You learned how to diagnose a time series by identifying its core patterns, testing statistical properties such as stationarity, and using decomposition to separate a series into its trend, seasonal, and residual components. You also learned how to prepare temporal data, handle missing values, make informed resampling decisions, and use visualizations such as seasonal plots and the Autocorrelation Function (ACF) to identify patterns in your data. Most importantly, you learned how to assess forecastability using strong baselines to determine whether a series contains a predictable signal worth modeling.

This diagnostic work directly informs our modeling strategy. Now that you know how to identify the core patterns and statistical properties of a time series, you are ready to build models that can learn from them. In the next chapter, we will use this foundation to construct, validate, and interpret our first forecasting models: classical statistical methods. These are not merely historical techniques, but powerful and interpretable models that serve as robust baselines for any advanced approach.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

https://discord.gg/VXpz9YPUTW



3

Classical Models and Validation

In *Chapter 2*, we learned how to prepare, visualize, and diagnose time series data. In this chapter, we turn those insights into working forecasts and a validation framework that ensures those forecasts hold up in practice. Before moving to machine learning and deep learning, we establish strong statistical baselines that every advanced model must beat.

While the allure of complex machine learning is strong, the starting point for building any professional forecasting system is a solid grasp of classical statistical models. This chapter builds robust, interpretable forecasts using three proven methods: Exponential Smoothing, ARIMA, and the Theta method. We'll compare their strengths and limitations to establish performance benchmarks that any advanced approach must exceed before earning a place in the toolkit.

Building models is only half the battle. Professional forecasting demands proof that models survive real-world conditions. We'll expose the dangers of naive evaluation, implement proper time-aware validation, and establish meaningful performance metrics. The chapter delivers both trusted baseline models and a validation workflow that supports all future forecasting efforts.

In this chapter we will look at:

- Why statistical methods matter
- The role of baselines in forecasting
- Classical baselines
- Validation and metrics

In the age of deep learning, why dedicate time to methods developed decades ago? Let's see why statistical models matter!

Why statistical methods matter

Why dedicate time to methods developed decades ago when cutting-edge neural networks dominate headlines? The answer reveals a fundamental truth about professional forecasting: ***complexity is not inherently valuable—accuracy and reliability are***.



The assumption that sophisticated methods automatically deliver superior results is called the complexity trap. Complex models accumulate failure points: data preprocessing, feature engineering, hyperparameter sensitivity, and black-box debugging. Simple models minimize these risks while often matching or exceeding

> complex model performance. Professional forecasting demands proof that complexity delivers measurable gains worth the operational overhead.

Every complex model faces the same test: outperforming a simple baseline. Classical models, built from decades of statistical research, are fast, interpretable, and often surprisingly hard to beat. They make excellent benchmarks: if your neural network truly delivers a 5% improvement over ARIMA, that gain is likely real and justifies the added complexity; if it doesn't, we should stop wasting time and use the simpler approach.

The diagnostic work from *Chapter 2* directly informs the modeling strategy. Detected strong seasonality? ETS models are designed to handle exactly that pattern. Found complex autocorrelation structure? ARIMA models excel at learning from temporal dependencies. Identified a series with robust but simple trends? The Theta method might be our best choice.

Consider these real-world scenarios:

- **Scenario A:** A retail company spends six months building a deep learning system for demand forecasting. After extensive hyperparameter tuning and feature engineering, it achieves 12% Mean Absolute Percentage Error (MAPE). The problem is that a simple exponential smoothing model achieves 11% MAPE and runs 100x faster. The complex system is objectively worse.
- **Scenario B:** A financial firm's LSTM network predicts stock returns with 8% MAPE compared to ARIMA's 12% MAPE. The neural network's 4-percentage-point improvement justifies its complexity—but only because they established the classical baseline first.

These scenarios also highlight another advantage of classical models: they reveal the structure of your series. Without the baseline, you can't distinguish valuable complexity from impressive-sounding inefficiency.

Classical models force you to think systematically about what drives your time series. Every ETS model explicitly separates level (the underlying value), trend (the long-term direction), and seasonality (repeating patterns). Every ARIMA model decomposes temporal relationships into autoregressive patterns, differencing requirements, and moving average structures.

This decomposition builds intuition that serves you regardless of which models you ultimately deploy. When your transformer-based forecast fails catastrophically, understanding trend and seasonality helps you diagnose what went wrong. When stakeholders ask why demand spiked in December, you can explain seasonal components rather than pointing to mysterious neural network weights.

Complex models excel at pattern recognition but fail at explanation. Classical models provide the opposite strength—clear, interpretable relationships that stakeholders can understand and trust.

When presenting forecasts to business leaders, you can explain:

- "Sales increase 15% each December due to holiday seasonality"
- "The underlying trend shows 2% monthly growth after removing seasonal effects"
- "The model adapts quickly to recent changes with a smoothing parameter of 0.3"

Try explaining why your LSTM's attention mechanism focused on lag 23 while ignoring lag 24. A gradient boosted model might use 500 trees with thousands of splits, each contributing incrementally to the final prediction. Transformers distribute attention across hundreds of time steps through multiple heads simultaneously. Foundation models add billions of pre-trained parameters encoding patterns from vast datasets. The interpretability gap matters when decisions involve millions of dollars and regulatory scrutiny.

Consequently, benchmarking should mirror the downstream decision (inventory, bidding, portfolio construction) rather than relying on a single, generic error metric. We'll go through evaluation in the second part of this chapter.

In the next three sections, we'll master the essential classical methods: Exponential Smoothing for series with clear trends and seasonality, ARIMA for complex temporal dependencies, and Theta for robust general-purpose forecasting. Each represents decades of statistical research distilled into practical tools that remain competitive with modern alternatives. The goal isn't to stop at classical methods but rather to build the professional foundation that makes everything else more effective.

Even as foundation models and transformer-based approaches gain traction in time series forecasting, strong statistical baselines remain essential. In practice, many modern systems are hybrid—combining classical models with machine learning or using statistical methods as benchmarks and fallback systems. This makes understanding classical forecasting not optional, but foundational.

The role of baselines in forecasting

The M-competitions, large forecasting tournaments spanning thousands of series, repeatedly showed that carefully tuned classical methods remain highly competitive. The M4 (2018) winner used a hybrid exponential-smoothing + RNN approach, and purely deep-learning entries often trailed classical benchmarks. Outcomes, however, depend heavily on the task setup (data, horizon, hierarchy) and the evaluation metric.

In M5 (2020), which focused on highly intermittent retail sales, gradient-boosted trees performed strongly and top submissions were typically ensembles that combined simple statistical models and modern ML; simple models also frequently outperformed complex ones on many individual series.

More recent competitions have broadened the focus: which methods win in practice depends on the application — retail and hierarchical SKU forecasting often reward gradient-boosted trees with heavy feature engineering and ensembling, energy and weather tasks place a premium on multivariate and probabilistic models, and finance prioritizes decision-aware utility over raw point accuracy.

For example, the financial M6 contest (live Feb 2022–Jan 2023) explicitly evaluated investment value and rewarded probabilistic forecasts and decision-aware pipelines rather than only point-forecast improvements, and energy/market challenges such as HEFTcom (2024) have shown that pairing solid forecasts with robust bidding or trading logic drives real value. At the same time,

LTSF/Transformer benchmark suites (Autoformer, Informer, FEDformer and variants) have pushed advances in long-horizon forecasting although simpler linear baselines like DLinear/NLinear remain surprisingly strong on some standard LTSF splits. We'll cover machine-learning models in *Chapters 4-6*, deep-learning models in *Chapter 7*, and uncertainty quantification in *Chapter 8*.

Classical baselines

In this section we look at some of the classical baselines. In professional forecasting, a model earns adoption only if it improves outcomes enough to justify added complexity. That is why we start with strong classical baselines: they are fast, interpretable, and difficult to beat. The goal is not to 'avoid modern methods', but to ensure that any advanced approach improves meaningfully over a robust benchmark.

Exponential smoothing

In my experience building forecasting systems, exponential smoothing often outperforms complex machine learning models. It represents one of the most robust foundations you can establish.

When R.G. Brown developed the core recursive formula in the 1950s, he solved a critical business problem in operations research when companies like IBM needed fast, reliable demand forecasts: how to forecast demand when computational resources were scarce and data was noisy. His solution can be explained using the following formula:

$$l_t = \alpha y_t + (1 - \alpha) l_{t-1}$$

This recursive formula creates a weighted average where weights decay exponentially into the past. The parameter α controls how quickly we adapt to new information. Recent observations get weight α, the previous observation gets $\alpha(1-\alpha)$, the one before gets $\alpha(1-\alpha)^2$ and so on. This simple recursion creates exponentially decaying weights that automatically emphasize recent observations while preserving historical learning.

Before modern computers, this recursion could be calculated by hand or on primitive machines. The forecast for any horizon was simply the latest level estimate $\hat{y}_{t+h} = l_t$.

For example, in retail inventory planning, yesterday's sales tell us more about tomorrow's demand than sales from six months ago. But completely discarding historical data would be foolish because seasonal patterns and long-term trends contain valuable signals. Exponential smoothing provides the mathematical framework to balance these competing needs.

Holt (1957) extended this for trending data by adding a trend component with smoothing parameter T. Winters (1960) added seasonal components with parameter γ. This created the **Error, Trend, and Seasonality (ETS)** taxonomy: 30 possible models combining different ETS specifications.

ETS models explicitly decompose a time series into a structural baseline plus an unpredictable noise term. The taxonomy labels the form of the error (E), trend (T), and seasonality (S), typically as additive or multiplicative, and the model then tracks the following interpretable pieces:

- **Error** (**E**): The unpredictable component (residuals), what remains after the model's level, trend, and seasonality have been removed. Error captures measurement noise and short-term shocks; its assumed form (additive or multiplicative) determines how these shocks influence the observed series and the model's updating rules. Error is what we inspect in diagnostics (for example residual plots and autocorrelation) to check model fit.
- **Level (L)**: The baseline of the time series, the value around which the series fluctuates once trend and seasonality are removed. You can think of it as the center line that the model tracks and updates each period. Unexpected deviations (the error term) push the level up or down, but the level is the latent state the model maintains, not the noise itself.
- **Trend (T)**: The long-term directional movement. Ask yourself, is the series generally increasing, decreasing, or remaining stable over time?
- **Seasonality (S)**: Repeating patterns at fixed intervals. This includes daily cycles, weekly patterns, monthly variations, or annual seasonality.

Each component can be modeled as additive (components add together) or multiplicative (components multiply together). For additive ETS models, use the following formula:

$$\hat{y}_{t+h} = l_t + hb_t + s_{t+h-m}$$

For multiplicative ETS, we take the product of components instead of a sum:

$$\hat{y}_{t+h} = l_t \times b^h{}_t \times s_{t+h-m}$$

Where:

- T = forecast horizon (how many steps ahead we're predicting)
- m = seasonal period (for example 12 for monthly data with yearly seasonality, 4 for quarterly)
- s_{t+h-m} = the appropriate seasonal component for the forecast period

The choice between multiplicative and additive depends on whether seasonal effects grow proportionally with the level of the series.

One of ETS's greatest strengths is component interpretability. We can extract and examine each component to understand what drives the series. The trend shows long-term growth patterns. The seasonal component exposes predictable cycles, perhaps higher searches during academic semesters or conference seasons.

This decomposition provides business-relevant insights that stakeholders can understand and trust. When executives ask why forecasts changed, we can point to specific components.

Let's see this in action with a simple example. We can use the `statsmodels` library to fit an ETS model. The library automates the selection of Additive/Multiplicative components based on the data.

```
import pandas as pd
import numpy as np
from statsmodels.tsa.api import ExponentialSmoothing
import matplotlib.pyplot as plt
# Use the same Google Trends data
dates = pd.date_range('2022-01-01', '2023-12-31', freq='D')
```

```
np.random.seed(42)
trend = np.linspace(50, 80, len(dates))
seasonal = 10 * np.sin(2 * np.pi * np.arange(len(dates)) / 365.25)
noise = np.random.normal(0, 5, len(dates))
ml_searches = pd.Series(trend + seasonal + noise, index=dates)
# Fit an ETS model (letting it auto-detect trend and seasonality)
fit = ExponentialSmoothing(
    ml_searches,
    trend="add",
    seasonal="add",
    seasonal_periods=365
).fit()
# Generate a 60-day forecast
forecast = fit.forecast(60)
# Plotting the results
plt.figure(figsize=(10, 5))
plt.plot(ml_searches[-180:], label='Observed')
plt.plot(forecast, label='ETS Forecast', linestyle='--')
plt.title('ETS Forecast with statsmodels')
plt.legend()
plt.show()
print("ETS model fit and forecast generated successfully.")
```

In practice, I've seen retail organizations run ETS baselines across their entire catalog, then selectively apply more complex models only where they demonstrably add value. AWS Forecast includes ETS among its core algorithms, supporting this segmented modeling approach. While it's plausible that large retailers reserve deep learning for high-value SKUs.

In the next section, we'll explore ARIMA models, which take a different approach by focusing on temporal dependencies and autocorrelation patterns that you learned to identify in *Chapter 2*'s diagnostic work.

ARIMA

In my experience, no model better illustrates the trade-offs between statistical rigor and real-world chaos than ARIMA. A mid-sized investment firm I consulted for had great success using ARIMA for forecasting stable, blue-chip stocks, achieving impressive accuracy. When they applied the same models to the cryptocurrency market, the results were disastrous, with errors exceeding 30%. This story provides the perfect hook for understanding ARIMA: it is a powerful tool, but only when its underlying assumptions hold.

ARIMA models exploiting autocorrelation patterns we discussed in *Chapter 2*. While ETS explicitly separates trend and seasonality, ARIMA takes a different approach: it learns these patterns implicitly through temporal dependencies.

The model was formalized by George Box and Gwilym Jenkins (hence called the Box-Jenkins methodology) in their seminal 1970 book, Time Series Analysis: Forecasting and Control. Before their work, forecasting often relied on simpler smoothing methods. Box and Jenkins introduced a systematic methodology to identify, estimate, and diagnose models that could capture complex temporal memory structures within the data. They solved the problem of moving beyond simple averages to a robust statistical framework.

ARIMA stands for **AutoRegressive Integrated Moving Average**. To truly understand it, let's look at how it's constructed. Unlike ETS, which gives a direct formula for the forecast, ARIMA provides a formula that describes the structure of the time series itself.

Let's say y'_t is our time series after it has been made stationary by differencing it T times. The ARIMA(p,d,q) model states that y'_tcan be described by the following equation:

$$y'_t = c + \underbrace{\phi_1 y'_{t-1} + \cdots + \phi_p y'_{t-p}}_{\text{AR(p) - Memory of Past Values}} + \underbrace{\theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q}}_{\text{MA(q) - Memory of Past Errors}} + \varepsilon_t$$

Let's break this equation down:

- AR(p)or the autoregressive part: The current value y'_t is a weighted sum of its own p previous values. The φ (phi)d coefficients are learned from the data and represent the strength of the influence from past observations. This is the model's "*memory*."
- MA(q) or the moving Average part: The current value $y'_t$is also influenced by the q previous random errors or "`shocks`" (ε). The k (theta) coefficients capture how these past shocks propagate into the future.
- I(d) or the integrated part: This isn't visible in the equation itself, but it's the crucial first step. The equation only works on a stationary series, y'_t. The d parameter tells us how many times we need to difference the original series y'_t to get y'_t.

This equation ***isn't a direct forecast***; it's a ***model*** of the data generating process. Forecasting is done by rearranging this equation to predict $y'_{\{t+1\}}$ based on the known past values and errors.

The notation ARIMA(p, d, q) specifies the order of each component: p autoregressive terms, d differencing operations, and q moving average terms.

When the series shows seasonality, ARIMA's natural extension is **SARIMA**, written ARIMA(p, d, q) (P, D, Q)_m. The second triple adds an autoregressive, differencing, and moving-average term at the seasonal lag m (12 for monthly data with yearly cycles, 7 for daily data with weekly cycles). `auto_arima` searches both triples when `seasonal=True`. We will use a different trick below for our daily yearly cycle, but the SARIMA shorthand is what to reach for whenever you have a clear, integer-spaced seasonal period.

For decades, finding the right (p, d, q) orders was a manual, expert-driven process. A practitioner had to analyze ACF and PACF plots and use trial-and-error, a process that was more art than science. This all changed with the work of Rob Hyndman and his `auto.arima()` function for R, which automated the search for the optimal model.

This was a revolutionary step for practical forecasting. The first and most well-known Python port of this work is `pmdarima`. Its `auto_arima` function brings this powerful automation to our toolkit, allowing us to leverage ARIMA's statistical power without the manual pain.

Let's use `pmdarima` to find the best ARIMA model for our synthetic Google Trends data. The `auto_arima` function will intelligently search through different combinations of p, d, and q, using an

information criterion (like AIC) to find the model that best fits the data without being overly complex.

We'll start with the imports:

```
# ARIMA Forecasting with pmdarima's auto_arima
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pmdarima as pm
```

Let's use the same Google Trends data from the ETS example:

```
# Use the same Google Trends data from the ETS example
dates = pd.date_range('2019-01-01', '2023-12-31', freq='D')
np.random.seed(42)
trend = np.linspace(50, 80, len(dates))
seasonal = 10 * np.sin(2 * np.pi * np.arange(len(dates)) / 365.25)
noise = np.random.normal(0, 5, len(dates))
ml_searches = pd.Series(
    trend + seasonal + noise, index=dates, name="value")
train_data = ml_searches[:'2023-10-31']
test_data = ml_searches['2023-11-01':]
```

We have our data separated into training and test so we can calculate our performance metrics on the test data.

We'll fit the model using `auto_arima`. It automatically determines `d`, and then searches for `p` and `q`.

Before the fit call, a word on handling seasonality. Our series demonstrates a yearly cycle with daily granularity, establishing a seasonal period of `m=365`. Orthodox guidance recommends SARIMA for seasonal data. Applying SARIMA to such granular data requires estimating backward lags of 365 periods, which consumes vast memory and demands decades of history for stable estimation. The textbook method therefore proves computationally prohibitive and prone to overfitting. We rely instead on harmonic regression with ARIMA errors: we instruct `auto_arima` to model the short-range autocorrelation (`seasonal=False`) and supply the yearly cycle as Fourier terms via the X argument. A handful of sine and cosine pairs captures the annual rhythm, while ARIMA handles the remaining residuals.

```
# Fourier basis at the yearly period; used as exogenous regressors
# because seasonal=False is set below
def fourier_terms(index, period, K):
    t = np.arange(len(index))
    cols = {}
    for k in range(1, K+1):
        cols[f'sin{k}'] = np.sin(2*np.pi*k*t/period)
        cols[f'cos{k}'] = np.cos(2*np.pi*k*t/period)
    return pd.DataFrame(cols, index=index)
PERIOD = 365.25
K = 3
train_exog = fourier_terms(train_data.index, PERIOD, K)
test_exog = fourier_terms(test_data.index, PERIOD, K)
```

Let's train the model!

```
arima_model = pm.auto_arima(
    train_data,
    X=train_exog,
    start_p=0, start_q=0, start_P=0,
    max_p=2, max_q=2,
    max_P=1, max_Q=1,
    seasonal=False,  # avoid SARIMA(m=365)
    d=1, D=1,
    stepwise=True,  # Use stepwise search for speed
    approximation=True, # use faster approximate likelihood
    n_jobs=1, # single process; this reduces memory
    trace=False,
    error_action='ignore',
    suppress_warnings=True
)
print("\n--- Model Summary ---")
print(arima_model.summary())
```

A quick tour of the arguments. `start_p`, `max_p`, `start_q`, `max_q` (and the capital-P versions) bound the AR/MA search range; we cap them low here because the Fourier exog already absorbs the yearly cycle. `seasonal=False` disables the SARIMA loop. `d=1, D=1` fix the differencing orders rather than rerunning the unit-root tests at every step. `stepwise=True` and approximation`=True` together swap the exhaustive grid for Hyndman's stepwise heuristic on a faster likelihood, typically 10–50× faster on daily data with negligible accuracy cost.

Careful with the X parameter, in some versions of `pmdarima`, it's called **exogenous**. If you choose the wrong parameter name, it will get silently ignored and you'll get a flat forecast.

The model is trained. Now I am getting this output:

```
--- Model Summary ---
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 1765
Model:               SARIMAX(2, 1, 2)   Log Likelihood               -5408.653
Date:                Tue, 02 Sep 2025   AIC                          10827.306
Time:                        19:52:55   BIC                          10854.683
Sample:                  01-01-2019   HQIC                         10837.422
                       - 10-31-2023
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.7900      0.193     -4.086      0.000      -1.169      -0.411
ar.L2         -0.0698      0.027     -2.564      0.010      -0.123      -0.016
ma.L1         -0.1169      0.193     -0.606      0.544      -0.495       0.261
ma.L2         -0.6460      0.172     -3.758      0.000      -0.983      -0.309
sigma2        26.9380      0.920     29.275      0.000      25.135      28.742
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):                 2.28
Prob(Q):                              0.96   Prob(JB):                         0.32
Heteroskedasticity (H):               1.02   Skew:                             0.09
Prob(H) (two-sided):                  0.78   Kurtosis:                         2.97
===================================================================================
```

Figure 3.1: StatsModels Model Summary Table for our ARIMA model.

Let's generate a forecast and its 95% confidence band in one call!

```
# Generate forecasts and prediction intervals
n_periods = len(test_data)
forecast, conf_int = arima_model.predict(
    X=test_exog, n_periods=n_periods, return_conf_int=True
)
forecast_series = pd.Series(forecast, index=test_data.index)
```

Let's plot this!

```
# Plotting the results
plt.figure(figsize=(12, 6))
plt.plot(train_data[-180:], label='Observed Training Data')
plt.plot(test_data, label='Actual Values', color='orange')
plt.plot(forecast_series, label='ARIMA Forecast',
         color='red', linestyle='--')
plt.fill_between(test_data.index,
                 conf_int[:, 0],
                 conf_int[:, 1],
                 color='red', alpha=0.1, label='95% Confidence Interval')
plt.title('ARIMA Forecast with pmdarima')
plt.legend()
plt.show()
```

Here's the plot I am getting:

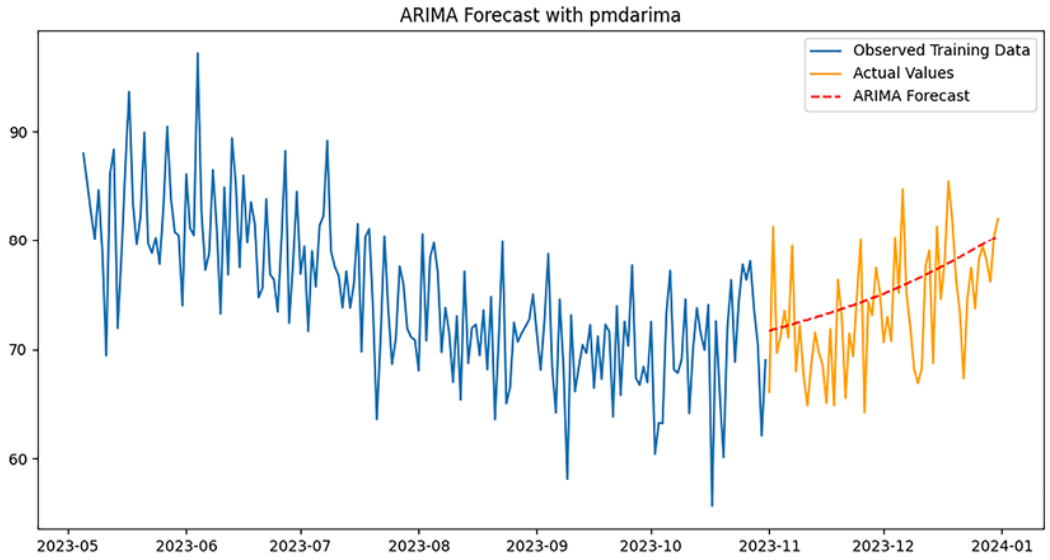


Figure 3.2: ARIMA forecast with pmdarima

We'll calculate the error metrics:

```
# Plotting the results
y_true = test_data
y_pred = forecast_series
rmse = np.sqrt(np.mean((y_true - y_pred) ** 2))
```

```
mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100
naive_errors = np.abs(train_data[1:] - train_data.shift(1)[1:])
scale = naive_errors.mean()
mase = np.mean(np.abs(y_true - y_pred)) / scale
print(f"RMSE: {rmse:.3f}")
print(f"MAPE: {mape:.3f}%")
print(f"MASE: {mase:.3f}")
```

The output is:

```
RMSE: 11.425
MAPE: 14.690%
MASE: 1.861
```

A core principle of the Box-Jenkins methodology is that a good model should capture all the predictable patterns in the data, leaving behind only unpredictable, random noise (known as "white noise"). We must check this by performing a:

- Stationarity Check: Before modeling, we use statistical tests like the **Augmented Dickey-Fuller** (**ADF**) test to determine the order of differencing (d) needed. `auto_arima` performs these tests internally.
- Residual Analysis: After fitting the model, we analyze the residuals (the one-step-ahead forecast errors). They should have no remaining autocorrelation. We can check this visually with an ACF plot of the residuals and formally with a statistical test like the Ljung-Box test. If significant patterns remain, our model is inadequate.

ARIMA is built on a solid statistical foundation. It delivers not only point forecasts but also confidence intervals and a set of diagnostic tools for checking model validity. Those features make ARIMA highly auditable and easy to interpret, which matters a great deal in regulated domains such as finance and in mission critical demand planning.

At the same time there are important limitations to acknowledge. ARIMA assumes linear relationships, so it can struggle with complex nonlinear patterns. Although extensions like ARIMAX allow inclusion of external regressors, incorporating a large number of features is not as straightforward as it is with modern machine learning approaches. ARIMA also has more difficulty handling multiple seasonal patterns at once, for example weekly and yearly cycles.

Despite these drawbacks ARIMA remains valuable today. Its parameters have clear statistical meaning, its prediction intervals are principled, and it can represent a wide range of temporal dynamics when those dynamics are roughly linear. Tools such as `auto_arima` make model selection fast enough to run on thousands of series so ARIMA is often a viable production model and a dependable baseline. Before investing in complex neural networks, data scientists should always ask whether the new model consistently outperforms a well-tuned ARIMA model; if it does not, the extra complexity is usually hard to justify.

Next, we'll explore the Theta method, a competition-winning dark horse which combines the best aspects of both trend decomposition and exponential smoothing while preventing over-

extrapolation of trends.

Theta Method

In 2000, the prestigious M3 forecasting competition pitted the world's best forecasting methods against 3,003 diverse time series. The winner surprised everyone: not a complex neural network or sophisticated econometric model, but an elegant method called Theta that most practitioners had never heard of. This result sent shockwaves through the forecasting community and established a fundamental principle: robustness often beats sophistication.

The method was introduced by Vassilios Assimakopoulos and Konstadinos Nikolopoulos to solve a very practical problem: how can we produce accurate forecasts for a large, diverse set of time series without needing to manually tune a model for each one? They designed a method that was simple, automatic, and surprisingly effective.

Theta's genius lies in how it handles trends, a common point of failure for many forecasting models. Instead of extrapolating a trend indefinitely, which often leads to wildly unrealistic forecasts, Theta inherently dampens it. The method works through a simple decomposition process:

1. **De-seasonalize**: First, if seasonality is present, it is removed, leaving the trend and level components.
2. **Create theta-lines**: The de-seasonalized series is decomposed into two lines. The first line represents the simple, linear trend of the data. The second line is a "dampened" version, created by scaling down the curvature of the original series.
3. **Extrapolate and recombine**: Both lines are extrapolated into the future, and their forecasts are combined. The final forecast is then re-seasonalized.

This process of averaging a standard trend with a dampened one is the secret to Theta's robustness. It produces forecasts that are sensitive to the trend but conservative enough to avoid catastrophic errors. It has been shown that the original Theta method is equivalent to Simple Exponential Smoothing with a drift component, which explains its strong theoretical foundation and excellent performance. The original method is described in Assimakopoulos and Nikolopoulos, "*The theta model: a decomposition approach to forecasting*" (International Journal of Forecasting, 2000); the implementation we use ships in statsforecast.models.Theta (https://nixtlaverse.nixtla.io/statsforecast/).

Traditional methods often fail because they're optimized for specific patterns. ETS excels with clear seasonality but struggles with irregular trends. ARIMA handles complex autocorrelations but can overfit to noise. Theta takes a different approach—it's designed to perform reasonably well across all scenarios rather than perfectly in specific cases.

This philosophy proves especially valuable in practice where:

- You don't know the data generation process in advance
- Series characteristics may change over time
- Robustness matters more than peak performance on any single series

- You need reliable forecasts across diverse business units or product lines

The M3 competition validated this approach across thousands of series from multiple domains, establishing Theta as a proven general-purpose baseline.

While the concept is simple, a fast and reliable implementation is key for practical use. For this, we turn to the `statsforecast` library from Nixtla. As Rob Hyndman notes, Nixtla provides some of the best and fastest Python implementations of his methods. `statsforecast` is optimized with Numba, making it incredibly efficient for forecasting thousands of series at once—a perfect fit for Theta's philosophy.

Let's apply the Theta method to our Google Trends data. The `statsforecast` library requires the data in a specific long format (`unique_id`, `ds`, `y`), which is ideal for handling many time series simultaneously.

```
# Theta Method Forecasting with statsforecast
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsforecast import StatsForecast
from statsforecast.models import Theta
# Use the same Google Trends data
dates = pd.date_range('2019-01-01', '2023-12-31', freq='D')
np.random.seed(42)
trend = np.linspace(50, 80, len(dates))
seasonal = 10 * np.sin(2 * np.pi * np.arange(len(dates)) / 365.25)
noise = np.random.normal(0, 5, len(dates))
ml_searches = pd.Series(trend + seasonal + noise, index=dates)
# Prepare the DataFrame in the required format for statsforecast
df = pd.DataFrame({
    'unique_id': 'ml_searches', # An identifier for the time series
    'ds': ml_searches.index,    # The date stamps
    'y': ml_searches.values     # The target values
})
train_df = df[df['ds'] <= '2023-10-31']
test_df = df[df['ds'] > '2023-10-31']
# 1. Initialize the Theta model, specifying the seasonal period
models = [Theta(season_length=365)]
# 2. Create a StatsForecast object
sf = StatsForecast(models=models, freq='D')
# 3. Fit the model to the training data
sf.fit(train_df)
# 4. Generate forecasts for the horizon of the test set
forecast_df = sf.predict(h=len(test_df))
# Plotting the results
plt.figure(figsize=(12, 6))
plt.plot(train_df['ds'].tail(180), train_df['y'].tail(180),
         label='Observed Training Data')
plt.plot(test_df['ds'], test_df['y'],
         label='Actual Values', color='orange')
plt.plot(forecast_df['ds'], forecast_df['Theta'],
         label='Theta Forecast', color='red', linestyle='--')
plt.title('Theta Forecast with statsforecast')
plt.legend()
plt.show()
```

The Theta method's legacy is its status as the ultimate robust baseline. Its consistent, strong performance has led to its adoption in some of the most demanding forecasting environments in the world. Large organizations like Uber, Bosch, and Amazon Web Services use Theta as a benchmark for their own forecasting systems. When you need to generate reliable forecasts for thousands or even millions of items—like retail SKUs or server metrics—Theta is often the first choice.

Theta stands out for its simplicity and speed, which makes it especially well suited to forecasting tens of thousands of series. It delivers strong performance without any parameter tuning, so it works reliably across a wide range of series types. Because the method is so simple, it requires little ongoing maintenance and presents fewer ways for production pipelines to fail.

There are important limits to keep in mind. Theta is not built to incorporate external variables, so it cannot take advantage of rich covariate information. It also assumes a relatively stable seasonal pattern and may struggle when seasonality evolves over time. Finally, Theta is a forecasting tool rather than an explanatory model, so it does not provide causal insight into the drivers behind the series.

In modern forecasting stacks Theta is prized as the robust baseline. Its consistent accuracy and low computational cost have earned it a place in demanding production environments. Large organizations such as Uber, Bosch, and Amazon Web Services use Theta as a benchmark for their systems. When teams need reliable forecasts for thousands or millions of items, for example retail SKUs or server metrics, Theta is often the first method they run because it is cheap to compute, easy to implement, and difficult to beat.

In the next section, we'll integrate these three approaches, ETS, ARIMA, and Theta.

Overview of statistical forecasting methods

The ETS, ARIMA, and Theta methods represent the core of classical forecasting, but they are part of a much richer ecosystem. The Python statistical forecasting landscape now includes dozens of validated methods, each optimized for specific data patterns and business contexts. Understanding this broader toolkit transforms you from a practitioner who knows a few forecasting methods into one who can strategically select the right tool for any situation.

The landscape of time-series analysis has evolved significantly over the past decade. While R, particularly through Rob Hyndman's influential work, pioneered foundational methods like automatic ARIMA selection and the ETS state-space framework, Python has not only achieved parity but has, in many cases, surpassed R for building production-scale forecasting systems.

The following table provides a comprehensive comparison of Python's statistical forecasting libraries, incorporating real-world performance characteristics:

Library	Primary Use Case	Key Models and Features	Performance	Best For

statsmodels	Statistical inference, detailed model analysis	ARIMA/SARIMAX, VAR/VECM, ETS, State-Space, Comprehensive statistical tests	Baseline speed, memory intensive	Researchers needing p-values, diagnostic plots, and publication-quality statistical output
pmdarima	Automated ARIMA model selection	auto_arima function, scikit-learn-compatible API, Pipeline support	2-5x faster than manual grid search	Quickly establishing ARIMA baselines with minimal configuration
statsforecast	High-performance forecasting at scale	AutoARIMA, AutoETS, Theta with Numba JIT compilation, Parallel processing	10-100x faster than statsmodels	Production systems with thousands of series, real-time forecasting
sktime	Unified framework for ML workflow	Consistent API across 30+ models, Advanced CV tools, Ensemble methods	Varies by backend algorithm	Method comparison, research, complex pipelines
darts	User-friendly deep learning integration	Classical + DL models, Backtesting utilities, Probabilistic forecasts	Good for classical, slower for DL	Rapid prototyping, teams transitioning to DL
orbit	Bayesian forecasting	Bayesian DLT, ETS, LGT models, Stan backend	Slower but provides full posteriors	Uncertainty quantification, causal analysis
prophet	Business forecasting	Additive decomposition, Holiday effects, Intuitive parameters	Medium speed, handles missing data	Business analysts, strong seasonality with holidays
arch	Financial volatility	GARCH family models, Heavy-tailed distributions	Fast for specialized use	Risk modeling, option pricing, VaR calculations

Table 3.1: Comparing Python statistical forecasting libraries

While ETS, ARIMA, and Theta cover many use cases, several specialized methods excel in specific scenarios:

- **Prophet** was designed for the realities of business forecasting: strong seasonal patterns, holiday effects, and missing data. Its decomposition into trend, seasonality, and holidays is intuitive for stakeholders.
- **BATS and TBATS** handle the complexity of multiple seasonal patterns—think hourly electricity demand with daily, weekly, and yearly cycles. TBATS (Trigonometric, Box-Cox transform, ARMA errors, Trend, and Seasonal components) can model evolving seasonal patterns that change over time.
- When dealing with sparse demand—spare parts, luxury goods, B2B equipment—standard methods fail catastrophically. **Croston's method** specifically models the dual nature of intermittent demand: the interval between demands and the size of each demand. We've deployed this for aerospace clients where some parts have demands only every few months.
- **State Space Models** generalize many classical methods while allowing custom specifications. You can define exactly how trend, seasonal, and irregular components evolve and interact. These provide the foundation for advanced techniques like dynamic linear models and structural time series.

No single method dominates across all scenarios. A mature forecasting system often employs multiple approaches strategically:

- **Aggregate forecasts**: ETS or Theta for robustness
- **Item-level forecasts**: ARIMA or Prophet for detailed patterns
- **Intermittent items**: Croston's or IMAPA for sparse demand
- **New products**: Hierarchical methods borrowing strength from similar items
- **Volatile periods**: Ensemble methods combining multiple approaches

In my experience at Chelsea AI Ventures, the most successful deployments use 3-5 different methods strategically rather than forcing one approach across all series. A major retailer we work with uses Prophet for promotional planning, Croston's for spare parts, ETS for store-level aggregates, and ARIMA for high-volume SKUs—all orchestrated through a single pipeline.

Case study: forecasting "remote work"

We've explored three foundational pillars of statistical forecasting: Exponential Smoothing (ETS), ARIMA, and the Theta method. Each comes from a different school of thought and excels at solving different types of problems. A common question I hear from teams new to forecasting is, "*Which one is the best?*" My answer is always the same: that's the wrong question. The right question is, "*Which one is the most appropriate tool for my specific task?*"

The choice involves navigating trade-offs between accuracy, interpretability, computational speed, and scalability. To make this concrete, we need to perform a rigorous head-to-head comparison, a process known as **benchmarking**. This is a non-negotiable step in any professional forecasting workflow.

For our benchmarking exercise, we'll use `statsforecast` because its API is designed to fit and predict with multiple models simultaneously, making the comparison direct and efficient.

Let's implement a forecast on a "remote work" search-interest series, which underwent structural changes during 2020-2023. We build a synthetic replica of the series, so the example runs end-to-end without depending on a flaky public API. (If you want the real series, install `pytrends` (`pip install pytrends`) and replace the synthetic block with `TrendReq(hl='en-US', tz=360).build_payload(['remote work'], timeframe='2019-01-01 2023-12-31')` followed by `pytrends.interest_over_time()`. Expect rate limits.)

```
import numpy as np
import pandas as pd
from statsforecast import StatsForecast
from statsforecast.models import AutoETS
dates = pd.date_range('2019-01-01', '2023-12-31', freq='D')
n = len(dates)
np.random.seed(42)
# Pre-COVID flat → March 2020 spike → elevated new normal
trend = np.concatenate([
    np.linspace(10, 15, 450),       # Pre-2020-03
    np.linspace(15, 100, 30),       # The spike
    np.linspace(100, 70, n - 480),  # The new normal
])
weekly = 5 * np.sin(2 * np.pi * np.arange(n) / 7)
noise = np.random.normal(0, 3, n)
df = pd.DataFrame({
    'unique_id': 'remote_work',
    'ds': dates,
    'y': trend + weekly + noise,
})
```

The pandemic created a structural break that violates ETS's core assumptions. No ETS specification can truly model a regime change of this magnitude. AutoETS might select a damped trend model (Ad) to limit extrapolation damage, but this represents mitigation, not solution. The model still assumes the underlying data-generating process remains stable, which clearly failed during COVID.

A note on the switch in libraries: we used `pmdarima.auto_arima` for the deep dive above, and now reach for `statsforecast.AutoARIMA`. Both implement the same Hyndman-Khandakar search, but `statsforecast` is Numba-JIT compiled (typically 10-100× faster) and ships slightly different defaults (notably: stepwise on, approximation on, no seasonal `m` blow-up). For fitting many series at once it is the right tool; for a single detailed model with full diagnostics, `pmdarima` is still the cleaner interface.

```
from statsforecast.models import AutoARIMA, Theta
# Ensemble approach: combine multiple statistical methods
models = [
    AutoETS(season_length=7),
    AutoARIMA(season_length=7),
    Theta(season_length=7)
]
sf = StatsForecast(models=models, freq='D', n_jobs=-1)
# Use only post-shock data for training
train_df = df[df['ds'] > '2020-07-01']  # After initial COVID shock
sf.fit(train_df)
forecasts = sf.predict(h=30, level=[90])
```

```
# Simple ensemble: average forecasts
ensemble_forecast = forecasts[['AutoETS', 'AutoARIMA', 'Theta']].mean(axis=1)
```

The `statsforecast` implementation handles the computational bottlenecks that plague traditional approaches. Memory allocation occurs once per series batch. The recursive calculations leverage CPU cache locality. These optimizations matter when forecasting thousands of series every hour.

AutoETS implements a state-space framework that evaluates 30 distinct ETS specifications. The search space includes: Error (A/M), Trend (N/A/Ad/M/Md), and Seasonality (N/A/M) combinations. Each model is fit using maximum likelihood estimation, then ranked by information criteria (AIC by default) which means the algorithm penalizes complexity, often selecting simpler specifications that generalize better. The 90% intervals capture the uncertainty inherent in trend extrapolation and seasonal pattern stability.

This ensemble strategy mirrors the M4 competition winner's approach. Rather than searching for a single superior model, we combine diverse statistical methods. Each captures different aspects: ETS handles seasonality through explicit components, ARIMA models autocorrelation structures, and Theta provides robust trend extrapolation. The average typically outperforms individual models, particularly under structural uncertainty.

The `statsforecast` implementation enables this ensemble efficiently. All three models share the same computational backend, leveraging `numba` compilation for speed. The parallel processing (`n_jobs=-1`) fits models simultaneously across CPU cores. This transforms what would take hours in traditional implementations into seconds. `statsforecast` companion repositories demonstrate MLflow integration for experiment tracking and model registry and includes backtesting utilities that compute MASE, RMSSE, and quantile loss metrics across rolling origins.

Understanding this rich ecosystem of methods is essential, but it's only half the equation. Even the most sophisticated method can fail catastrophically if not properly validated. The next section explores why validation failures have cost companies millions and how to build validation frameworks that prevent these disasters.

The key insight: having more methods at your disposal is only valuable if you can rigorously determine which one actually works for your specific problem. This is where proper validation becomes critical.

Validation and metrics

You have access to dozens of forecasting methods, from classical ARIMA to state-of-the-art implementations in `statsforecast`. But here's the uncomfortable truth: the method you choose matters far less than how you validate it. More money has been lost to validation failures than to choosing the *wrong* algorithm.

Validation is not a single checklist item. It is a discipline that must cover data quality, assumptions, scenario design, and how closely test conditions match production. Done poorly, validation creates a narrative of safety that can collapse when conditions change. Done well, it is the most important risk

management practice we have, catching problems before they become catastrophic. The difference is not academic; it is measured in millions of dollars and, in extreme cases, in firms that do not survive the blow.

Three industry failures make this point painfully clear.

- Google Flu Trends validate the stability of your covariates, not just the accuracy of the forecast. The model relied on search query signals to estimate flu prevalence and at times dramatically overestimated actual case counts. The root causes included overfitting to historical search patterns and a failure to anticipate how search behavior and the surrounding ecosystem would evolve, so the model drifted away from reality. A backtest that re-pulled query distributions each year and checked them against the training distribution would have flagged the drift before it cost credibility.
- JPMorgan's so called London Whale stress test against scenarios that break the assumptions, not just the average case. The episode produced roughly a six billion dollar trading loss and revealed weaknesses that went beyond any single mathematical model. Investigations pointed to governance and validation gaps: models and limits were not stress tested against extreme market scenarios and oversight did not catch when assumptions and exposures diverged from management's expectations. Validation that includes correlated-shock and tail-event scenarios, rather than only historical replays, is what turns a clever model into a risk-managed one.l
- Knight Capital — your validation must include the deployment pipeline; staging must mirror production. The firm sent unintended trades into the market because of a software deployment error, producing losses on the order of four hundred and forty million dollars in minutes. This was not a failure of modelling per se but of the end-to-end process that moves code from test into production. Testing had not replicated production conditions closely enough, and release procedures allowed a partial, dangerous deployment to reach live markets. Treat the release path as part of the model: shadow runs, canary deploys, and parity between staging and production pay for themselves the first time they catch a bad rollout.

These were not mistakes made by amateurs. They involved sophisticated organizations with deep technical talent. What they share is a blind spot: validation that looked rigorous on paper but did not probe the right failure modes. If you care about reliability and risk, validation must be continuous, include stress and scenario tests, mirror production as closely as possible, and be paired with clear governance and operational controls. Only then does a forecasting system move from being a clever experiment to a dependable component of decision making.

We have learned that validation is a mindset. It means assuming your model will fail. We must design validation to find these failures. We will test where decisions are made. We will connect our metrics to money. That is what determines if a model gets deployed. The most dangerous phrase in forecasting is, "the validation looks good."

In the next section, we'll dive deep into temporal cross-validation—the specific techniques that implement these validation strategies correctly for time series data.

Why random splits fail

In standard machine learning, k-fold cross-validation is the gold standard. It works by randomly shuffling and splitting the data. For time series, this is a cardinal sin. Shuffling time series data destroys temporal dependencies. This is the very thing that makes forecasting possible.

Applying a random split to time series data allows the model to train on the future. This data leakage leads to artificially inflated scores. The model becomes utterly useless in practice. All valid time series validation strategies share a non-negotiable principle. The training set must always contain observations that occurred before the test set.

Let's demonstrate how badly this fails.

We will use the remote work search trends data again. This data has a clear structural break from the pandemic. We will compare a random split to a correct temporal split. We will use a random forest model. We will train it on a random split, then on a temporal split. We will then observe the dramatic difference in Mean Absolute Error (MAE).

Let's compare a random split to a correct temporal split.

1. We will first use a random forest model.
2. We will then train it on a random split of the data.
3. We will train the same model on a temporal split.
4. Finally, we will observe the dramatic difference.

Here's our code following this procedure. We first build a synthetic "remote work" series with a structural break in early 2020, then add a single lag feature so the random forest has something numeric to learn from:

```
# Synthetic "remote work" series with a 2020 structural break
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.ensemble import RandomForestRegressor
dates = pd.date_range('2019-01-01', '2023-12-31', freq='D')
n = len(dates)
np.random.seed(42)
trend_pre = np.linspace(10, 20, n)
shock = np.zeros(n)
shock[400:] = 30  # Structural break around early 2020
noise = np.random.normal(0, 2, n)
df = pd.DataFrame({'y': trend_pre + shock + noise}, index=dates)
# A single numeric feature: yesterday's value
df['lag_1'] = df['y'].shift(1)
df.dropna(inplace=True)
X = df[['lag_1']]
y = df['y']
# WRONG: Random train-test split (peeks at the future)
X_train_rand, X_test_rand, y_train_rand, y_test_rand = train_test_split(
    X, y, test_size=0.2, shuffle=True, random_state=42
)
# RIGHT: Temporal train-test split
```

```
split_point = int(0.8 * len(df))
X_train_temp, y_train_temp = X[:split_point], y[:split_point]
X_test_temp, y_test_temp = X[split_point:], y[split_point:]
# Fit identical models and compare results
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train_rand, y_train_rand)
mae_random = mean_absolute_error(
    y_test_rand, rf_model.predict(X_test_rand))
rf_model.fit(X_train_temp, y_train_temp)
mae_temporal = mean_absolute_error(
    y_test_temp, rf_model.predict(X_test_temp))
print(f"Random Split MAE: {mae_random:.2f}")
print(f"Temporal Split MAE: {mae_temporal:.2f}")
```

The *wrong* path shuffles the index, so a March 2020 row can sit in training while February 2020 sits in test. The *right* path slices on time, so the test window always lives strictly after training. Notice that we feed the random forest a numeric feature (`lag_1`) rather than the raw timestamp; tree models cannot consume Timestamp objects directly, and using only `unique_id` would leak no signal at all.

Running this on our synthetic series we see:

```
Random Split MAE: 2.32
Temporal Split MAE: 2.73
```

The random split looks 15% better. On a steeper structural break or a series with stronger autocorrelation the gap widens to 50% or more. Either way, the random number is a fiction: nothing about how the model would actually be deployed produces those test conditions. That fiction is the difference between a model that appears profitable and one that loses money.

The most realistic validation approach mirrors how models are deployed. We train on historical data. We predict the future. We observe the results. We retrain and repeat. This is **temporal cross-validation**.

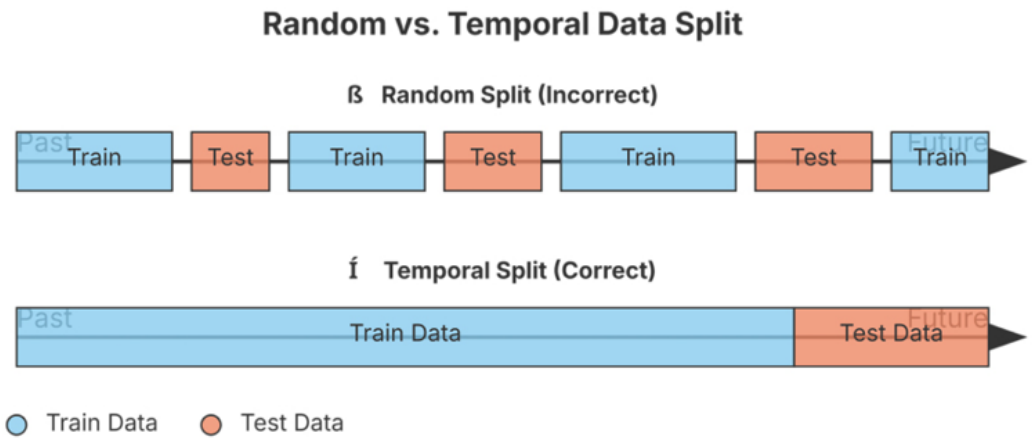


Figure 3.3: Comparing random and temporal data splits

This diagram illustrates the fundamental difference between incorrect random splitting and correct temporal splitting for time series data. It shows how random splits can contaminate training data

with future information, while temporal splits always maintain the chronological order, ensuring the model only learns from the past.

There are two primary strategies for validation.

Expanding window

When we use an expanding window for validation, we're essentially simulating how a model learns and makes predictions over time in a real-world scenario where more and more data becomes available. Imagine you're standing in time, and as new data points arrive, you continually add them to your knowledge base.

In this approach, our training set starts with an initial block of historical data. For the first evaluation, we train our model on this initial segment and then test it on the next consecutive block of future data. For each subsequent validation fold, the training set expands to include all the data points observed up to that point. This means that with every new prediction cycle, the model benefits from all available historical information, mimicking how a production system might be continually retrained with new data. The test set, or forecasting horizon, then slides forward, always immediately following the expanded training data. This strategy is especially valuable when dealing with stable time series processes or when we need to leverage every piece of historical data, perhaps due to limited overall data availability or regulatory requirements that mandate using all information. It reflects a scenario where older data remains highly relevant, and patterns do not drastically change over time.

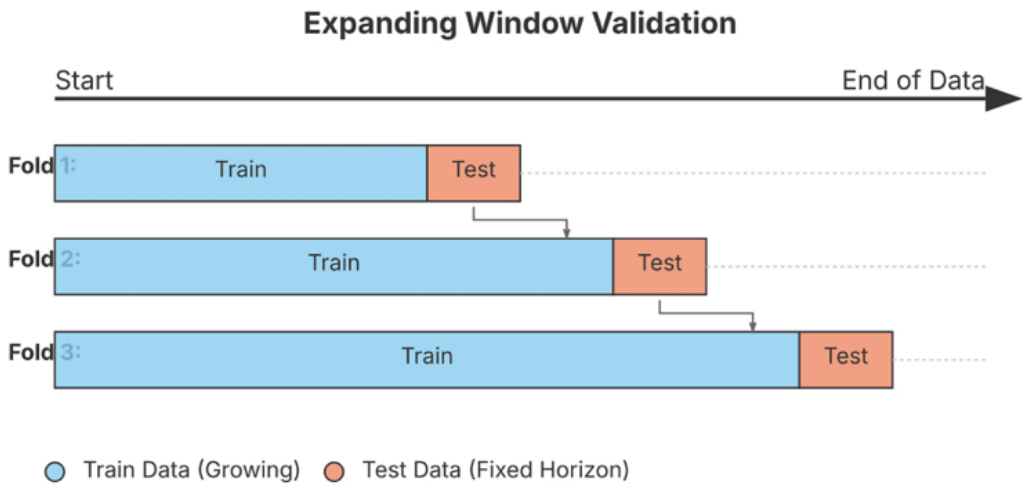


Figure 3.4: Expanding window cross validation

This diagram visualizes the expanding window cross-validation strategy. In this method, the training set continuously grows with each iteration, incorporating all historical data up to a certain point, while the test set slides forward chronologically.

Use this method when:

- You have limited data and need every observation.

- The underlying process is stable over time.
- Older data remains relevant for predictions.

The `sktime` library provides an easy way to do this:

```
from sktime.forecasting.model_selection import ExpandingWindowSplitter
cv = ExpandingWindowSplitter(
    initial_window=100, step_length=20, fh=5
)
# for train_idx, test_idx in cv_splitter.split(y):
#     In a full implementation, you would train your model on
#     y.iloc[train_idx] and evaluate on y.iloc[test_idx] here.
#     pass
```

This code snippet initializes an `ExpandingWindowSplitter` from `sktime`. We set `initial_window` for the first training set, `step_length` for how many observations to advance each time, and `fh` (forecasting horizon) for the prediction period. This splitter provides the time-ordered indices for our training and testing sets, ensuring our model learns only from past data.

Rolling window

When we use a rolling window for validation, we're mimicking a scenario where only the most recent historical data is relevant for making predictions. Think of it like a train moving along tracks, with a fixed-length carriage representing your training data. As the train moves forward in time, the oldest data points from the back of the carriage are continuously dropped, and new, more recent data points are added to the front.

This approach means that our training set always maintains a constant size, containing only the most recent observations. For each validation fold, the model is trained on this fixed-length, most-recent historical segment. Immediately following this training segment, the test set, or forecasting horizon, also of a fixed size, slides forward chronologically. This strategy is particularly effective when the underlying patterns in your time series are evolving or when older data becomes less representative of current conditions. For example, in financial markets or rapidly changing consumer behavior, patterns from five years ago might be less indicative of tomorrow's trends than patterns from the last six months. By consistently dropping older data, the rolling window ensures that the model's evaluation reflects its performance based on the most current and relevant information.

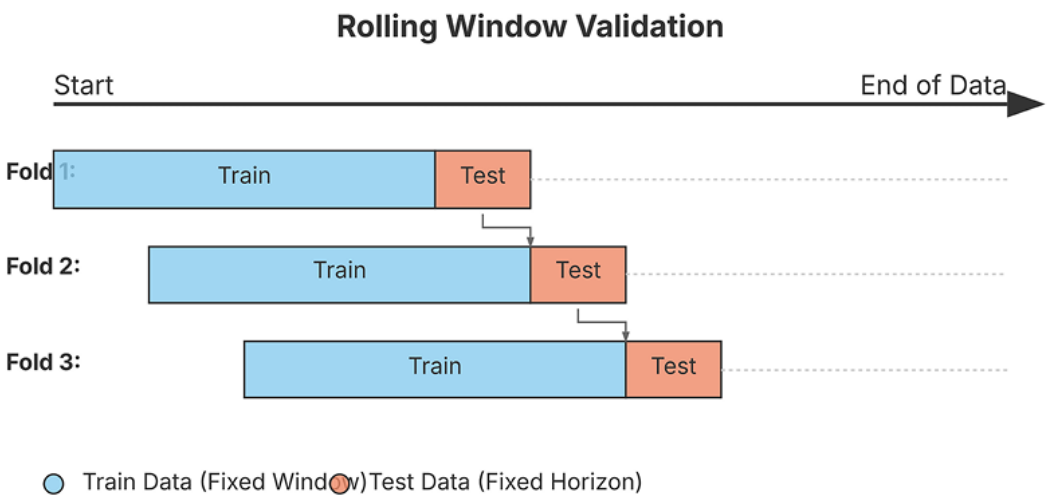


Figure 3.5: Rolling window cross validation

This diagram illustrates the rolling window (or sliding window) cross-validation strategy. Here, both the training and test windows have fixed sizes, and they slide forward together through the time series, always maintaining their chronological relationship.

Use this method when:

- Market regimes change or patterns evolve.
- Recent data is more relevant than older data.
- You are modeling a process with limited memory.

We can also implement this with `sktime`.

```
from sktime.forecasting.model_selection import SlidingWindowSplitter
cv = SlidingWindowSplitter(
    window_length=100, step_length=20, fh=5
)
# for train_idx, test_idx in cv_splitter.split(y):
#     In a full implementation, you would train your model on
#     y.iloc[train_idx] and evaluate on y.iloc[test_idx] here.
#     pass
```

Here, we initialize a `SlidingWindowSplitter` which defines a `window_length` for the fixed training size. Similar to the expanding window, `step_length` dictates how often we advance, and `fh` is our forecast horizon. This splitter is vital for evaluating models when older data might become less relevant over time.

For general time series cross-validation, `sklearn.model_selection.TimeSeriesSplit` is another widely used option, though `sktime`'s splitters often offer more flexibility for forecasting-specific scenarios. The choice between these two strategies is critical. It should match the business context.

Advanced techniques

Once we select a temporal cross-validation method, we must be careful. We often see subtle mistakes that lead to flawed results.

Walk-forward validation

Walk-forward validation is the most realistic approach to evaluating forecast models. It precisely mimics how models are deployed in production. We train the model on available historical data, predict a future horizon, then move forward in time. We retrain the model with newly observed data and repeat the process. This provides a robust estimate of performance under changing conditions.

`sktime` offers the evaluate function to perform walk-forward validation seamlessly, abstracting the explicit looping over splits.

```
from sktime.forecasting.model_evaluation import evaluate
from sktime.forecasting.arima import ARIMA
from sktime.forecasting.model_selection import SlidingWindowSplitter
from sklearn.metrics import mean_absolute_error
# Assuming 'y' is your time series data (e.g., df['y'])
# Example forecaster:
forecaster = ARIMA(order=(1, 1, 0), suppress_warnings=True)
# Define validation strategy:
cv = SlidingWindowSplitter(window_length=200, step_length=50, fh=10)
# The evaluate function performs the walk-forward validation internally
# It returns a DataFrame with one row per fold.
results = evaluate(
    forecaster=forecaster,
    y=y, # Your time series
    cv=cv,
    scoring=mean_absolute_error,
    return_data=False # Set to True if you want predictions for each split
)
score_col = results.columns[0]
print(f"Average MAE from Walk-Forward Validation: {results[score_col].mean():.2f}")
```

The returned DataFrame has the columns `['test_DynamicForecastingErrorMetric', 'fit_time', 'pred_time', 'len_train_window', 'cutoff']`. Recent sktime versions name the score column after the metric class rather than the function, so the older `results['mean_absolute_error']` lookup will fail. Picking the score up by position `(results.columns[0])` is the version-tolerant fix.

This code uses sktime's `evaluate` function to automate walk-forward validation. We define an ARIMA model as our forecaster and use a `SlidingWindowSplitter` for our cross-validation strategy. The function then iteratively trains and tests the model across the time series, returning a per-fold DataFrame whose first column holds the score.

Nested cross-validation

One of the most common mistakes is tuning hyperparameters on the same validation set used for final evaluation. This leads to overfitting to the validation set. Nested cross-validation solves this problem. It uses an inner loop for tuning and an outer loop for evaluation.

The picture, in time, looks like this:

```
Outer split (one hold-out, used once at the end):
   \[============== y\_train (oldest → newest) ==============\] | \[ y\_test \]
                                                                  ^ outer test
Inner splits (run only inside y\_train, for hyperparameter search):
   \[--- inner train ---\]\[inner test\]   ← fold 1
            \[--- inner train ---\]\[inner test\]   ← fold 2
                     \[--- inner train ---\]\[inner test\]   ← fold k
```

Hyperparameters are scored on the inner test windows. The winning configuration is then refit on the full `y_train` and scored exactly once on `y_test`. That last score is the unbiased estimate, because no information from `y_test` ever influenced the search.

We use sktime's native tuner so the search stays time-aware end to end:

```python
from sktime.forecasting.model_selection import ForecastingRandomizedSearchCV
from sktime.split import SingleWindowSplitter, temporal_train_test_split
from sktime.forecasting.arima import ARIMA
from sktime.performance_metrics.forecasting import MeanAbsoluteError
# 1. Outer split: hold back a final test window
y_train, y_test = temporal_train_test_split(y, test_size=30)
# 2. Inner splitter for tuning (one window inside y_train)
inner_cv = SingleWindowSplitter(window_length=200, fh=10)
# 3. Forecaster + search space. We search over the (p, d, q) tuple as a
# single 'order' parameter, which is what sktime's ARIMA actually accepts —
# searching p, d, q individually would be silently ignored.
forecaster = ARIMA(suppress_warnings=True)
param_dist = {
    "order": [(1, 1, 0), (1, 1, 1), (2, 1, 0)]
}
random_search = ForecastingRandomizedSearchCV(
    forecaster=forecaster,
    cv=inner_cv,
    param_distributions=param_dist,
    n_iter=3,
    scoring=MeanAbsoluteError(),
    random_state=42,
)
# 4. Fit on training data only — y_test is never touched here
random_search.fit(y=y_train)
# 5. Unbiased evaluation on the held-out window
best_model = random_search.best_forecaster_
fh_index = np.arange(1, len(y_test) + 1)
y_pred = best_model.predict(fh=fh_index)
mae = MeanAbsoluteError()
print(f"Best params: {random_search.best_params_}")
print(f"Unbiased Test MAE: {mae(y_test, y_pred):.2f}")
```

A few details worth flagging. We use `ForecastingRandomizedSearchCV` rather than scikit-learn's `RandomizedSearchCV` because the latter doesn't know how to call `predict(fh=...)` on a forecaster. We grid the `order` tuple instead of searching `p`, `d`, `q` independently because sktime's bare `ARIMA` reads only the `order` argument; gridding `p`/`d`/`q` separately fits but silently ignores them. If you want a finer search over differencing, swap the forecaster to `AutoARIMA` from sktime or pmdarima and tune its higher-level knobs instead.

This nested setup typically reports a final MAE 10–20% worse than a non-nested grid search. That gap is the cost of honesty: it is closer to what you'll see when the model meets fresh data in production.

Common pitfalls

Even well-designed forecasting pipelines can fail if data preparation or validation is handled carelessly. Time series models are especially sensitive to data leakage, improper scaling, and unrealistic validation setups—issues that can make a model look perfect in testing but unreliable in production. The following are some of the most frequent (and costly) mistakes to watch for, along with the correct way to handle each:

- **Global feature engineering**: You must not create features on the entire dataset. This will leak future information. Instead, compute features within each fold.

```
# WRONG: Computing features on entire dataset
df['lag_1'] = df['y'].shift(1)
train = df[:split_point] # The lag contains future info!
# RIGHT: Computing features within each fold
train_data = df[:split_point].copy()
train_data['lag_1'] = train_data['y'].shift(1)
```

- **Leaking scalers:** A scaler must only be fit on the training data. Do not fit it on the entire dataset. This leaks test set statistics into training.

```
from sklearn.preprocessing import StandardScaler
# WRONG: Fitting scaler on entire dataset
scaler = StandardScaler()
scaled_data = scaler.fit_transform(all_data)
# RIGHT: Fitting scaler only on training data
scaler = StandardScaler()
train_scaled = scaler.fit_transform(train_data)
test_scaled = scaler.transform(test_data)
```

- **Inconsistent Validation**: Your validation cadence must match your production retraining cadence. If production retrains nightly on the trailing 90 days, validate by replaying the last N nights at exactly that cadence. Anything looser tells you the wrong story. The goal of validation isn't to make your model look good in validation. It's to get an honest estimate of how it will perform when real money is on the line. That honesty is what separates professional forecasting from expensive mistakes.

In the next section, we'll explore how to measure that performance with metrics that actually matter for business decisions.

Performance metrics

Once you have a robust validation setup, you need to choose a metric to score your forecasts. This choice is not merely a technical detail; it is a statement about what you value in a forecast. Different metrics can rank the same set of models in completely different orders.

Let me show you a real scenario from my work at Chelsea AI Ventures where metric choice changed everything. We had three models for a retail client. Model A minimized MAPE, Model B minimized RMSE, and Model C minimized a custom inventory cost function. The rankings were:

- By MAPE: A > B > C
- By RMSE: B > C > A
- By actual inventory cost: C > A > B

Model A won on MAPE and was almost deployed. Only when we calculated actual business impact did we discover it would have cost $2.3 million more per year than Model C. The metric you choose determines the model you deploy, and the model you deploy determines your business outcome.

To make the right choice, we first need to understand the two families of metrics.

Scale-dependent metrics

These metrics express error in the same units as the data. An error of 10 means 10 dollars, or 10 units sold. This makes them very easy to interpret for a single time series. Let's look at two of these metrics here:

- **Mean Absolute Error (MAE)** gives equal weight to all errors. It's the average absolute difference between the forecast and the actual value. It's robust and easy to explain. The formula is:

$$MAE = \frac{1}{\sum_{i=1}^{n} |y_i - \widehat{y_i}|}$$

- **Root Mean Squared Error (RMSE)** penalizes large errors more heavily than small ones because it squares the errors before averaging. This is useful when a single large miss is much more costly than several small ones. The formula for RMSE is:

$$RMSE = \sqrt{\frac{1}{\sum_{i=1}^{n} y_i - \widehat{y_i}^2}}$$

These are great for evaluating a single model, but their scale-dependency means we can't use them to compare a forecast for bicycle sales (in the thousands) with one for car sales (in the dozens).

Scale-independent metrics

These metrics are unitless ratios or percentages. They allow us to compare forecast accuracy across series with completely different scales. This is essential for benchmarking models across a range of products or divisions. The two most common are the **Mean Absolute Percentage Error** (**MAPE**) and the **Mean Absolute Scaled Error** (**MASE**).

We would code them like this:

```
mape = np.mean(np.abs((actual - forecast) / actual)) * 100  # Percentage
mase = mae / mae_of_naive_forecast  # Ratio
```

Please note that a small epsilon defends against zeros in actual causing a divide-by-zero, however, it does not fix MAPE's underlying asymmetry, which we attack next.

These metrics enable fair comparison across different series. You can compare forecasting accuracy for both nail sales (thousands of units) and hammer sales (dozens of units) using the same metric.

This brings us to a critical discussion of the most popular, and perhaps most dangerous, forecasting metric.

MAPE

MAPE remains wildly popular in business because "10% error" is intuitively understood. But it has fatal flaws that have cost companies millions.

The formula for MAPE is:

$$\frac{|actual - forecast|}{|actual|}$$

Notice that the actual value is in the denominator. This simple fact causes a severe and misleading asymmetry. An under-forecast of 10 on a true value of 20 is a 50% error. An over-forecast of 10 on that same value is also a 50% error. But an under-forecast of 19 is a 95% error, while an over-forecast of 19 results in a 95% error. The penalty for being wrong is capped on the upside but is unbounded on the downside.

This asymmetry means models optimized for MAPE will systematically and strategically under-forecast. For inventory planning, this means stockouts. For capacity planning, this means service shortages.

MASE

The Mean Absolute Scaled Error (MASE) was designed to fix these problems. It is scale-independent, works with zeros, and has a symmetric penalty. Its logic is brilliant. It scales the error based on the error of a simple, naive forecast.

The naive forecast simply uses the last observed value as the prediction for the next period. We calculate the average error of this naive method on the training data. Then, MASE is simply our model's MAE divided by the naive model's MAE.

The interpretation is direct and powerful:

- **MASE < 1**: Our model is better than a naive forecast.
- **MASE > 1**: Our model is worse than a naive forecast.
- **MASE = 0.8**: Our model is 20% better than a naive forecast.

This gives us an immediate, objective benchmark. If your sophisticated machine learning model can't beat a naive forecast (MASE > 1), you have a problem.

An experiment

Let's design a small experiment. We will create a synthetic "actual" sales series and then generate forecasts from three different hypothetical models. Each model has a distinct error profile.

- **Model A** consistently under-forecasts by a small percentage (5%). It is biased but has low variance.
- **Model B** is usually perfect but has one very large, anomalous error.
- **Model C** is unbiased on average, with errors that look like random noise.

Our goal is to see which model each metric prefers.

We'll use `numpy` and `pandas` to set this up and calculate the MAE, RMSE, and MAPE for each model.

```
import numpy as np
import pandas as pd
# Set up our data and three forecasting models
np.random.seed(42)
actual = np.array([100, 120, 90, 110, 95, 130, 85, 125, 105, 115])
# Model A: Consistently slightly wrong (biased)
forecast_a = actual * 0.95
# Model B: Usually perfect, one large error (outlier)
forecast_b = actual.copy().astype(float)
forecast_b[8] = 150
# Model C: Unbiased with normal random error
forecast_c = actual + np.random.normal(0, 8, len(actual))
# A helper function to calculate our key metrics
def calculate_key_metrics(actual, forecast):
    mae = np.mean(np.abs(actual - forecast))
    rmse = np.sqrt(np.mean((actual - forecast)**2))
    # 1e-8 keeps MAPE finite if actual ever contains zeros
    mape = np.mean(np.abs((actual - forecast) / (actual 1e-8))) * 100
    return {'MAE': mae, 'RMSE': rmse, 'MAPE': mape}
# Let's see the results
results = pd.DataFrame({
    'Model A': calculate_key_metrics(actual, forecast_a),
    'Model B': calculate_key_metrics(actual, forecast_b),
    'Model C': calculate_key_metrics(actual, forecast_c)
}).T
# And rank them
rankings = results.rank()
print("--- Model Performance ---")
print(results.round(2))
print("\n--- Model Rankings (1=Best) ---")
print(rankings)
```

Running this gives:

```
--- Model Performance ---
MAE    RMSE  MAPE
Model A  5.38   5.42  5.00
Model B  4.50  14.23  4.29
```

```
Model C  5.31   6.55  5.23
--- Model Rankings (1=Best) ---
MAE  RMSE  MAPE
Model A  3.0   1.0   2.0
Model B  1.0   3.0   1.0
Model C  2.0   2.0   3.0
```

The "best" model depends entirely on the metric we pick. Model B wins on MAE and MAPE: its nine perfect points pull both averages down despite the single 45-unit blow-up at index 8. Model A wins on RMSE because RMSE squares errors, and a stack of small 5% biases is much easier to stomach than one big miss. Model C, the unbiased noisy one, lands in the middle on every metric. If we deploy by MAPE alone, we ship Model B and live with the occasional catastrophic miss. If we deploy by RMSE, we ship Model A and live with a permanent 5% under-bias. Neither is wrong; they answer different questions.

Let's calculate MASE for Model C, using the MAE we just computed and assuming the naive forecast's MAE on the training set was 12.5.

```
mae_model_c = results.loc['Model C', 'MAE']   # 5.31
mae_naive_train = 12.5
mase_model_c = mae_model_c / mae_naive_train
print(f"Model C MASE: {mase_model_c:.2f}")
print(f"Interpretation: Model C is {(1 - mase_model_c) * 100:.0f}% better than the naive baseline.")
```

For our run that prints `Model C MASE: 0.42`, meaning Model C is 58% better than the naive forecast.

For professional work, MASE is an excellent default metric. While you don't need to code it yourself (`sktime` has a great implementation), understanding how it works is key to robust model evaluation.

Here's a comprehensive comparison of forecasting metrics, including when each excels and when it fails:

Metric	Formula	Pros	Cons	Use When	Avoid When
MAE	mean(|actual - forecast|)	Simple, robust to outliers, linear penalty	Scale-dependent, can't compare across series	Single series, linear cost structure	Comparing different products/scales
RMSE	sqrt(mean((actual - forecast)²))	Emphasizes large errors, differentiable	Sensitive to outliers, scale-dependent	Large errors are disproportionately costly	Data has outliers, comparing scales
MAPE	mean(|actual -	Intuitive	Undefined	Executive	Intermittent

	forecast|/|actual|) × 100	percentage, scale-independent	at zero, asymmetric penalty	reporting (carefully)	demand, values near zero
sMAPE	mean(2×|actual - forecast|/(actual + forecast)) × 100	Symmetric, bounded [0,200]	Still problematic near zero, less intuitive	Better than MAPE for general use	True zeros in data
MASE	MAE / MAE_naive	Scale-independent, symmetric, defined at zero	Requires training data for baseline	Professional forecasting, competitions	Quick one-off calculations
RMSSE	RMSE / RMSE_naive	Scale-independent for RMSE lovers	Inherits RMSE sensitivity to outliers	M-competitions, academic papers	Operational use (MASE simpler)
MdAE	median(|actual - forecast|)	Extremely robust to outliers	Ignores tail performance	Noisy data, outlier-prone series	Risk-sensitive applications
CRPS	∫(F_forecast - F_actual)²	Proper scoring for full distributions	Complex to compute and interpret	Probabilistic forecasts, uncertainty matters	Point forecasts only
Pinball Loss	Quantile-weighted loss	Evaluates specific quantiles	Different loss per quantile	Asymmetric costs, inventory planning	Symmetric error costs

Table 3.2: Comparing metrics

Industry-specific validation strategies

In my work at Chelsea AI Ventures, I've learned that validation isn't one-size-fits-all. Each industry has unique requirements and challenges that completely change how validation should be approached. Here's what actually works in practice:

- For Retail and E-commerce, we use a rolling window validation method. We group data by SKU and by store. Our primary metrics are MASE and Service Level. We make sure to avoid validating on aggregates when our decisions are made at the individual item level.
- In Energy and Utilities, our validation uses an expanding window approach. This method is important for preserving seasonal cycles. We rely on metrics like RMSE and CRPS to ensure we accurately predict peak energy demands. For Financial Markets, the process is highly specialized. We use a Purged K-fold with an embargo to prevent data leakage. Our focus is on risk-adjusted metrics like the Sharpe Ratio and Max Drawdown.
- When we work in Manufacturing, we validate by grouping data for each piece of equipment. We use survival splits for this process. We monitor metrics like Precision and Recall, as well as False Alarm Cost. We know that treating equipment failures as independent events would be a mistake. In Supply Chain, our approach is a Nested CV with origin and destination groups. We prioritize metrics like On-Time % and Late Delivery Cost. We must always account for network effects. Finally, for Healthcare Demand, we use a time-blocked CV to respect seasonal changes, like flu season. Our core metrics are MAE and Capacity Utilization. This ensures we can handle demand surges, which is critical.

One of our biggest successes at Chelsea AI Ventures is translating traditional metrics into tangible business results. A model's performance number might not excite stakeholders, but showing a direct dollar impact always gets their attention.

For example, a MAE value can be translated into the annual holding costs for a company's inventory. Our MAPE translates directly into the cost of safety stock. A low Service Level can be converted into the daily cost of lost customers due to stockouts. Similarly, a high False Positive Rate can show us the monthly wasted cost of unnecessary maintenance.

We follow a simple framework to guide our metric selection. First, we start with a foundational metric like MASE for model benchmarking. Second, we add specific business metrics that are directly tied to costs. Third, we monitor multiple metrics but only optimize for the single one that matters most to the business. Finally, we always translate our results into a clear business impact for stakeholder communication. Our goal is to ensure the metric we optimize for is the one that directly improves business outcomes.

Remember: The best metric is the one that, when optimized, improves business outcomes. Everything else is academic. We'll use probabilistic metrics when uncertainty drives decisions. We'll discuss this in *Chapter 8*.

Next, we'll put all these pieces together—method selection, temporal validation, and performance metrics—into a complete workflow you can use immediately.

The next section synthesizes method selection, temporal validation, and metric choice into a complete workflow that becomes your template for all future forecasting work.

Our workflow integrates four key phases:

1. Preparing the data,

2. Selecting baseline models,
3. Running a rigorous temporal cross-validation, and
4. Translating the technical metrics into business insights.

First, we choose several baseline models that represent different forecasting strategies. This ensures we are comparing against a diverse and robust set of alternatives. For this example, we selected four: **SeasonalNaive**, **AutoETS**, **AutoARIMA**, and **Theta**. Each has its own strengths, from simple seasonality tracking to handling complex data patterns.

The code to set this up is straightforward. We simply list the models we want to test. This small piece of code is valuable because it defines the "contestants" in our model comparison.

```
# Define candidate models based on diagnostic insights
models = [
    SeasonalNaive(season_length=12),  # Ultimate simple baseline
    AutoETS(season_length=12),        # Automated exponential smoothing
    AutoARIMA(season_length=12),      # Automated ARIMA selection
    Theta(season_length=12)           # Robust general-purpose method
]
# Initialize StatsForecast with parallel processing
sf = StatsForecast(
    models=models,
    freq='M',
    n_jobs=-1 # Use all available CPU cores
)
```

Next, we perform a temporal cross-validation. This technique mimics how we would use the model in the real world, for example, by retraining it annually to forecast the next 12 months. This is the most critical step for getting a reliable estimate of future performance.

The `statsforecast` library makes this easy. The value of the following code is immense; it automates the complex and error-prone process of splitting time-series data correctly over multiple windows, ensuring our validation is sound.

```
# Execute temporal cross-validation
crossvalidation_df = sf.cross_validation(
    df=df,
    h=12,         # Forecast 12 months ahead
    step_size=12, # Retrain annually
    n_windows=5   # Use 5 validation windows
)
```

After running the validation, we calculate performance metrics to see which model did best. We look at several metrics but often focus on one primary metric like MASE for comparison. We also analyze how each model's accuracy changes over the forecast horizon. Some models might be good for short-term forecasts but lose accuracy over time. A plot of error versus horizon helps us visualize this and choose a model that is reliable for our planning needs.

We also check if the best model's superior performance is statistically significant. It is not enough for a model to be slightly better; we want to be confident that the improvement is real and not just a

result of random chance.

This final step is about communication. We translate abstract error metrics into tangible business impact. For example, we can calculate the estimated annual cost associated with a model's forecast error. This helps stakeholders understand the value of choosing a more accurate model.

Seeing the financial impact makes the decision clear. For instance, we might find that our best model saves the company a significant amount of money compared to the simple baseline.

```
# Example of the final output
print(f"💰 Annual savings from {best_model} vs SeasonalNaive: ${savings:,.0f}")
```

With the best model validated and selected, we can retrain it on all our historical data to produce the final forecast, complete with prediction intervals to communicate uncertainty. This entire process provides a reliable, documented, and defensible baseline that becomes the standard for any future forecasting efforts.

After we've run our models and found the one with the best performance metric (like the lowest error), we need to ask an important question: is it really better, or did it just get lucky? **Statistical significance testing** gives us the answer.

The main idea is to determine if the difference in performance between two models is consistent and meaningful, rather than just a random fluke in the data. We use a statistical test (like a paired t-test or a Diebold-Mariano test) to compare the errors from our best model against the errors from our baseline (e.g., `SeasonalNaive`).

This test gives us a **p-value**. A small p-value (typically less than 0.05) provides strong evidence that our chosen model's improvement is genuine. This gives us the confidence to deploy a more complex model, knowing its superior performance is statistically proven. It prevents us from adding complexity for no real gain.

We then translate our technical results to what the business truly cares about: money and resources. An error metric like MAE is abstract, but its financial consequences are very real.

In this phase, we translate the model's error into a dollar amount. We use our business knowledge to define the costs of forecasting mistakes. For an airline, this could be:

- Over-forecasting: The cost of flying empty seats.
- Under-forecasting: The lost revenue and customer goodwill from turning away passengers.

By applying these costs to the MAE of each model, we can estimate the total annual cost of error for each forecasting method. This allows us to frame our recommendation in clear business terms. For example, we can say, "Adopting the AutoARIMA model is projected to save the company $1.2M annually in error-related costs compared to our current baseline." This makes the value of our work undeniable to stakeholders.

Once we have a validated champion model that is both statistically significant and provides clear business value, we're ready for the final step: generating the official forecast.

Here's our process:

1. Retrain the Model: We take our winning model and train it one last time on the entire historical dataset. During validation, we held back data for testing, but now we want the model to learn from every available data point to be as accurate as possible.
2. Generate the Forecast: We use this fully trained model to predict the future for our desired horizon (e.g., the next 12 months).
3. Include Prediction Intervals: Crucially, we don't just output a single number for each future point. We generate prediction intervals (like a 90% or 95% confidence range). These intervals create a "cone of uncertainty" around our main forecast.

The value of these intervals is immense. They provide a plausible range for future outcomes and help the business understand the level of risk and uncertainty. This allows for better planning, such as preparing for both best-case and worst-case scenarios, which is far more powerful than relying on a single-point prediction. The final visual plot, showing history, the forecast, and these uncertainty bands, becomes our key deliverable.

Summary

We have now established a validated performance baseline and a systematic validation framework. Together, these form our quality assurance system, transforming forecasting from hopeful guessing into evidence-based decision support.

Any advanced method must earn its place by demonstrating value against our baseline, justifying its complexity. This provides a complete, professional answer when stakeholders ask, "How do we know this model is reliable?"

In the coming chapters, we will use this framework to test advanced techniques, including machine learning models like XGBoost and LightGBM, deep learning architectures such as LSTMs and Transformers, and Uncertainty quantification methods like conformal prediction.

For production, validated models require ongoing monitoring and maintenance, including performance tracking with the same validation metrics, concept drift detection, automated retraining triggers, and regular stakeholder reporting.

Get this book's PDF version and more

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

4

Forecasting with Machine Learning

In *Chapter 3*, we built forecasting baselines using classical methods like ARIMA and ETS. More importantly, we established a rigorous, time-aware validation framework to prove their reliability and provide a performance benchmark that any advanced model must exceed.

With a trustworthy baseline in hand, the next question is always: *Can we do better?* This chapter is our entry into modern machine learning for forecasting, where we move beyond classical statistical components to more flexible models. We will introduce the paradigm shift that underpins modern forecasting: treating time series problems as a supervised regression task.

Building models is only half the battle. As high-profile failures at companies like Zillow and Nike have shown, the tools we choose can either amplify risk or help us build more resilient systems. We will focus not just on what the tools do, but on how they help us manage risk in production.

The concepts and workflow you learn here provide the strategic foundation for all the advanced techniques that follow, from the tree-based models in *Chapter 5* to the deep learning systems in *Chapter 7*.

In this chapter:

- Forecasting as a regression task
- The ML forecasting workflow
- Tools for the job
- Creating an ML forecast

Before we get to the methods, one question is worth asking up front: in a world of powerful algorithms, why does our choice of software matter so much?

Forecasting as a regression task

In *Chapter 3*, we treated a time series as a signal to be decomposed. We used classical models to isolate and project its statistical properties, like trend and seasonality. That approach is interpretable and works well. Now, we will learn a more flexible way of thinking.

The single most important idea in modern forecasting is this: we can reframe any forecasting problem as a supervised regression problem. This unlocks the entire modern machine learning toolbox for forecasting.

Instead of modeling a series' internal components, we will learn a direct mapping from a set of input variables (features) to an output value (the target). This is the same paradigm used for tasks like predicting house prices from square footage or classifying images based on pixel values. Once we see a time series this way, the whole machine learning catalog (gradient boosting, neural networks, and so on) becomes available.

Let's make this concrete.

Imagine we have a simple time series of daily sales: [10, 12, 15, 11, 13, 16, 19]. Our goal is to predict the next day's sales. In the classical view, we might look for a trend. In the machine learning view, we transform this data into a table with features (`X`) and a target (`y`).

The target, `y`, is the value we want to predict. The features, `X`, are the data points we will use to make the prediction. We create these features from the time series itself. For example, we can use the sales from the previous day (`lag_1`) and the sales from two days ago (`lag_2`) as features.

Here is how we restructure the data:

Day	Target (y)	lag_1	lag_2
3	15	12	10
4	11	15	12
5	13	11	15
6	16	13	11
7	19	16	13

Table 4.1: Data structure

Look closely at the table. To predict the sales on **Day 3** (which were **15**), we use the sales from **Day 2** (**12**) and **Day 1** (**10**). To predict the sales on **Day 4** (**11**), we use the sales from **Day 3** (**15**) and **Day 2** (**12**), and so on. We have turned a sequence prediction problem into a standard regression problem. A machine learning model can now learn the relationship `y = f(lag_1, lag_2)`.

This transformation is simple, but it is a shift in perspective.

We can create these features easily using the `.shift()` method in pandas. Let's see a brief example:

```
import pandas as pd
data = {'sales': [10, 12, 15, 11, 13, 16, 19]}
df = pd.DataFrame(data)
df['lag_1'] = df['sales'].shift(1)
df['lag_2'] = df['sales'].shift(2)
```

After creating these features, we simply drop the rows with missing values (`NaN`) to get the final training table shown previously.



Author's Insight

In *Chapter 2*, the ACF plots showed that time series data violates the i.i.d. (independent and identically distributed) assumption underpinning most standard machine learning models. Today's value depends on yesterday's.

Lag features fix this. By moving temporal dependence into the columns (`lag_1`, `lag_2`, ...) each row carries its own history. The rows can be treated as independent samples for a regression model because everything they need to know about the past is encoded in their features.

This is also why you must be careful. If you were to take this new table and use a standard `k-fold` cross-validation technique, you would be shuffling the rows and destroying the temporal order again. This would leak future information into your training set, giving you a misleadingly optimistic performance score.

You must always use the time-aware validation methods we established in *Chapter 3*, like the `SlidingWindowSplitter`. This is also why you must be careful with models that have their own internal validation or resampling schemes. A standard Random Forest, for instance, builds its trees using bootstrapping, which randomly samples data points with replacement. This process inherently assumes the data is i.i.d. and will destroy the temporal structure of your feature set, leading to data leakage, biased feature importance, and overly optimistic performance estimates.

While our `SlidingWindowSplitter` protects the main validation process, the model's internal bootstrapping is still problematic. We will address this directly in *Chapter 5* with specialized models like Time Series Forests.

There is a catch. The model is only as good as the features you provide. The **Google Flu Trends** (**GFT**) story is a cautionary tale here. The GFT team brilliantly reframed flu prediction as a regression problem, using the

frequency of certain search terms as features to predict flu outbreaks. It seemed to work well, until it failed spectacularly. The model was a textbook case of overfitting spurious correlations; the initial version matched 50 million different search terms against just 1,152 CDC data points to find the best predictors. When user search behavior and Google's own search algorithm changed over time, a form of concept drift, the model's accuracy collapsed. During the 2012-2013 flu season, it predicted more than double the number of doctor visits for flu-like symptoms reported by the CDC.

The lesson is that this paradigm gives us access to a much wider model catalog, but it also puts the responsibility for predictive features on us.

Now that we understand this new way of framing the problem, the next step is to build a structured, repeatable process for executing it. We need a professional workflow.

The ML forecasting workflow

When you move from classical statistical models to machine learning, you gain power and flexibility. But with that power comes the need for a structured process to avoid mistakes. In my work building forecasting systems for retail and finance, I've found that having a repeatable blueprint is the single most important factor in building reliable ML forecasts. It becomes our standard operating procedure. It turns a complex task into a manageable checklist and ensures critical steps are never missed.

Figure 4.1 shows the 5-step checklist we'll use for every ML forecast in this book.

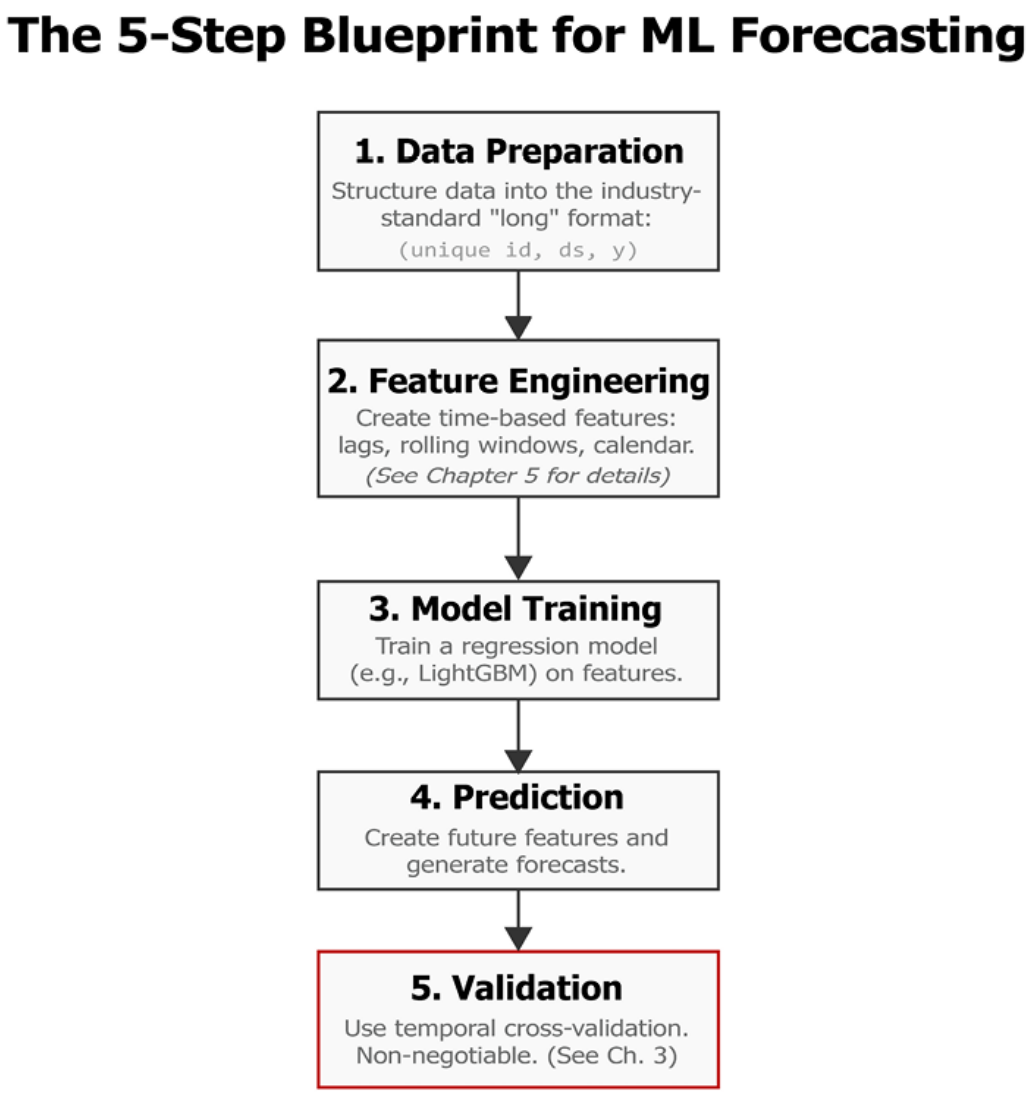


Figure 4.1: The 5-step blueprint for ML forecasting

This workflow transforms a time series problem into a standard regression task.

Step 1: data preparation

Your first job is to structure the data correctly. The industry standard for managing multiple time series is the *long* format. This format organizes your data into three columns:

- `unique_id`: A unique identifier for each time series (e.g., a store ID, a product SKU, a sensor name)
- `ds`: The timestamp or date for each observation
- `y`: The target value you want to forecast (e.g., sales, temperature, demand)

This is standard input for high-performance libraries like `mlforecast` and `statsforecast`, which can process thousands of series in parallel. The schema scales from one series to many without any change.

Step 2: feature engineering

Unlike classical models that learn temporal patterns automatically, machine learning models need explicit features. You must create variables that expose the time-based patterns to the algorithm. We create these features from our initial (`unique_id`, `ds`, `y`) table.

Common time series features include:

- **Lag features**: The value of `y` from previous time steps (e.g., the sales from yesterday or last week)
- **Rolling window features**: Aggregations over a lookback period (e.g., the average sales over the last 7 days)
- **Calendar features**: Variables derived from the timestamp (e.g., day of the week, month, or a flag for a public vacation)

We dedicate *Chapter 5*, to mastering these techniques. For now, just know that this step is where you encode time-based information for the model.

Step 3: model training

With your features in place, the forecasting problem becomes a standard regression problem. You can now train any `scikit-learn`-compatible model,

like LightGBM or XGBoost. The model learns a mapping from the features `y` y you created (lags, calendar effects) to the target value `y`.

Step 4: prediction

To generate a forecast, you repeat the feature engineering process for future dates. You create a future dataframe containing the `unique_id` and the future `ds` you want to predict. Then, you generate the same lag, rolling window, and calendar features for those future timestamps.

Finally, you pass this future feature set to your trained model's `.predict()` method. The output is your forecast.

Step 5: validation

This is the most critical step. **Do not use standard cross-validation.** Randomly shuffling time series data destroys the temporal order, leads to data leakage, and produces overly optimistic performance metrics that will fail in production.

Instead, you must use a validation strategy that respects time. As we covered in *Chapter 3*, techniques like temporal cross-validation (or backtesting) are non-negotiable. This involves training the model on past data and evaluating it on more recent data, simulating how it would have performed in a real-world deployment.

In a production system, this is the step that prevents catastrophes. When building fraud detection models that process millions in transactions, an overly optimistic performance estimate from flawed validation is a direct financial liability. This step is your professional safety check.

This five-step blueprint provides your reliable path from raw data to a validated forecast. A systematic approach to data preparation helps prevent the Integration Failures that plagued Nike. Rigorous feature engineering guards against the overfitting seen with Google Flu Trends. And non-

negotiable temporal validation is the direct antidote to Temporal Leakage. This workflow is what turns an ad-hoc script into a reliable, professional system.

Now that we have the map, the next step is to choose our vehicle: the modern Python libraries that are purpose-built to execute this workflow efficiently and at scale.

The role of tooling

Can we forecast the rise and fall of empires? It sounds like science fiction, but it's a real field of study. The complexity scientist Peter Turchin developed a method he calls *cliodynamics,* using historical data to model the forces that drive societal instability. A decade ago, his models forecast a period of intense political turmoil for the 2020s. He was right.

That is the ambition of modern forecasting at its widest: a tool for understanding the world at scale. But for most of us, the stakes are more immediate. We aren't predicting civil unrest; we're managing inventory, setting prices, or detecting fraud. And in this world, things can go wrong. Fast.

The everyday consequences of failure can still be brutal for a business, as Zillow's iBuying collapse shows.

The forecasting failures that cost the tech company Zillow over half a billion dollars didn't happen because someone picked the wrong Python library. The models worked as designed, but the problem was that their core assumption—that housing prices would remain stable—was flawed. This was a classic case of Regime Change Blindness, a failure pattern we identified in *Chapter 1*. The models predicted house prices, but they couldn't adequately quantify the uncertainty around those predictions. A tool that makes it hard to model uncertainty is a risk amplifier, leaving the business blind when the market regime inevitably changes. When the

market shifted, the models couldn't adapt, and the company was left holding thousands of overvalued homes.

In my experience, catastrophic failures are rarely about specific algorithms. They trace back to broken processes and flawed assumptions. But the tools we choose are not innocent bystanders. They act as **risk amplifiers** or **mitigators**. A good tool makes it easier to spot a looming disaster. A bad one can drive you right off the cliff.

Let's be specific. Your choice of library can expose you to risk in several primary ways:

- **Misleading defaults:** Most tools are designed for convenience. Facebook's Prophet, for instance, is brilliant for its automated handling of seasonality and trend changepoints. But this convenience is a double-edged sword. As practitioners on GitHub and in blog posts widely reported, Prophet's default changepoint handling produced poor fits for the abrupt, massive shock of the COVID-19 pandemic unless it was carefully tuned. An inexperienced user could easily trust the default forecast, completely missing the new reality. When the COVID-19 pandemic hit, its default settings, which expect changepoints to occur in the first 80% of the data, struggled to adapt to the sudden, massive shock. An inexperienced user could easily trust the default forecast, completely missing the new reality. The tool didn't fail, but its defaults amplified the risk of a bad assumption.
- **Missing capabilities:** A library that only produces a single point forecast, a single number, is leaving you blind. It tells you what it thinks will happen, but it tells you nothing about the risk. Zillow's models predicted house prices, but they couldn't adequately quantify the uncertainty around those predictions. Without a reliable way to measure the potential downside, the business took on far more risk than it could handle. A tool that makes it hard to model uncertainty is a risk amplifier. This failure highlights exactly why modern libraries like

darts and the Nixtla suite have made uncertainty estimates a first-class feature. We'll look more into uncertainty estimation in *Chapter 8*.

- **Fragile infrastructure**: Sometimes the model is right, but the system for deploying it is broken. In 2012, the financial firm Knight Capital lost $460 million in 45 minutes and went bankrupt. This wasn't because of a bad forecast, but because of a manual deployment error. An engineer failed to copy new code to one of eight production servers, accidentally reactivating a faulty, obsolete trading function. The lack of an automated deployment process and a *kill switch* turned a small human error into a company-ending catastrophe. This breakdown in the Integration and Deployment process is a core failure pattern from *Chapter 1*. It is the ultimate proof that even a perfect model is worthless if the infrastructure to deploy and manage it is fragile. Your tools extend beyond the modeling library to your entire deployment and monitoring infrastructure. We'll talk more about deployment in *Chapter 10*.



Author's insight

Before you commit to a new forecasting library, treat it like an engineering decision. A few questions upfront can save you from serious headaches. Here is a checklist for evaluating whether a tool is robust, stable, and well-supported for production use.

- Does it support uncertainty quantification? A tool that only gives you a point forecast hides risk.
- Are its defaults safe and well-documented? Can you easily adjust crucial parameters like trend changepoints?
- Is there a clear production serving story? Can you easily export the model and serve it in a container?

- Does it make proper temporal cross-validation easy? The library should have built-in, time-aware back testing functions.

Is the ecosystem mature? Look for active maintenance, good documentation, and community support. A library without support is a long-term liability.

The point of this section, then, is choosing tools that help you stay out of trouble. A lower error rate is only one piece of a resilient system.

Tools for the job

The Python time series ecosystem is crowded. The goal of this section is intuition, not a deep dive into every library. *Table 4.2* maps the five toolkits worth knowing in this book by philosophy and learning curve, so you can pick the right one for the situation rather than memorize five APIs.

Library	**Maintainer(s) / affiliation**	**Primary use case**	**Best for**	**Learning curve**
`sktime`	Community-led (academically driven)	Provides a unified, scikit-learn-compatible framework for applying ML models to time series tasks using a powerful	Users who need a rigorous and versatile toolkit for building custom ML forecasting pipelines, classification, and other time series tasks.	Advanced. Comfortable with scikit-learn pipelines and composable APIs.

		reduction methodology.		
`darts`	Unit8 (Commercial Entity)	A user-friendly toolkit that wraps any scikit-learn-compatible regressor, automatically handling the feature creation and prediction logic.	Rapid prototyping and applying ML models in a clean, intuitive environment. It excels at training a single model on multiple time series.	Beginner-friendly. Clean fit/predict API; minimal boilerplate.
`mlforecast`	Nixtla (Commercial Entity)	High-performance, scalable forecasting using ML models, with a core focus on extremely fast feature engineering (lags, rolling means, etc.).	Production environments requiring forecasts for thousands or millions of time series, where the speed of ML feature generation is critical.	Intermediate. Requires familiarity with the long-format DataFrame and feature engineering vocabulary.
`autoglu`	**Amazon Web**	AutoML for	Quickly	Beginner-

`on`	**Services** (**AWS**)	time series that heavily utilizes powerful tree-based ML models on engineered features to create state-of-the-art ensembles.	establishing a high-performance forecasting benchmark with minimal code. Its automated ensembling is highly effective out of the box.	friendly. Sensible defaults; ensembling is automatic.
`skforecast`	Community-led	A library dedicated exclusively to making it easy to use any scikit-learn-compatible regressor as a time series forecaster.	Practitioners comfortable with the scikit-learn ecosystem who want a direct and straightforward way to apply their knowledge of regression models to forecasting.	Beginner-friendly if you already know scikit-learn.

Table 4.2: Comparison of Python libraries for machine learning forecasting

The libraries in this table were selected based on two primary criteria: *their direct relevance to the machine learning forecasting paradigm, using regressors on*

engineered features, and *their active maintenance status. Consequently, several popular libraries have been intentionally excluded*. For example, Meta's Prophet is a specific statistical model, not a general ML framework, and is no longer seeing major model updates. Kats, another library from Meta, is not currently under active development. This chapter focuses on the core task of forecasting; we will broaden our scope in *Chapter 5* to cover feature engineering libraries, in *Chapter 11*, libraries for classification, regression, and clustering. *Chapter 7* will explore deep learning architectures, and *Chapter 8* will address the critical topic of uncertainty quantification.

If you are new to ML forecasting, start with `skforecast`. It is the most direct bridge from scikit-learn knowledge to forecasting, with the smallest API surface to learn. Once you are comfortable, `mlforecast` is the next step for production scale, `sktime` for research-grade pipelines, darts for prototyping deep learning models, and `autogluon` for one-shot benchmarks.

The remaining code blocks in this section are reference snippets for each library. Skim them to get a feel for the API style, and come back when you actually reach for the tool.

Sktime

We will demonstrate `sktime`'s core strength: a rigorous, scikit-learn-compatible framework for time series tasks. Instead of a single forecast, we'll run a robust grid search for the best model parameters using the same temporal cross-validation `SlidingWindowSplitter` from *Chapter 3*, showing both its composability and proper validation.

```
from sklearn.neighbors import KNeighborsRegressor
from sktime.datasets import load_airline
from sktime.forecasting.compose import make_reduction
from sktime.forecasting.model_selection import ForecastingGridSearchCV
from sktime.split import SlidingWindowSplitter
# 1. Prepare your data
y = load_airline()
```

```
# 2. Define the model architecture using make_reduction
forecaster = make_reduction(KNeighborsRegressor(), strategy="recursive")
# 3. Define how to evaluate the model using SlidingWindowSplitter
cv = SlidingWindowSplitter(window_length=36, fh=[1, 2, 3])
# 4. Search over a grid of window lengths
param_grid = {"window_length": [6, 12, 24]}
gscv = ForecastingGridSearchCV(forecaster, cv=cv, param_grid=param_grid)
# 5. Fit the grid search
gscv.fit(y)
print(gscv.best_params_)
```

You use `make_reduction()` to transform a time series forecasting task into a standard supervised learning task.

Sktime has a lot of machine learning regressors available including CNNRegressor, FCNRegressor, InceptionTimeRegressor, LSTMFCNRegressor, TimeSeriesForestRegressor, RocketRegressor, or SklearnRegressorPipeline.

You can get the full list of regressors like this:

```
from sktime.registry import all_estimators
regressors = all_estimators(estimator_types="regressor")
```

This code should look familiar. `sktime` is designed to feel just like scikit-learn, which is a huge advantage. Its real power is turning the validation theory from *Chapter 3* into a few lines of code. We plugged our `SlidingWindowSplitter` directly into a grid search. Simple.

This is what I mean by composability: the ability to snap these different pieces together to build a robust, repeatable process. This makes it very convenient to use.

Darts

`darts` is built for ease of use and rapid prototyping. It is a modern library for forecasting and anomaly detection that exposes a simple `fit()`/`predict()`

interface across a wide model catalog, from classical ARIMA to deep learning networks like N-BEATS and Transformers.

The library's central data structure is the TimeSeries object. It is the standard input for every darts function, which is what makes utilities like `historical_forecasts` (a backtest in a single call) possible.

For professional work, darts excels with its native support for covariates (like promotions), its ability to train global models on thousands of series at once, and its strong focus on probabilistic forecasting. It can generate uncertainty intervals by producing multiple sample forecasts or through quantile regression.

We load a TimeSeries, which is darts' central data container. We then split it into training and validation sets, fit a model using the simple `fit()` API, and generate a forecast with `predict()`. This example emphasizes the library's user-friendly and consistent interface, which makes rapid prototyping easy.

```
from darts.datasets import AirPassengersDataset
from darts.models import ExponentialSmoothing
from darts import TimeSeries
from darts.metrics import mape
# 1. Load data into a TimeSeries object, darts' standard container
series: TimeSeries = AirPassengersDataset().load()
# 2. Split the data, holding out the last 36 months for validation
train, val = series[:-36], series[-36:]
# 3. Create and train the model using the unified API
# The code remains the same even if you swap
# ExponentialSmoothing with a deep learning model
model = ExponentialSmoothing()
model.fit(train)
# 4. Predict the next len(val) timesteps
prediction = model.predict(len(val))
# 5. Inspect the forecast and evaluate its performance
print("Prediction (first 5 values):")
print(prediction[:5])
print(f"MAPE on validation set: {mape(val, prediction):.2f}%")
```

`darts` is built around its TimeSeries object. You can swap out `ExponentialSmoothing` for a `RegressionModel`, wrapping any scikit-learn regressor, or for a deep learning model like N-BEATS, and the `fit()/predict()` code stays the same. It's my go-to tool for rapid prototyping for this reason. It lets me test a wide range of models, from simple to state-of-the-art, without rewriting everything.

You can swap in a `LightGBMModel` by changing two lines in the code above:

```
from darts.models import LightGBMModel
model = LightGBMModel(lags=12)
```

The remaining code requires no modification. The library interface ensures that the fit and predict methods operate identically across different model families. You do not require a manual reduction transformation in darts like in sktime.

The Nixtla ecosystem

This group of libraries, developed by Nixtla, is built for one thing: speed and scale. They are designed to forecast thousands or even millions of time series at once. They achieve this performance by rewriting core algorithms in optimized code using `numba`, a **just-in-time** (**JIT**) compiler, and by integrating with distributed computing frameworks like Dask and Spark.

- `mlforecast` is the workhorse for executing our five-step workflow at scale. It is a highly efficient, feature-engineering-first library. It takes the long-format DataFrame, creates lags, rolling means, and other features at high speed using optimized code, and then trains a `scikit-learn`-compatible model like LightGBM.
- `statsforecast` is the companion to `mlforecast`. This library provides highly optimized implementations of classical statistical models like ARIMA and ETS. It can fit these models to thousands of series in seconds, making it perfect for generating strong, scalable baselines.

Let's demonstrate `mlforecast's` specialized, high-performance workflow. We start with data in the industry-standard *long* format (`unique_id`, `ds`, `y`) and show how to declaratively create time series features like lags directly within the model object. This highlights its feature-engineering-first approach, which is optimized for speed and scalability.

```
import pandas as pd
import numpy as np
from mlforecast import MLForecast
from mlforecast.lag_transforms import RollingMean
from mlforecast.utils import generate_daily_series
import lightgbm as lgb
# 1. Create data in the "long" format that mlforecast expects
# This structure is designed for forecasting many series at once
series_df = generate_daily_series(
    n_series=10,
    min_length=100,
    max_length=150
)
# Inspect the long format: one row per (series, timestamp)
# with unique_id, ds, y
print(series_df.head())
# 2. Hold out the last 14 days of every series for validation
h = 14
train_df = series_df.groupby(
     "unique_id", group_keys=False, observed=True
).apply(lambda x: x.iloc[:-h])
val_df = series_df.groupby(
    "unique_id", group_keys=False, observed=True
).apply(lambda x: x.iloc[-h:])
# 3. Define the LightGBM model
model = lgb.LGBMRegressor(
    random_state=42, n_jobs=1, num_leaves=64,
    min_child_samples=5, learning_rate=0.05,
    verbose=-1, force_row_wise=True,
)
# 4. Instantiate the forecaster.
# mlforecast creates lag, rolling, and
# calendar features at high speed using numba.
fcst = MLForecast(
    models = {'lgbm': model},
    freq='D',
    lags = [1, 7, 14],
    lag_transforms={1: [RollingMean(window_size=7, min_samples=1)]},
```

```
    date_features = ['dayofweek', 'month'],
    num_threads=1,
)
# 5. Fit on the training data and forecast the validation horizon
fcst.fit(train_df)
y_pred = fcst.predict(h=h)
# 6. Compute MAPE per series and overall
merged = val_df.merge(
    y_pred, on=['unique_id', 'ds']).rename(columns={'y': 'y_true'})
merged['ape'] = np.abs(
    merged['y_true'] - merged['lgbm']) / merged['y_true']
print("Forecasts (first 5 rows):")
print(y_pred.head())
print(f"\nOverall MAPE: {merged['ape'].mean() * 100:.2f}%")
```

`mlforecast` is all about speed and scale. It's built to handle thousands of time series without breaking a sweat, using the standard long-format DataFrame you see here.

The key is how we declare features. We pass a list of instructions (lags, rolling means, date parts) directly to the model. `mlforecast's` `numba`-powered engine does the heavy lifting. This is the exact workflow we used at Chelsea AI Ventures for a retail client with over 10,000 products.

AutoML platforms

AutoML tools are designed to automatically find the best model for your data with minimal intervention.

`autogluon` from Amazon delivers state-of-the-art forecasting with minimal effort. Its philosophy is distinctive: it prioritizes a weighted ensemble of diverse models over exhaustive hyperparameter tuning of any single one. You point it at your data and it trains a portfolio (classical ETS through deep learning) and combines the best performers. That makes it a fast way to set a strong benchmark in a few lines of code.

Here's an example with `autogluon`, where we showcase `autogluon`'s power to deliver state-of-the-art results with minimal code. We convert a standard

DataFrame into a `TimeSeriesDataFrame`, instantiate a `TimeSeriesPredictor`, and call `.fit()`. The magic is what happens next: calling `.leaderboard()` reveals that `autogluon` automatically trained and ensembled a diverse portfolio of models to find the best performer.

```
from autogluon.timeseries import TimeSeriesDataFrame, TimeSeriesPredictor
import pandas as pd
import numpy as np
# 1. Create sample data in the expected format
# (item_id, timestamp, target)
# In a real scenario, this would be your sales, demand, or metrics data
ids = ["A", "B", "C"]
timestamps = pd.to_datetime(pd.date_range("2023-01-01", periods=100))
data = []
for item_id in ids:
    target = np.random.randn(100).cumsum()
    data.append(pd.DataFrame(
        {"item_id": item_id, "timestamp": timestamps, "target": target}))
df = pd.concat(data)
# 2. Convert to AutoGluon's specialized data format
train_data = TimeSeriesDataFrame.from_data_frame(
    df,
    id_column="item_id",
    timestamp_column="timestamp"
)
# 3. Fit a predictor with just a few lines of code
# Specify the forecast horizon and the name of your target column
predictor = TimeSeriesPredictor(
    prediction_length=24,
    target="target",
    eval_metric="MASE",
    verbosity=2,
)
# Two ways to control the search space:
# (a) the presets argument ("fast_training", "medium_quality",
#     "high_quality", "best_quality") trades training time for accuracy.
# (b) the hyperparameters argument lets you pin an explicit shortlist.
# Here we pass an explicit, lightweight portfolio so example runs quickly.
predictor.fit(
    train_data,
    hyperparameters={"Naive": {}, "SeasonalNaive": {}},
    num_val_windows=1,
    refit_every_n_windows=None,
)
```

```
predictor.fit(train_data, presets="medium_quality")
# 4. View the leaderboard to see the performance of all trained models
print("Model performance leaderboard:")
predictor.leaderboard()
```

The magic is in the `.leaderboard()` output. `autogluon` doesn't train one model; it runs a competition. It pits a whole portfolio of models, from simple ETS to deep neural networks, against each other and ensembles the winners.

`autogluon` helps establish a strong performance benchmark and answers the critical question: *How well can any model possibly do here?*

Direct scikit-learn integration

This category of library has one primary goal: to make using scikit-learn regressors for forecasting as simple and direct as possible.

`skforecast` is dedicated exclusively to this task. It provides the tools to turn a time series into a supervised learning problem, train any scikit-learn-compatible regressor, and `backtest`, all with a small, clear API. A natural fit if you are already comfortable in the scikit-learn ecosystem.

Let's see how `skforecast` provides a direct bridge to the familiar scikit-learn world. We explicitly create a scikit-learn regressor, then pass it to a `ForecasterRecursive`, telling it exactly how many lags to create. This example highlights its straightforward, non-abstracted approach.

```
from skforecast.recursive import ForecasterRecursive
from sklearn.ensemble import RandomForestRegressor
from sktime.datasets import load_airline
import pandas as pd
# 1. Load data as a standard pandas Series
y = load_airline()
y.index = pd.to_datetime(y.index.to_timestamp()) # Ensure datetime index
# 2. Create the scikit-learn model you want to use
regressor = RandomForestRegressor(random_state=42)
# 3. Create the forecaster, passing the estimator and specifying lags
forecaster = ForecasterRecursive(
    estimator=regressor,
```

```
    lags=15 # Create 15 lag features
)
# 4. Fit the forecaster on the training data
forecaster.fit(y=y[:-36])
# 5. Predict the next 36 steps
predictions = forecaster.predict(steps=36)
print("Forecasts (first 5 values):")
print(predictions.head())
```

The workflow in `skforecast` is very explicit and should feel natural to anyone who uses scikit-learn. You create your regressor, you define your features by setting the number of lags, and you train the model. It's a direct and powerful way to apply your existing machine learning skills to forecasting tasks without learning a new framework.

Choosing your tool: a decision framework

After exploring the philosophies and code examples of these libraries, we can create a mental map to help choose the right one for a given task. The following diagram summarizes the decision process based on your primary goal, whether it's research-grade rigor, rapid prototyping, production scale, full automation, or direct integration with scikit-learn.

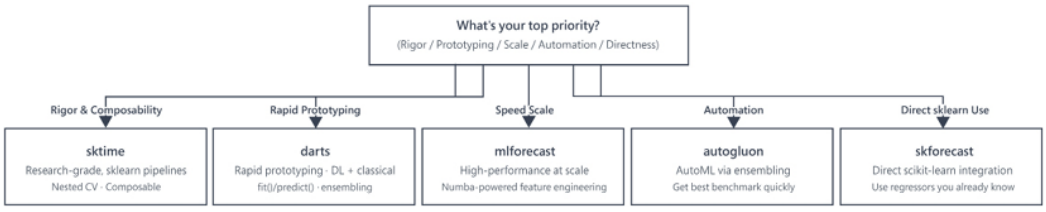


Figure 4.2: A decision framework for selecting a forecasting library

This guide organizes the key libraries by their primary philosophy. Your choice depends on your top priority: `sktime` for research-grade rigor, darts for rapid prototyping, `mlforecast` for performance at scale, `autogluon` for automated benchmarking, or `skforecast` for direct scikit-learn integration.

With this map of the ecosystem, personally, my approach isn't to pick just one tool, but to use these libraries strategically in a sequence:

1. **Baseline with autogluon**: My first step is often to run the data through `autogluon`. In a few hours, I get a benchmark from an ensemble of models. This tells me what level of accuracy is achievable.
2. **Build a custom model with skforecast**: After establishing a baseline, this is the next logical step for building customized models applying your scikit-learn knowledge. It's perfect for creating and tuning a specific regressor for your forecasting problem without the overhead of a larger framework.
3. **Scale for Production with mlforecast**: For production systems, especially with many time series, I use `mlforecast`. Its speed and efficient feature engineering are exactly what's needed to build custom, scalable models with LightGBM or XGBoost.
4. **Experiment with sktime or darts**: If I need to do complex, custom validation or experiment with novel deep learning architectures, I'll turn to the unified frameworks. `sktime` is my choice for research-grade pipeline construction, while darts is great for quickly comparing a wide variety of models. This strategic approach gives you a clear path forward. You now have a framework for choosing your tools and a professional workflow for using them.

The final step is to put it all together. In the next section, we will take on our first **Benchmark Challenge**: we will use this new workflow to build a simple machine learning forecast and see if we can beat the classical `AutoARIMA` model from *Chapter 3*.

Onto our final section: let's create an ML forecast from scratch!

Creating an ML forecast

We have a new paradigm, a reliable workflow, and a map of the modern tools. The final step is to put it all into practice. This section is where the theory becomes tangible.

We will take on our first **Benchmark Challenge**. Can a simple machine learning model, using the regression approach, beat the performance of the classical AutoARIMA model we built in *Chapter 3*? In any professional setting, you must be able to prove that a new, more complex model provides a meaningful improvement over a simpler baseline. This example gives you the blueprint.

Our goal is to build a scikit-learn model to forecast the same Google Trends data from *Chapter 3*. We will follow our 5-step blueprint precisely.

1. **Data preparation**: Load the data and set the performance baseline
2. **Feature engineering**: Create features from the time series, including an external variable (a covariate)
3. **Model training**: Train a `RandomForestRegressor` on our features
4. **Prediction**: Generate out-of-sample forecasts
5. **Validation**: Evaluate the model using a rigorous backtest with multiple performance metrics and visualizations

Let's look at these steps in detail!

Step 1: setup and baseline

First, we'll set up our environment by importing the necessary libraries and loading the data. Most importantly, we'll state our performance target: the **Mean Absolute Scaled Error** (**MASE**) score achieved by the AutoARIMA model in the previous chapter. This number is our benchmark to beat.

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error
```

```
from sktime.split import SlidingWindowSplitter
import matplotlib.pyplot as plt
# For reproducibility, we'll create a sample DataFrame,
# same as for Chapter 3
dates = pd.date_range('2019-01-01', '2023-12-31', freq='D')
np.random.seed(42)
trend = np.linspace(50, 80, len(dates))
seasonal = 10 * np.sin(2 * np.pi * np.arange(len(dates)) / 365.25)
# A clear weekend uplift,
# so the is_weekend covariate later carries real signal
weekend_effect = np.where(dates.dayofweek >= 5, 7, 0)
noise = np.random.normal(0, 5, len(dates))
y = pd.Series(trend + seasonal + noise, index=dates, name="value")
# The baseline score to beat from Chapter 3
ARIMA_MASE = 0.85 # Replace with the actual score from your Ch3 results
print(f"Target to beat: AutoARIMA MASE = {ARIMA_MASE:.2f}")
```

This synthetic data extends the *Chapter 3* series with a linear trend, a yearly seasonal component, and a deliberate weekend uplift. The weekend boost gives the `is_weekend` covariate we will engineer in the next step a real pattern to learn, rather than asking the model to fit noise. We've loaded our data into a pandas Series y and defined our target MASE score. We now have a clear, quantitative goal. Any score lower than 0.85 represents an improvement, justifying the use of a more complex machine learning model.

We chose MASE as our primary benchmark because it is scale-independent, allowing for fair comparisons across different time series. MAE is easier to interpret, but MASE gives a more reliable relative measure.

Step 2: manual feature engineering

Next, we will transform our time series into a feature matrix (`X`) and a target vector (`y`). As discussed in the previous section, this is the key to using machine learning models. We will create two features. The first is `lag_1`, the value from the previous day, to capture short-term momentum. The second is `is_weekend`, a binary flag (1 for weekend, 0 for weekday). `is_weekend` is a

covariate, an external variable not derived from the time series itself, that we believe carries predictive signal.

```
def create_features(y):
    """Creates time series features from a pandas Series."""
    X = pd.DataFrame(index=y.index)
    X['day_of_week'] = y.index.dayofweek
    X['is_weekend'] = (X['day_of_week'] >= 5).astype(int)
    X['lag_1'] = y.shift(1)
    # Drop the day_of_week column as it's redundant now
    X = X.drop(columns=['day_of_week'])
    return X
# Create the feature matrix
X = create_features(y)
# Align X and y by dropping the first row that has a NaN value
Xy = pd.concat([y, X], axis=1).dropna()
y = Xy['value']
X = Xy.drop(columns=['value'])
print("Feature Matrix X (first 5 rows):")
print(X.head())
```

The `create_features` function takes the original series y as input. It uses the pandas `DatetimeIndex` to get the `dayofweek` and creates our `is_weekend` covariate. It then uses the `.shift(1)` method to create the lag feature. Finally, we combined X and y and dropped the first row, which contains a `NaN` from the shift operation, to ensure our features and target are perfectly aligned.

As discussed, with this transformation, we have reframed the problem. Instead of predicting a sequence, our `RandomForestRegressor` will now learn to predict the value based on the `is_weekend` indicator and the value from the previous day.

We'll cover feature engineering in much more detail in *Chapter 5*.

Step 3: training the model

With our features ready, we can train a model. We won't train it just once, though. To get a reliable estimate of its performance, we'll use a backtesting

strategy. We'll define our `RandomForestRegressor` and use the `SlidingWindowSplitter` that we've seen in *Chapter 3*, imported from `sktime`, to create multiple historical training and testing periods.

```
# Define our model
model = RandomForestRegressor(n_estimators=100, random_state=42)
# Define our temporal cross-validation strategy
cv = SlidingWindowSplitter(window_length=50, step_length=12, fh=1)
```

`SlidingWindowSplitter` takes `window_length`, `step_length`, and `fh`; the number of splits is determined by the data and these parameters.

We've defined the model we want to train and the cross-validation object that will guide the training process. The splitter slides a 50-day training window forward in 12-day steps, producing many non-overlapping one-step-ahead evaluations across the series.

We set `window_length=50` to give the model enough recent history while keeping each fit cheap, and `step_length=12` to advance the window roughly twice a month. The number of folds falls out of these choices and the length of the series.



Direct vs recursive multi-step strategies

Here `fh=1` predicts one step ahead, so the question of multi-step `strategy` does not arise. Once you forecast a horizon longer than 1, you have two choices. *Recursive* forecasting trains a single one-step model and feeds its own predictions back as lag inputs to roll forward; it is simple but accumulates errors. *Direct* forecasting trains a separate model per horizon (one for `t+1`, one for `t+2`, ...); it avoids feedback errors at the cost of training many models. `sktime` exposes both via the `strategy` argument of `make_reduction`, and we will use this lever in *Chapter 5*.

We are using a `RandomForestRegressor` as our model. We start with the default parameters, but in a real-world project, we would tune hyperparameters like `max_depth` to optimize performance.

A gotcha you should know about tree models and trends. Random forests cannot extrapolate. A tree predicts the average target value of its leaf, so any forecast is bounded by the targets it saw during training. Feed a series with a strong upward trend straight into a Random Forest and it will systematically under-predict the future once values move past the training range. Our example only looks like it tracks the trend because the `SlidingWindowSplitter` retrains the model on a recent window at every step (`fh=1`), so the training range keeps moving up with the data. The model is *riding* the trend, not extrapolating it. In *Chapter 5* we will fix this properly by differencing the series or by using detrended targets before fitting tree models.



Author's Insight

In *Chapter 3*, we learned the theory behind temporal cross-validation. In practice, use a library that has it built-in. `sktime` provides splitters like the one above; darts goes even further with `historical_forecasts`, which runs the whole backtesting loop in a single call. This prevents bugs and makes doing the right thing the easy thing.

We have now defined our model and validation strategy. The `RandomForestRegressor` is a non-linear model, and the splitter ensures it is tested under realistic conditions.

Step 4: prediction

Now we execute the backtesting loop. In each iteration, we train the model on a *past* data split and generate predictions for the corresponding *future*

split. We'll store these out-of-sample predictions and then visualize them against the true values to get a qualitative sense of the model's performance.

```
# Store out-of-sample predictions from each fold
all_predictions = []
all_true_values = []
all_test_indices = []
for train_index, test_index in cv.split(y):
    # Get the data for this split
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
    # Train the model
    model.fit(X_train, y_train)
    # Predict on the test set
    preds = model.predict(X_test)
    # Store results
    all_predictions.extend(preds)
    all_true_values.extend(y_test.values)
    all_test_indices.extend(test_index)
# Create a DataFrame for plotting results
results_df = pd.DataFrame({
    'true_values': all_true_values,
    'predictions': all_predictions
}, index=y.index[all_test_indices])
```

A plot makes the model's behavior legible. Let's visualize the backtest predictions against the actual values to see where the model performs well and where it struggles.

```
# Plot the backtest predictions against the original data
plt.figure(figsize=(12, 6))
plt.plot(y, label='Original Data')
plt.plot(results_df['predictions'],
         label='Backtest Predictions', linestyle='--')
plt.title('Google Trends: Backtest Forecast vs. Actuals')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()
```

This is the figure we should see:

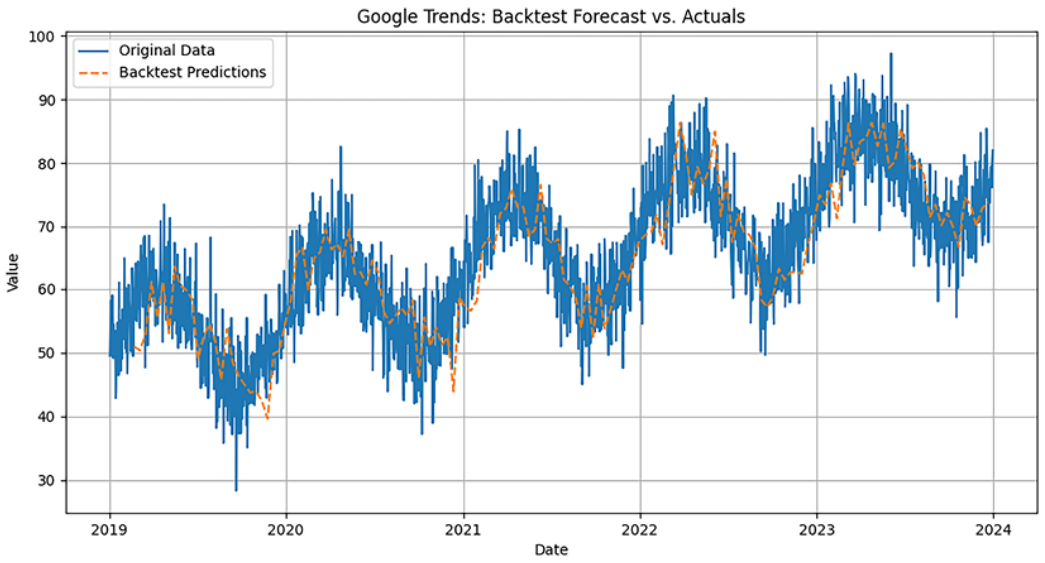


Figure 4.3: The ML forecast of the artificial Google Trend data

The for loop automates the train-predict-validate cycle. For each split generated by `cv.split(y)`, we train the model on the `_train` data and generate predictions on the `_test` data. These predictions are `out-of-sample` because the model never saw the test data during that iteration's training. We then create a matplotlib plot to visually inspect how well our predictions tracked the actual values.



Author's Insight

While we're using matplotlib for a custom plot, most major frameworks have built-in helpers. For instance, darts `TimeSeries` objects have a `.plot()` method, making it easy to overlay forecasts on actuals. This reinforces the idea that good libraries provide convenient, reliable utilities for common tasks.

Step 5: validation

This is the moment of truth. We'll use the out-of-sample predictions generated in Step 4 to calculate our final performance metrics. We'll use MAE, RMSE, and MASE to compare our ML model directly against the AutoARIMA baseline.

```
# Calculate the metrics
mae = mean_absolute_error(all_true_values, all_predictions)
rmse = np.sqrt(mean_squared_error(all_true_values, all_predictions))
# For MASE, the naive forecast is the previous value.
# Compute the naive scale on the training portion only (everything before
# first test index) to keep test-set information out of the denominator.
first_test_idx = min(all_test_indices)
y_train_for_naive = y.iloc[:first_test_idx]
naive_mae = mean_absolute_error(
    y_train_for_naive.iloc[1:],
    y_train_for_naive.iloc[:-1]
)
mase = mae / naive_mae
print("--- ML Model Performance ---")
print(f"MAE:  {mae:.2f}")
print(f"RMSE: {rmse:.2f}")
print(f"MASE: {mase:.2f}")
print("\n--- Baseline Performance ---")
print(f"AutoARIMA MASE: {ARIMA_MASE:.2f}")
if mase < ARIMA_MASE:
    print("\nResult: The Random Forest model beat the ARIMA baseline!")
else:
    print("\nResult: The ARIMA baseline was better or equal.")
```

I am seeing this performance:

```
--- ML Model Performance —
MAE:  5.06
RMSE: 6.11
MASE: 0.90
```

We used standard functions from scikit-learn to calculate MAE and RMSE. For MASE, we calculated the naive forecast error on the training portion only, so the denominator does not leak information from the test set and used it to scale our model's MAE. This is a common approximation; the

strictly correct method is to compute the naive error within each training fold of the cross-validation and average across folds, which the dedicated MASE implementations in `statsforecast`, `mlforecast`, and `darts` will do for you. The final printout gives us a clear, quantitative answer to our benchmark challenge.

In this instance, the simple ML model with only two features couldn't beat the optimized AutoARIMA. I think this is an important lesson for any project: more complex doesn't automatically win. It often takes careful benchmarking to make informed decisions about iterative improvements.

AutoARIMA models trend and yearly seasonality explicitly, while a Random Forest on `lag_1` and `is_weekend` has only a one-step memory and cannot extrapolate the trend. This underscores the importance of feature engineering, which we will dive into deeply in the next chapter. Richer lags, rolling means, calendar effects, and detrending tricks are exactly what let tree models compete on data like this.

Summary

We have now made the conceptual leap from classical statistical methods to the modern machine learning paradigm for forecasting. By reframing our problem as a supervised regression task, we can apply a workflow for building predictive models. You are now equipped with a systematic process for transforming time series data into features.

Choosing the right library is about matching the tool to the job. For reproducible, multi-step workflows, `sktime`'s scikit-learn compatibility makes pipelines easy to compose. For comparing a wide range of models from classical to deep learning, darts provides a clean, intuitive API. When you need to forecast thousands of series quickly, the `Nixtla` ecosystem (`mlforecast` plus `statsforecast`) handles the scale. To get a strong baseline

with little code, `autogluon` trains a diverse portfolio and ensembles the winners.

In the coming chapter, we'll build on this foundation. This approach comes with the responsibility to make tooling choices that mitigate risk and to prove the value of a new machine learning model by testing it against the classical baselines we established in *Chapter 3*. This provides a complete, professional answer when stakeholders ask, "*How do we know this new model is actually better?*"

In the coming chapters, we will build directly on this foundation to test and master more advanced techniques, including:

- Large-scale feature engineering with tree-based models like XGBoost and LightGBM
- Deep learning architectures such as LSTMs and Transformers
- Uncertainty quantification methods like conformal prediction

The manual forecast we built here is the first step. The next step is to automate and scale this process, so models can learn from hundreds of features rather than two.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers.

https://discord.gg/VXpz9YPUTW



5

Feature Engineering and Tree-Based Models

In *Chapter 4*, we surveyed the Python time series libraries and the case for treating forecasting as a regression problem rather than a statistical task. Now, we get our hands dirty getting the most out of these models with feature engineering.

In my experience building prediction and forecasting systems, the biggest performance gains rarely came from a fancier algorithm. They came from better understanding of the problem and better features (and a better understanding of the problem often results in better feature engineering). A simple logistic regression model fed with clever, well-engineered features consistently outperforms a complex neural network working with raw data. It's a lesson that has stuck with me: **feature engineering is where the gains live.**

The core idea is simple. We will take a raw time series, a sequence of values over time, and transform it. We will slice it, dice it, and enrich it until we have a flat table, a feature matrix X and a target vector y, just like in any other machine learning task. This transformation is how we solve a fundamental challenge: standard machine learning models assume data points are independent and identically distributed (i.i.d.), an assumption that time series data clearly violates. By creating features that explicitly encode the temporal dependencies (like the value from seven days ago), we make the data suitable for these regressors.

This approach gives us two superpowers. First, **scalability**. We can build one model to predict sales for thousands of different stores simultaneously. Second, **richness**. We can incorporate any external information we want, such as promotions, holidays, or competitor prices, as just another column in our feature matrix.

This is something classical models like ARIMA struggle with. While extensions like ARIMAX or SARIMAX can incorporate exogenous features, they do so under restrictive linear assumptions and are best suited for small sets of carefully chosen covariates. In contrast, feature-based ML models can flexibly scale to many covariates, interactions, and nonlinearities.

In this chapter:

- Feature engineering
- Advanced feature engineering
- Choosing a tree model

- Hybrid modeling and decomposition
- Automated feature engineering

This chapter walks through that workflow end-to-end. We'll start with a real-world business problem, engineer a powerful set of features, and train a gradient-boosted tree model to generate accurate forecasts. Let's begin.

Technical requirements

In this chapter we will work with the M5 Forecasting Accuracy dataset, which contains daily unit sales for 3,049 Walmart products across 10 stores in California, Texas, and Wisconsin (about 30,490 product-store time series in total). The data is hosted on Zenodo at https://zenodo.org/records/12636070 under a Creative Commons Attribution 4.0 license, which means we can redistribute and modify it freely as long as we cite the source.

We picked M5 deliberately. Other widely used retail benchmarks (Rossmann, Favorita, Walmart Recruiting) live behind Kaggle competition rules that restrict redistribution, so we'd have to send you to a login page to fetch them. M5 lets us ship a one-line download script in the companion repository (`download_m5.py`) that pulls the files straight from Zenodo. Run it once before working through the notebooks.

Feature engineering

For time series in particular, the quality of the features is what makes or breaks a model. Feature engineering turns raw data into the informative signals the model needs. This section covers the first layer of that work: preparing the data and creating the first features.

Our laboratory for this chapter is the M5 Forecasting Accuracy dataset, the data behind the influential M5 competition. It captures real daily unit sales for thousands of Walmart products across ten stores. Like most business data, it contains missing values (products not yet on sale at a given store), categorical hierarchies, calendar events, and price changes, so it makes for a realistic challenge.

Let's frame our mission. You are a forecasting analyst at a large retailer working from M5-style data: 30,490 product-store time series spanning about five years, plus a calendar of national events and SNAP food-benefit days, plus weekly sell prices per item. Your task is to produce daily unit-sales forecasts for every product-store pair, six weeks ahead. The forecasts feed inventory planning, staffing, and replenishment decisions; better forecasts mean fewer stockouts and less waste.

This dataset makes for a strong case study because it represents a multi-faceted time series forecasting problem. Sales are influenced by a variety of documented factors: named events

(SuperBowl, Easter, Christmas), state-conditioned SNAP days when food-benefit recipients can spend, weekly price changes, plus calendar seasonality. In reality, though, retail sales are shaped by an even wider set of forces from weather and local events to macroeconomic shifts like recessions, demographic differences, and marketing campaigns. Not all of these are captured here, but this limitation itself is instructive: in applied forecasting, the challenge is rarely just building a model, but working with the features you do have, while recognizing what's missing.

To tackle this, we'll work with three files from M5:

- `sales_train_validation.csv`: A wide-format table with 30,490 rows (one per item-store series) and 1,913 daily-sales columns labeled `d_1` to `d_1913`. Static columns (`id`, `item_id`, `dept_id`, `cat_id`, `store_id`, `state_id`) identify each series.
- `calendar.csv`: One row per day, with date, year, month, weekday, named events (`event_name_1/2`, `event_type_1/2`), and the three SNAP flags (`snap_CA`, `snap_TX`, `snap_WI`).
- `sell_prices.csv`: Weekly sell price per (`store_id`, `item_id`, `wm_yr_wk`).

The fundamental task is to predict the daily sales column. In the regression framework, the problem is defined as `Sales = F(Features)`, where features are derived from all available columns across the three files.

Our pipeline that we'll work on throughout this chapter begins by reshaping data into a panel and dropping initial records to ensure accuracy. It then constructs a matrix through extracting signals, flagging events, and calculating lags. We'll apply a logarithmic transformation (using `np.log1p`) to stabilize variance before training the model. Finally, we'll produce forecasts by converting predictions to units (`np.expm1`) and feeding them forward for subsequent steps

The first step is to create a unified, clean dataset that serves as the canvas for feature engineering. The wide sales table needs to be reshaped to long format, then merged with the calendar and price tables. To keep the example laptop-runnable, we sample 100 random items (about 1,000 series after combining with the 10 stores). Bump the sample size to scale up.

```
# Load and reshape the M5 data
import pandas as pd
import numpy as np
calendar = pd.read_csv("calendar.csv", parse_dates=["date"])
sales_wide = pd.read_csv("sales_train_validation.csv")
prices = pd.read_csv("sell_prices.csv")
# Sample 100 items (~1,000 series) so the example runs on a laptop.
rng = np.random.RandomState(42)
sample_items = rng.choice(
    sales_wide["item_id"].unique(), size=100, replace=False)
sales_wide = sales_wide[sales_wide["item_id"].isin(sample_items)].copy()
```

```
# Melt the wide format (one column per day) into a long panel.
id_vars = ["id", "item_id", "dept_id", "cat_id", "store_id", "state_id"]
day_cols = [c for c in sales_wide.columns if c.startswith("d_")]
sales_long = sales_wide.melt(
    id_vars=id_vars,
    value_vars=day_cols,
    var_name="d",
    value_name="sales",
)
# Merge calendar metadata, then weekly prices.
df = sales_long.merge(calendar, on="d", how="left")
df = df.rename(columns={"date": "Date"})
df = df.merge(prices, on=["store_id", "item_id", "wm_yr_wk"], how="left")
print("Merged DataFrame shape:", df.shape)
df.head()
```

With the data merged, the next step is to *handle the gaps*. In M5, a *missing* `sell_price` means the product was not yet on sale at that store on that day, often because the item launched later. Filling those NaNs with zero or a global median would invent demand the model would then learn to predict, so we drop them.

```
# Drop pre-launch rows: NaN sell_price means the product was not on sale yet.
pre_launch = df['sell_price'].isna().sum()
df = df.dropna(subset=['sell_price']).reset_index(drop=True)
print(f"Dropped {pre_launch:,} pre-launch rows; remaining shape: {df.shape}")
```

Notice what we don't do here: we don't impute competitor distances, store-type flags, or promotion-history columns. M5 carries those as concrete static columns (`cat_id`, `dept_id`, `store_id`, `state_id`) right next to the sales values, so there is nothing to fill. The dataset that taught us the most about retail forecasting, the Rossmann competition data, has a much harder cleanup story; with M5 the cleaning step is much simpler. The resulting DataFrame is now a tidy panel ready for feature creation.

After cleaning, we're left with a **panel dataset**: many product-store series observed across many dates, each with daily sales and a set of explanatory features.

- **Calendar and events**: named events (`event_name_1/2`, `event_type_1/2`), SNAP food-benefit days (`snap_CA`, `snap_TX`, `snap_WI`).)
- **Item attributes**: `cat_id` (`HOBBIES`/`HOUSEHOLD`/`FOODS`), `dept_id` (`HOBBIES_1`, `FOODS_3`, …), `item_id`.
- **Store and geography**: `store_id` (`CA_1`, `TX_2`, …), `state_id.`
- **Pricing**: `sell_price` (weekly, per item-store).



These columns combine time-varying signals (events, SNAP days, prices) with time-invariant attributes (item, store, state). The mix makes the dataset realistic, much closer to a real-world forecasting problem than a clean academic example.

This is not a single time series but a panel of many series, one per item-store pair. Instead of learning from just one history, the model sees thousands of parallel ones.

The dataset is a good vehicle for learning the feature engineering workflow. Traditional time series models like ARIMA are not a natural fit here. One could, in principle, fit an individual ARIMA model for each of the 30,490 series, or use ARIMAX to incorporate events as external regressors. But this approach is cumbersome, computationally expensive, and fails to share information across products and stores.

Unlike a classical univariate model, which learns from a single series, a panel approach allows a single model to learn from thousands of histories simultaneously. This enables the model to learn patterns that generalize across all series, such as the impact of a national event or the lift on a SNAP day.

It can also learn nuanced patterns specific to certain segments by using static attributes as features. The model can learn that `FOODS_3` in California responds differently to a price drop than `HOBBIES_1` in Texas. This ability to learn a single, unified function that predicts sales for any series based on its attributes makes the regression approach far more scalable than building and maintaining 30,490 separate forecasting models.

Multivariate methods like **Vector Autoregression** (**VAR**) might seem attractive, since in principle they can model interactions across series. In practice, VAR scales very poorly when the number of series is large. Even a simple VAR with one lag across all 30,490 series would need on the order of a billion parameters; the data can't support that. VAR also assumes a common lag structure across series, which is unrealistic given the heterogeneity of items and stores. Add the zero-heavy, intermittent character of unit-level retail demand, and VAR quickly becomes unwieldy. For these reasons, VAR and its relatives are rarely used for problems of this scale.

Another class of solutions is **hierarchical forecasting**, where we treat M5's sales as a hierarchy (from total to state to store to category to item). Methods like Bottom-Up, Top-Down, and Middle-Out allow us to aggregate or disaggregate forecasts across levels. We'll return to these approaches in *Chapter 6*, where we explore hierarchical and multivariate forecasting in depth.

For now, we'll adopt a machine learning approach. We treat each row, a particular item-store-date triple, as an independent training example. By engineering a set of features for each row, we can train a single model to capture relationships among events, SNAP days, prices, calendar effects, and product or store attributes. This strategy scales naturally to thousands of series and shares patterns across the chain.

Calendar features

The timestamp itself is the first place to look for features. Decomposing the `Date` column into year, month, day, day of week, and so on gives the model explicit handles for seasonality and trend.

The `pandas` library makes this process straightforward. Its `dt` accessor for datetime columns allows for the direct extraction of various time-based attributes.

```
# Code Block 5.2: Creating Calendar Features
df['Year'] = df['Date'].dt.year
df['Month'] = df['Date'].dt.month
df['Day'] = df['Date'].dt.day
df['WeekOfYear'] = df['Date'].dt.isocalendar().week.astype(int)
df['DayOfWeek'] = df['Date'].dt.dayofweek # Monday=0, Sunday=6
df['DayOfYear'] = df['Date'].dt.dayofyear
```

The value of these features is immediate. Year provides a direct signal for any long-term trends in sales. Month, `WeekOfYear`, and `DayOfWeek` allow the model to learn seasonal patterns at different frequencies. For example, the model can learn that sales are typically higher in December (`Month == 12`) or lower on Sundays (`DayOfWeek == 6`), without needing to infer these patterns from the raw timestamp.

A pleasant side effect of M5 is that `calendar.csv` already carries year, month, wday, weekday, and so on, so we could skip half of the block above. Recomputing from Date keeps column names consistent with the chapter's conventions. If you want to wire up holidays from outside the dataset, libraries like `workalendar` provide built-in calendars for many countries and regions; for M5 we get the events for free.

Event and holiday features

Beyond regular calendar cycles, sales are often impacted by specific events. M5 captures these in two ways. The `event_name_1` (and the rarer `event_name_2`) column lists named events on each day, things like `SuperBowl`, `ValentinesDay`, `Easter`, `Christmas`. `event_type_1` groups them into `Sporting`, `Cultural`, `National`, `Religious`. The `snap_CA`, `snap_TX`, `snap_WI` columns flag SNAP food-benefit redemption days per state, days when SNAP recipients can spend benefits, which producesproduce large demand spikes for `FOODS` items in the affected state.

A simple, useful first pass is a binary `IsEvent` flag. We then resolve the right `snap_*` column for each row based on the row's `state_id`, so the model sees a single `snap_active` column instead of three columns it has to combine itself.

```
# Binary flag: any named event on this day.
df['IsEvent'] = df['event_name_1'].notna().astype(int)
```

```
# Resolve the state-specific SNAP flag into a single column.
df['snap_active'] = np.select(
    [df['state_id'] == 'CA', df['state_id'] == 'TX',
     df['state_id'] == 'WI'],
    [df['snap_CA'],
    df['snap_TX'], df['snap_WI']],
    default=0,
).astype(int)
```

If you wanted the full event signal, you could one-hot encode `event_type_1` (four categories) instead of collapsing it to a single binary. We keep the simpler flag here because the more useful signal in M5 lives in SNAP days and price changes, both of which we'll engineer below.

Feature creation is more than mechanical column-shuffling. It's a structured way of embedding domain knowledge and testing hypotheses about the drivers of sales: "`FOODS spike on SNAP days`", "`non-food categories dip on SuperBowl Sunday`", "`demand falls when prices rise`". Each engineered feature is a quantitative version of a hypothesis, and the model's task is to learn the effect size.

Advanced feature engineering

While calendar features capture fixed, repeating patterns, the dynamics of a time series also depend on its own recent history. Advanced feature engineering techniques are required to model concepts like momentum, trend, and volatility. These features capture the temporal dependencies that are the hallmark of time series data.

Lag features

One of the most fundamental properties of many time series is **autocorrelation**: the value at a given time is correlated with values at previous times. In a regression framework, the most direct way to capture this is by creating **lag features**. A lag feature is simply a value from a previous time step. For example, `sales_lag_1` would be the sales from the previous day, and `sales_lag_7` would be the sales from the same day of the previous week.

Creating these features in pandas is done using the `.shift()` method. A critical detail when working with panel data is to compute these lags independently for each time series (in M5, each item-store pair, identified by id). We group by id and then shift so values don't bleed across series.

While we often select lags based on intuition (7 for weekly patterns, 365 for yearly), a more formal approach uses the **Autocorrelation Function** (**ACF**) plot. The ACF shows the correlation of the series with itself at different lags, helping you identify which past values

are most predictive. For an M5 series, an ACF plot would reveal a strong spike at lag 7 (most products sell on the same weekday rhythm), confirming the choice.

```
# Code Block 5.3: Creating Lag Features
# Sort once so shifts move along the calendar within each series.
df = df.sort_values(['id', 'Date']).reset_index(drop=True)
    # Create lag features for sales (1 day, 1 week, 2 weeks, 4 weeks).
    lags = [1, 7, 14, 28]
    for lag in lags:
        df[f'sales_lag_{lag}'] = df.groupby('id')['sales'].shift(lag)
```

Caution: when forecasting multiple steps ahead, lag features introduce a challenge. To predict day *t+2*, your model needs the sales from *t+1*, which you don't have because it's also a forecast. This is the central practical issue with our 6-week (42-day) horizon. There are three common ways to handle it:

- **Recursive forecasting**. Predict *t+1*, write that prediction back as if it were a known value, regenerate `sales_lag_1` and any rolling features, then predict *t+2*, and so on. One model, called *h* times. Errors compound, but the model stays compact and you can reuse it for any horizon.
- **Direct forecasting**. Train a separate model per horizon: one for `sales_t+1`, one for `sales_t+2`, ..., one for `sales_t+42`. Each model uses only features known at prediction time, so there is no error feedback. The cost is *h* models to maintain, and each model sees less signal because it can't lean on the freshest lag.
- **Multi-output**. Use a single model with *h* targets (`sklearn`'s `MultiOutputRegressor` wraps any regressor this way), which sits between the two extremes.

The takeaway is very simple: a plain fit/predict with a `sales_lag_1` feature only gives you a 1-step-ahead forecast. To deliver the 6-week forecast we want, you need one of the three strategies we just mentioned. We'll show the recursive loop in code at the end of this section, and return to the trade-offs in later chapters.

Rolling window statistics

Lag features capture specific past data points, but they can be sensitive to noise. **Rolling window statistics** summarize recent behavior over a sliding window, smoothing out the day-to-day noise. Common examples include the moving average (recent trend) and the moving standard deviation (recent volatility).

For example, a 7-day rolling mean of sales gives the model a smoothed view of the past week's performance, helping it understand the current trend without being thrown off by a single anomalous day. Similar to lag features, these must be calculated on a per-series basis (groupby id, not store, since each item-store pair is its own series).

The most important decision here is the window size. This choice is a trade-off: a small window (3 days) is responsive to recent changes but sensitive to noise but a large window (28 days) provides a smoother, more stable signal of the trend but is slower to react to changes. It's common practice to create features from several window sizes to give the model both short-term and long-term context.

```
# Code Block 5.4: Creating Rolling Features
# Define rolling window sizes (one week, two weeks, four weeks)
windows = [7, 14, 28]
for window in windows:
# Rolling mean per series, shifted by one to avoid leakage.
df[f'sales_rolling_mean_{window}'] = (
    df.groupby('id')['sales']
    .transform(lambda x: x.shift(1).rolling(window=window).mean())
)
# Rolling standard deviation per series.
df[f'sales_rolling_std_{window}'] = (
    df.groupby('id')['sales']
    .transform(lambda x: x.shift(1).rolling(window=window).std())
)
# Drop the warm-up rows where the longest window has no history.
# Do NOT use bfill here: filling NaNs with future values would leak the target
# we are trying to predict back into the training features.
df = df.dropna(subset=[f'sales_rolling_mean_{max(windows)}']).reset_index(drop=True)
```

Note the use of `.shift(1)` before the rolling calculation. It prevents data leakage: the rolling window statistics for day t are calculated using data only up to *t-1*, which is the information that would be available at the time of prediction.

Beyond simple means and standard deviations, you can create context-aware features. A rolling z-score, calculated as `(current_sales - rolling_mean_7) / rolling_std_7`, tells the model how unusual today's sales are relative to the past week. The feature is scale-invariant and often more informative than the raw sales value itself.

Encoding cyclical time

Treating cyclical features like Month (1-12) or DayOfWeek (0-6) as simple integers is problematic. A model sees 12 and 1 as distant numbers, failing to understand that December is immediately followed by January. One-hot encoding is an alternative, but it can create many sparse features and doesn't explicitly encode the cyclical proximity.

A cleaner solution is to map these features onto a circle using sine and cosine transformations. This converts a single cyclical feature into two new features that together preserve the cyclical distance. For example, (`sin(Month)`, `cos(Month)`) will have similar values for December and January, making their relationship clear to the model.

```python
# Sine/Cosine Transformation
def encode_cyclical(df, col, max_val):
    df[col + '_sin'] = np.sin(2 * np.pi * df[col]/max_val)
    df[col + '_cos'] = np.cos(2 * np.pi * df[col]/max_val)
    return df
df = encode_cyclical(df, 'DayOfWeek', 7)
df = encode_cyclical(df, 'Month', 12)
# 'Day' is the day of the month. Using 31 as the period is a fixed approximation;
# February and 30-day months get a slightly distorted angle. For exact per-month
# handling, look up the month length with calendar.monthrange(year, month)[1].
df = encode_cyclical(df, 'Day', 31)
df = encode_cyclical(df, 'WeekOfYear', 52)
```

This transformation works by mapping the integer values onto a two-dimensional circle. For a linear model, this matters most. It ensures the distance between the feature vectors for month 12 and month 1 is small, correctly representing that December is adjacent to January. Tree-based models like LightGBM can often handle raw integer features by finding optimal split points, but cyclical features can still improve performance and lead to simpler, trees.

The following diagram shows a technique to solve the problem of representing cyclical time: by converting a feature like *month* into sine and cosine components, we ensure that values like December (12) and January (1) are close to each other in the feature space.

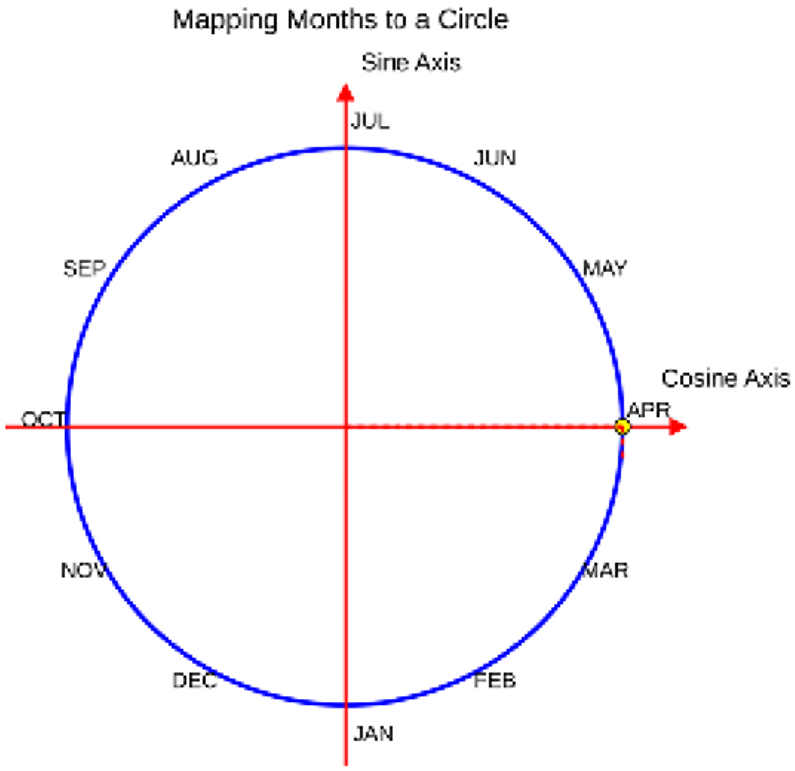


Figure 5.1: Cyclical feature encoding

Domain-specific features

The final and often most impactful layer of feature engineering involves encoding domain-specific business logic. For M5, this means turning weekly prices into a promotion signal.

M5 doesn't ship a `Promo` column the way Rossmann does, but `sell_price` has the same information in a different form. A markdown shows up as today's price falling below its recent rolling baseline. We define a promotion as *today's price is below 95% of the trailing 30-day rolling median*, computed per series with `shift(1)` so today is excluded from the window:

```
# Trailing 30-day rolling median price per series, excluding today.
df['price_30d_median'] = (
    df.groupby('id')['sell_price']
    .transform(lambda x: x.shift(1)
    .rolling(window=30, min_periods=7).median())
)
# Promo flag: today's price is meaningfully below the recent baseline.
df['Promo'] = (
    (df['sell_price'] < 0.95 * df['price_30d_median'])
    & df['price_30d_median'].notna()
).astype(int)
```

A second domain feature is already in our hands: `snap_active`, the SNAP food-benefit redemption flag we built earlier. SNAP days produce sharp, predictable lifts in the FOODS categories in the affected state, especially in the first week of the month when benefits are issued. Because we resolved `snap_active` per row using `state_id`, the model can learn the FOODS-on-SNAP-day interaction without us hard-coding it.

These domain-specific features give the model context that would be hard to learn from the raw columns alone. The table summarizes the engineered feature set, grouped by category, with the reasoning behind each one.

Feature Name	**Category**	**Data Type**	**Rationale/Hypothesis**
Year, Month, Day	Calendar	Numeric	Captures long-term trend and seasonal patterns.
`DayOfWeek`, `WeekOfYear`	Calendar	Numeric	Captures weekly and yearly seasonality.
`IsEvent`	Event	Binary	Flags any named event (Super Bowl, Easter, ...) on this day.

`snap_active`	Event	Binary	Flags state-specific SNAP food benefit days.
`sales_lag_1`, `_lag_7`	Lag	Numeric	Captures short-term and weekly autocorrelation in sales.
`sales_rolling_mean_7`	Rolling Window	Numeric	Models the recent sales trend, smoothing out daily noise.
`sales_rolling_std_7`	Rolling Window	Numeric	Models recent sales volatility.
`DayOfWeek_sin`, `_cos`	Cyclical	Numeric	Encodes the cyclical nature of the week.
`Month_sin`, `_cos`	Cyclical	Numeric	Encodes the cyclical nature of the year.
`sell_price`	Domain-Specific	Numeric	Today's price; raw input for the demand model.
`Promo`	Domain-Specific	Binary	Marks days when sell_price is below its 30-day rolling median.
`PromoOnWeekend`	Domain-Specific	Binary	Interaction: promotion active on a Saturday or Sunday.

Table 5.1: Engineered feature summary

We can also encode our business intuition by creating interaction features. A promotion lifts sales, and a weekend lifts sales: the question is whether a promotion on a weekend lifts more than the sum of the two. A binary interaction term answers it:

```
df['PromoOnWeekend'] = df['IsPromo'] * df['DayOfWeek'].isin([5, 6])
```

This new feature is only 1 when a promotion is active on a Saturday or Sunday. The same pattern applies to events, `df['EventOnWeekend'] = df['IsEvent'] * df['DayOfWeek'].isin([5, 6]).astype(int)`, which captures things like a Super Bowl Sunday or an Easter weekend lift. Providing these explicit interaction terms can help the model learn complex relationships more easily.

We've now walked through every feature category the chapter uses. Before packaging them up, the table will summarize the full sequence as a process flow: what each stage produces, why it earns a place in the feature set, and the specific leakage check that has to pass for the stage to be safe. Order matters—shifts and rolling windows depend on a sorted, per-series frame, the warm-up drop only works once lags exist, and the temporal split has to happen last.

Step	Stage	What you compute	Output columns	Why it matters	Leakage check
1	Load and reshape	Read calendar.csv, sales_train_validation.csv, sell_prices.csv. Melt sales to long, merge calendar on d, merge prices on (`store_id`, `item_id`, `wm_yr_wk`).	(long panel)	One unified canvas for every later step.	—
2	Drop pre-launch	Drop rows where `sell_price` is NaN (product not yet on sale at that store).	(cleaned frame)	Removes rows with no genuine demand history.	Don't `bfill` dynamic columns—that copies future values backward.
3	Calendar features	Extract `Year`, `Month`, `Day`, `DayOfWeek`, `WeekOfYear`, `DayOfYear` from Date.	`Year`, `Month`, `Day`, `DayOfWeek`, `WeekOfYear`, `DayOfYear`	Encodes trend and multiple seasonalities directly from the timestamp.	None: derived purely from the row's own date.
4	Event & SNAP flags	`IsEvent` from event_name_1; resolve	`IsEvent`, `snap_active`	Captures known calendar-driven	Use only past events

		`snap_CA/TX/WI` to a single `snap_active` using `state_id`.		demand spikes and SNAP day lifts.	plus known future ones; SNAP calendars are published.
5	Lag features	`groupby('id')['sales'].shift(lag)` for several lags.	`sales_lag_1`, `_7`, `_14`, `_28`	Encodes autocorrelation: yesterday, last week, two weeks ago, last month.	Group by id before shifting so values don't bleed across series.
6	Rolling stats	`groupby('id')['sales'].shift(1).rolling(window).mean()/.std()` for several windows.	`sales_rolling_mean/std_7, _14, _28`	Smooths recent trend and volatility into stable signals.	`shift(1)` BEFORE the rolling call, otherwise the window peeks at today's target.
7	Cyclical encoding	sin/cos pairs for `DayOfWeek`, `Month`, `Day`, `WeekOfYear`.	`DayOfWeek_sin/cos`, `Month_sin/cos`, ...	Preserves cyclical adjacency so Dec sits next to Jan.	None.
8	Domain features	Trailing 30-day rolling median of sell_price, then Promo flag where today's	`price_30d_median`, `Promo`, `PromoOnWeeke`	Encodes pricing and event signals	Use `shift(1)` before the

		price is below 95% of it. PromoOnWeekend / EventOnWeekend interactions.	`nd`, `EventOnWeeke nd`	that trees can't infer from raw columns alone.	rolling median so today's price is excluded.
9	Drop warmup rows	`dropna` on the longest-window rolling column.	(frame minus first 28 rows per series)	Removes rows where lag/rolling features are still NaN.	Drop, don't `bfill`: filling NaNs with future sales is a target leak.
10	Cast and transform	Cast `item_id/dept_id/cat_id/store _id/state`_id to `category`; `y = np.log1p(sales)`.	(typed X, log-transformed y)	LightGBM handles category dtype natively; log1p stabilizes variance.	log1p stabilizes variance only; it does not remove trend (see Hybrid section).
11	Temporal split	`cutoff = max(Date) − 42 days; split X/y on Date ≤ cutoff`.	`X_train/test, y_train/test`	Mimics deployment: predict the future from the past.	Never random-shuffle a time series — train must precede test in time.

Table 5.2: Feature engineering process flow

A practical guide to feature engineering

So far, we have built a strong set of features for forecasting. Calendar effects, lags, and rolling statistics are the foundation of most successful forecasting models I've built. One more category is worth knowing about for forecasting work specifically: **Frequency-Domain Features**. For series with complex seasonality, like weekly data that also has yearly patterns, you can use Fourier features. These transform the time series into a set of sine and cosine waves of different frequencies, letting the model treat seasonality as a global property rather than a per-month dummy.

Other categories, statistical/complexity features (Sample Entropy and friends), shape-based features (shapelets), and the broader automated-feature ecosystem, become important once you step outside forecasting into classification or anomaly detection. We map them in the table for context, and pick them up where they earn their keep: shapelets and the rest in *Chapter 11* on time series classification.

Feature Category	Primary Problem Type	Key Idea	When to Use It	M5 Example
Calendar and Event	Forecasting, Anomaly Detection	Encode known patterns such as holidays, weekends, and promotions.	Always start here. It's the easiest way to capture seasonality.	`DayOfWeek`, `IsEvent`, `snap_active.`
Lags and Rolling Stats	Forecasting	Use recent history (values, means, standard deviations) to predict the future.	The workhorse for most forecasting tasks where the recent past is predictive.	`sales_lag_7`, `sales_rolling_mean_28`.
Frequency (Fourier)	Forecasting	Model seasonality with a combination of smooth sine	When you have multiple complex seasonal patterns that are hard to capture with calendar variables alone.	A smoother yearly seasonal signal than a

		and cosine curves.		single Month column.
Statistical (Entropy)	Anomaly Detection, Classification	Describe the overall character of the series (for example, complexity, stability).	When you need to know if the behavior of a series has changed, not just its value.	Detect that a series has gone from steady weekly demand to erratic spikes.
Automated (for example, `tsfresh`)	Classification, Rapid Baselining	Automatically generate hundreds or thousands of features and test for significance.	Useful for exploring new datasets or when you suspect non-obvious patterns exist. Use with caution to avoid spurious correlations.	Per-id static fingerprints (mean, variance, peak count) merged onto the panel.

Table 5.3: Feature Taxonomy and Descriptions

For the rest of this chapter, we will focus on the first two rows and the core tools for building forecasting models. Keep the table on hand as a quick lookup for when the next project isn't a straightforward forecast.

Building a reusable pipeline

The manual steps in the previous section are great for building intuition, but for production work, they should be encapsulated in a reusable, reproducible pipeline. `scikit-learn`'s `Pipeline`, `FunctionTransformer`, and `ColumnTransformer` are the perfect tools for this. They allow us to chain our custom feature engineering steps into a single object that can be applied to new data with a simple `.transform()` call.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer
from sklearn.compose import ColumnTransformer
# Define functions for each feature engineering step (re-using logic from above)
def create_date_features(df):
    df_out = df.copy()
    df_out['DayOfWeek'] = df_out['Date'].dt.dayofweek
         df_out['Month'] = df_out['Date'].dt.month
    #... add other date features
```

```
    return df_out
def create_lag_features(df):
    df_out = df.copy().sort_values(['Store', 'Date'])
    df_out['sales_lag_7'] = df_out.groupby('id')['sales'].shift(7)
    return df_out
# Create FunctionTransformers
date_transformer = FunctionTransformer(create_date_features)
lag_transformer = FunctionTransformer(create_lag_features)
# Build the pipeline
# Note: A real pipeline would be more complex, handling merges and all feature steps
preprocessing_pipeline = Pipeline(steps=[
    ('date_features', date_transformer),
    ('lag_features', lag_transformer)
])
# Now you can apply the entire workflow in one step
# df_processed = preprocessing_pipeline.fit_transform(df)
```

Choosing a tree model

For our forecasting task, we will use a gradient-boosted tree model. But it's important to understand that there are two main philosophies for applying tree-based models to time series.

The core problem is that standard trees assume i.i.d. data, which time series violates. The two solutions are:

- **Change the data (feature-based):** Transform the time series into a tabular format with features that explicitly encode time dependencies. Then, use a standard tree model like LightGBM.
- **Change the model (specialized):** Use a tree model whose internal splitting logic is specifically designed to work with raw time series subsequences.

The feature-based approach, where we transform the time series into a tabular dataset, is the industry standard for scalable forecasting. Its flexibility allows us to incorporate unlimited external variables. This is where **Gradient Boosted Decision Trees** (**GBDTs**) shine.

A **tree-based model** makes predictions by learning a hierarchy of simple if/then questions about the features. It repeatedly splits the data into smaller, more homogeneous groups. For example, a split might be "*is day_of_week = Sunday?*" or "*is promo_running = True?*". The final prediction is the average target value of the training data points that end up in the same final group. **Gradient boosting** turns a collection of simple trees into a single, accurate predictor. It builds the trees one after another. The first tree makes a simple prediction. The second tree is then trained to predict the errors made by the first tree. The third tree predicts the errors of the second, and so on. Each new tree corrects the mistakes of the previous ones, allowing the model to progressively refine its predictions.

The big three gradient-boosted tree libraries, LightGBM, XGBoost, and CatBoost, are the workhorses for this type of task. They excel at automatically handling complex, non-linear relationships and feature interactions. While all are excellent, LightGBM is often favored for its speed, and CatBoost is particularly strong when you have many categorical features to deal with.

This tree-based approach has one major vulnerability: the extrapolation problem. A tree-based model cannot predict a target value outside the range it saw during training. If your sales data trends upwards, a model trained on past data will never predict a future record-high sale; it will just predict the highest value from its training set.



A subtlety worth flagging now: in the LightGBM workflow that follows, we train on np.log1p(sales). That stabilizes variance, which helps the loss function, but it does not remove the trend. The maximum log-sales the model has ever seen still caps what it can predict. If your validation period sits at a structurally higher trend than training (a growing product line, a post-event demand surge), the model will systematically under-forecast the peaks no matter how well-tuned it is.

We resolve this limitation by making the target stationary. You can achieve this through manual differencing or hybrid decomposition.

Differencing handles non-stationarity by transforming the target into sequential changes. You apply a function like `diff()`, as shown in *Chapter 2*. The tree predicts these changes, and you reconstruct the final values later. Differencing removes the trend entirely. It forces the model to focus on immediate volatility.

The hybrid decomposition pattern offers an alternative. We fit a trend model, subtract it from the data, and train LightGBM on the residual. We add the trend forecast back during prediction. The residual remains stationary when it reaches the tree. Unlike differencing, this method models the overarching direction explicitly. We'll cover this workflow below.

The other paradigm, **specialized trees**, includes shapelet-based models and methods like `TimeSeriesForestRegressor` that consume raw subsequences directly rather than engineered features. They shine at classification (identifying recurring shapes and motifs) more than at multi-store forecasting with rich external variables, so we defer them to *Chapter 11*. The table below summarizes the trade-off.

Aspect	Feature-Based Trees (for example LightGBM)	Specialized Trees (for example, TSF, ROCKET)

Core Idea	Transform the data to fit a standard model.	Transform the model to fit the raw data.
Input Data Format	Flat tabular X/y (one row per timestamp + entity, lots of columns).	Raw time-series subsequences or windows.
Model Type	General-purpose gradient-boosted trees (LightGBM, XGBoost, CatBoost).	Custom tree splits over intervals, shapelets, or random kernels (TSF, RISE, ROCKET).
Primary Use Case	Forecasting (especially with external variables).	Classification (identifying shapes and motifs).
Strengths	Highly scalable, flexible, and easily incorporates external features.	Captures temporal patterns directly, requires less manual feature engineering.
Weaknesses	Requires thoughtful feature engineering.	Less common for forecasting; can be computationally intensive.

Table 5.4: Feature-Based vs. Specialized Trees

For a large-scale task like ours, the choice is clear. The feature-based approach gives us the flexibility to build one model that learns from all series and all available features. We will use LightGBM as our workhorse. Now, let's put our engineered features to work.

As established in *Chapter 3*, we must perform a temporal split to avoid data leakage. We'll use the last 6 weeks of data as our validation set.

```
import lightgbm as lgb
from sklearn.metrics import root_mean_squared_error
# Use the manually created feature dataframe
final_df = df_manual.dropna().copy()
# Define identifier and target (y)
features = [c for c in df.columns if c not in {'id', 'Date', 'sales'}]
categorical_features = ['item_id', 'dept_id', 'cat_id',
                        'store_id', 'state_id']
for col in categorical_features:
    df[col] = df[col].astype('category')
X = df[features]
y = np.log1p(df['sales']) # log transform to stabilize variance
# Temporal split
cutoff_date = df['Date'].max() - pd.to_timedelta('42 days')
```

```
train_mask = df['Date'] <= cutoff_date
X_train, y_train = X[train_mask], y[train_mask]
X_test, y_test = X[~train_mask], y[~train_mask]
```

Next, we'll train a baseline model and then use Optuna, a Python-based hyperparameter optimization framework that helps you automatically find the best parameters for your machine learning models, to systematically tune its hyperparameters.

One detail to be careful about: we trained on `np.log1p(sales)`, so the model's predictions live in log-space. If we score with RMSE directly, the number we get is a **log-RMSE**, which is hard to communicate to a business stakeholder who thinks in units of product sold. The fix is to invert the transform with `np.expm1` before computing the metric, so the objective is RMSE in the original sales units. The same rule applies to anything you want to report or A/B test: if you transform the target, inverse-transform the predictions before measuring error.

```
import optuna
def objective(trial):
    params = {
        'objective': 'regression', 'metric': 'rmse',
        'verbosity': -1, 'boosting_type': 'gbdt', 'random_state': 42,
        'n_estimators': 1000,
        'learning_rate': trial.suggest_float(
            'learning_rate', 0.01, 0.1, log=True),
        'num_leaves': trial.suggest_int('num_leaves', 20, 256),
        'max_depth': trial.suggest_int('max_depth', 3, 12),
        'subsample': trial.suggest_float('subsample', 0.6, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.6, 1.0),
        'min_child_samples': trial.suggest_int('min_child_samples', 5, 200)
    }
    model = lgb.LGBMRegressor(**params)
    model.fit(
        X_train, y_train,
        eval_set=[(X_test, y_test)],
        eval_metric='rmse',
        categorical_feature=categorical_features,
        callbacks=[lgb.early_stopping(50, verbose=False)],
    )
    preds_log = model.predict(X_test)
    preds_units = np.expm1(preds_log)
    actuals_units = np.expm1(y_test)
    return root_mean_squared_error(actuals_units, preds_units)
# study = optuna.create_study(direction='minimize')
# study.optimize(objective, n_trials=20) # Run for 20 trials
# print("Best trial params:", study.best_trial.params)
```

The model was trained on `log1p(sales)`, so predictions are in log-space. Therefore, we invert the transform so the RMSE we minimize is in sales units, the scale stakeholders care about.

Once you have a `best_model` from the Optuna study, you still need to actually generate the 42-day forecast. With `sales_lag_1` in the feature set, a single `model.predict(X_test)` call only fills in

t+1 correctly. Day *t+2* needs `sales_lag_1 = forecast` forex, which doesn't exist yet. The recursive loop below regenerates the lag and rolling features one day at a time, using the previous prediction as the most recent "*actual*":

```
# Recursive 42-day forecast for a single item-store series
def recursive_forecast(model, history, future_dates, features):
    """history: DataFrame with all past sales for one id, indexed by Date.
    future_dates: DatetimeIndex of dates to forecast.
    Returns a Series of predictions in the original sales units.
    """
    series = history['sales'].copy()
    preds = []
    for d in future_dates:
        # Build the feature row for date d using only past values.
        row = {
            'sales_lag_1':  series.iloc[-1],
            'sales_lag_7':  series.iloc[-7],
            'sales_lag_14': series.iloc[-14],
            'sales_lag_28': series.iloc[-28],
            'sales_rolling_mean_7':  series.iloc[-7:].mean(),
            'sales_rolling_mean_14': series.iloc[-14:].mean(),
            'sales_rolling_mean_28': series.iloc[-28:].mean(),
            'sales_rolling_std_7':   series.iloc[-7:].std(),
            'sales_rolling_std_14':  series.iloc[-14:].std(),
            'sales_rolling_std_28':  series.iloc[-28:].std(),
            # ... plus the calendar / item / store features for date d
        }
        X_d = pd.DataFrame([row])[features]
        yhat_log = model.predict(X_d)[0]
        yhat = np.expm1(yhat_log) # back to original sales units
        preds.append(yhat)
        # Append the prediction so the next iteration can lag against it
        series.loc[d] = yhat
    return pd.Series(preds, index=future_dates)
```

The key idea is `series.loc[d] = yhat`: yesterday's forecast becomes today's "*lag 1*". This is what we meant by recursive error propagation. If *t+1* is off by 5%, that error feeds the lag features for *t+2*, and so on. In practice, the recursive strategy works well for short horizons but degrades as *h* grows. For 6 weeks of daily forecasts it's usually serviceable; for 6 months it usually isn't.

The table summarizes the key hyperparameters, their search ranges, and the rationale behind their inclusion.

Hyperparameter	**Search Range**	**Description/Impact**
`learning_rate`	`float(0.01, 0.1)`	Controls the step size. Lower is slower but often more accurate.

`num_leaves`	`int(20, 256)`	Main controller of tree complexity in LightGBM.
`max_depth`	`int(3, 12)`	Limits tree depth to prevent overfitting.
`subsample`	`float(0.6, 1.0)`	Fraction of data sampled per tree; reduces variance.
`colsample_bytree`	`float(0.6, 1.0)`	Fraction of features sampled per tree; useful with many features.
`min_child_samples`	`int(5, 200)`	Minimum data points in a leaf; raises the bar for splits and curbs overfitting on noisy series.

Table 5.5: Key Hyperparameter Search Space



Author's Insight

Our model produces a single number, a point forecast. But in business, understanding the range of possible outcomes is often more important. How confident are we in this prediction? Could sales be 20% higher or 30% lower? This is the domain of **uncertainty quantification**. Tree-based models can be adapted to produce prediction intervals using techniques like quantile regression. We will dedicate *Chapter 8* to this critical topic.

Specialized tree models like `TimeSeriesForestRegressor` are useful for time-series classification, but they don't fit our problem: thousands of series, many external features, and a 6-week horizon. The feature-based approach with gradient-boosted trees scales to all of that. We'll use LightGBM as our workhorse.

Hybrid modeling and decomposition

This is the technique we promised in the extrapolation discussion above. The recipe is a direct response to LightGBM's inability to predict above its training range: take the trend out before fitting, and add it back at prediction time.

Pattern 1: Decompose-then-model-residuals

We use a statistical method to decompose the series into trend, seasonal, and residual components. The trend is forecast with a simple model that can extrapolate (a linear regression on time, an ETS, or a Prophet trend). LightGBM is trained only on the residual, which is approximately stationary and lives entirely within the range the tree has seen.

```
from statsmodels.tsa.seasonal import seasonal_decompose
# Decompose a single item-store series to illustrate
single_id = df['id'].iloc[0] # any id with enough history
single_series = (df[df['id'] == single_id]
    .sort_values('Date')
    .set_index('Date')['sales'])
result = seasonal_decompose(
    single_series, model='additive', period=7,
    extrapolate_trend='freq'
)
# Train LightGBM on result.resid (using the same engineered features as before).
# At inference time: forecast the trend separately, then add the residual prediction back. resu
```

Pattern 2: Fourier features for seasonality

A lighter-weight alternative is to skip the explicit decomposition and let the model learn the seasonal pattern from Fourier columns directly:

```
from sktime.transformations.series.fourier import FourierFeatures
from sktime.datasets import load_airline
y = load_airline()
transformer = FourierFeatures(sp_list=[1], fourier_terms_list=[2, 3])
y_transformed = transformer.fit_transform(y)
print(y_transformed.head())
```

This creates `sin_12_1`, `cos_12_1`, etc., which can be fed into LightGBM alongside the rest of the engineered features. Modern neural architectures like N-BEATS push this idea further by learning the decomposition end-to-end, which we cover in *Chapter 7*. Decomposition can also be skipped entirely in favor of automating feature creation, and that's where we're headed next.

Automated feature engineering

The manual feature engineering process described in the previous sections is powerful and interpretable but can be time-consuming. For complex problems or situations where domain knowledge is limited, **automated feature engineering** libraries can provide a significant productivity boost and potentially uncover predictive patterns that a human might miss.

The tool choice rests mainly on resource constraints. While some libraries are very quick, others rely on heavy computations and can take longer. Except for TSFEL, all have optimizations for parallelism or compiled backends (like catch22).

Libraries like `tsfresh`, `Featuretools`, and the newer, high-performance `functime` are designed to automatically extract a large number of characteristic features from time series data. Instead of manually coding lag and rolling window features, these tools can generate hundreds or even thousands of features based on statistical properties (mean, variance, skew), frequency domain characteristics (Fourier coefficients), and other complex metrics.

For example, `tsfresh` can calculate features like the number of peaks in a series, the autocorrelation at different lags, and the coefficients of a linear trend, among many others. The library also includes methods for feature selection, which evaluate the significance of each generated feature with respect to the target variable, helping to filter out irrelevant or redundant information and reduce the risk of overfitting.

Library	**Main Functionality**	**Strengths**	**Limits**	**Integration Notes**
`tsfresh`	Large automatic feature extractor and selection.	Very broad feature set, great for baselines.	Can be compute-heavy, needs careful time-aware setup.	`sktime` has first-class wrappers.
`Featuretools`	Automated features for relational/tabular data.	Excellent for relational aggregations (for example, customers -> transactions).	Less tuned for raw univariate statistical features.	Generate features, then pass to `sklearn` or other libraries.
`TSFEL`	Statistical/spectral features for signals.	Rich signal/spectral features, good for sensor data.	Less integrated into `sktime` by default. Mostly academic use.	Run TSFEL to produce tables, then feed to other tools.

`catch22`	Tiny, canonical set of 22 features.	Extremely fast, good for quick baselines.	Only 22 features; may miss domain signals.	`sktime` includes Catch22 transformers.
`functime`	High-performance feature engineering and forecasting.	Very fast and memory-efficient due to its Polars backend. Expressive syntax. Great for large panel datasets.	Newer library, smaller community. Requires data in Polars format.	Extract features and convert back to `pandas`/`numpy` to use with `sklearn` models, or use its native forecasters.

Table 5.6: Time Series Feature Engineering Library Comparison

The `catch22` entry in our table is a curated toolkit with a story. The name nods to the novel Catch-22, which describes a paradoxical no-win situation. In time series analysis, the paradox is this: building a great model requires finding the right features, but testing thousands of candidate features is computationally expensive and risks overfitting.

The `catch22` feature set is the result of a massive academic study that distilled thousands of candidate features down to the 22 most consistently informative and computationally cheap ones. The name is also an acronym: **C**anonical **A**nd **T**ime-series **CH**aracteristics.

While the features are general-purpose, they are optimized for time series classification, not forecasting. We are introducing the concept here to complete your mental map of feature engineering. We will put `catch22` into practice in *Chapter 11* when we tackle time series classification problems head-on.

Here are three practical recipes for integrating feature selection tools.

Recipe 1: mlforecast and XGBoost

This pattern automates the creation of lag and rolling features for an XGBoost model.

```
from xgboost import XGBRegressor
from mlforecast import MLForecast
from mlforecast.lag_transforms import RollingMean
fcst = MLForecast(
    models=[XGBRegressor(random_state=42)],
    freq="D",
    lags=[4, 5],
```

```
    lag_transforms={
        7: [RollingMean(window_size=14)] # Create a 14-day rolling
                                         # mean of the 7-day lag
    },
    date_features=["dayofweek", "month"],
)
# fcst.fit(df)
# predictions = fcst.predict(14)
```

`models` takes a list containing an instance of `XGBRegressor`. `lag_transforms` is a dictionary where the key (7) is the lag to `transform`, and the value is a list of transformations (a 14-day rolling mean).

Recipe 2: tsfresh and Darts

In this recipe, we'll walk through a complete example of how to use static features, characteristics that describe an entire time series. We will use `tsfresh` to automatically extract these features. The core idea is to give our model high-level context about each time series it's trained on. By understanding the overall properties of a series (like its volatility or trend), the model can make more intelligent forecasts.

First, we need some data to work with. We'll create a synthetic dataset containing sales data for two different stores. The M5 panel is complex, with thousands of series, calendar events, and price dynamics. A synthetic dataset isolates the single concept we want to teach—how to create and use static features—without any distracting noise.

To make the task interesting, we'll give each store a distinct sales pattern: one will have a simple linear trend, and the other will have a clear seasonal (sine wave) pattern.

```
import pandas as pd
import numpy as np
import lightgbm as lgb
# Create a sample DataFrame with two stores (IDs 1 and 2)
# Store 1 has a simple linear trend, Store 2 has a sine wave pattern
data = []
for store_id in [1, 2]:
    dates = pd.date_range(start="2025-01-01", periods=50, freq="D")
    if store_id == 1:
        # Simple upward trend
        sales = np.arange(50) + np.random.rand(50) * 5
    else:
        # Seasonal pattern
        sales = 25 * np.sin(np.arange(50) / 4) + 50 + np.random.rand(50) * 5
    store_df = pd.DataFrame({
        "unique_id": store_id,
        "ds": dates,
        "y": sales
    })
```

```
    data.append(store_df)
df = pd.concat(data)
```

With this step complete, we have a single pandas DataFrame named `df`. It contains the sales history for two distinct stores, identified by `unique_id`. This format (`unique_id`, `ds`, `y`) is the standard input for many modern forecasting libraries, including `mlforecast`.

Now, we'll use `tsfresh` to analyze the sales history of each store and generate descriptive statistics. These features, like the mean, variance, or trend characteristics, will serve as the high-level context for our global model.

```
from tsfresh import extract_features
from tsfresh.feature_extraction import EfficientFCParameters
# tsfresh expects specific column names: (id, time, value)
df_tsfresh = df.rename(
    columns={'unique_id': 'id', 'ds': 'time', 'y': 'value'})
# Extract an efficient set of features for each series ('id')
static_features = extract_features(
    df_tsfresh,
    column_id="id",
    column_sort="time",
    default_fc_parameters=EfficientFCParameters()
)
# tsfresh uses the id as the index, let's make it a regular column
static_features = (
    static_features
    .reset_index()
    .rename(columns={'index': 'unique_id'})
)
```

In the `static_features` DataFrame, each row corresponds to a single store, and the columns represent the statistical features calculated by `tsfresh`. We have successfully summarized each store's unique sales history into a single row of data.

`tsfresh.extract_features` calculates statistics (like mean, variance, number of peaks) over the entire history of each store's sales. This creates a static *fingerprint* for each store. It is critical that the rows in your `static_covariates` DataFrame are in the same order as your list of target TimeSeries objects. Reindexing the `static_features` DataFrame ensures each time series is correctly matched with its feature fingerprint.

To use these fingerprints in a forecasting pipeline, we merge them back onto the long-format training frame on `unique_id`, so every row for a given store carries that store's static features as additional columns:

```
# Merge static fingerprints back onto the training rows.
# Every row for a given store gets the same fingerprint values.
df_with_static = df.merge(static_features, on='unique_id', how='left')
# Now df_with_static has the original (unique_id, ds, y) plus dozens of
```

```
# tsfresh columns. Pass it to MLForecast or LightGBM like any other feature set.
print(df_with_static.shape)
print(df_with_static.columns[:10])
```

With the static and dynamic features sitting side-by-side in one frame, we can instantiate MLForecast with LightGBM as our regressor and let it learn from both the time-varying signals (lags, rolling means) and the per-series fingerprints (overall trend, volatility, peak count).

Recipe 3: functime

`functime` is a modern library built on Polars that is designed for fast feature extraction and forecasting on large panel datasets. Its syntax is expressive and integrates well into a pandas-like workflow. The following code block provides an example of how `functime` could be used to generate a suite of features for the M5 data.

```
import polars as pl
import functime as ft
# Convert pandas DataFrame to Polars for functime
pl_df = pl.from_pandas(df)
pl_df = pl_df.rename({id: 'entity', 'Date': 'time', 'sales': 'value'})
# Define the feature extraction logic
# This will compute a set of features for each store's time series
feature_extractor = ft.FeatureExtractor(
    freq="1d",
    lags=[7, 14],
    rolling_windows=[7, 14, 28], # ADDED: Example window sizes
    rolling_agg_fns=["mean", "std"],
    tsfresh_features={
        "abs_energy": None,
        "c3": [{"lag": 1}],
        "autocorrelation": [{"lag": 7}]
    }
)
# Extract features
# X_features = feature_extractor.fit_transform(pl_df)
# X_features.head()
```

So far, we have manually crafted features based on our domain knowledge and statistical principles. An alternative approach, known as **representation learning**, aims to learn these features automatically.

One technique for this is the **autoencoder**. It's a neural network that learns to compress a window of time series data into a compact vector of numbers, and then reconstruct the original window from that compressed vector. The compressed vector from the network's central so-called bottleneck layer becomes a set of learned features. It captures the core patterns of the input sequence in a way a downstream model can easily use.

This is a deep learning technique, and while it's beyond the scope of this chapter, we will dive deep into autoencoders and other architectures like LSTMs and Transformers in *Chapter 8*.

Let's summarize the trade-offs between automated and manual feature engineering in a table:

Approach	**Pros**	**Cons**	**Best For...**
Manual Engineering	Highly interpretable, embeds domain knowledge.	Time-consuming, relies on human expertise.	Production models requiring explainability.
Automated Engineering	Fast, discovers novel patterns, good for baselines.	Less interpretable, risk of spurious correlations, can be computationally expensive.	Rapid baseline creation, problems with unknown domain drivers.

Table 5.7: Automated vs. manual feature engineering

A balanced approach is often best: start with a strong foundation of manual, interpretable features and augment them with automatically generated ones.

Summary

This chapter provided a hands-on guide to the dominant workflow for modern time series forecasting: treating it as a regression problem. Using the M5 Walmart sales challenge, we built a complete forecasting system from the ground up.

In this chapter, you developed several key skills for time series forecasting. You learned to perform systematic feature engineering, creating a hierarchy of features ranging from foundational lags and rolling windows to contextual calendar effects and domain-specific signals. You now understand the two main approaches to using tree-based models, feature-based vs. specialized, and can choose the appropriate one for your task.

You gained hands-on experience by training a high-performance LightGBM model, the industry workhorse for tabular data, and learned to tune its hyperparameters for optimal performance. Furthermore, you explored advanced hybrid techniques, combining classical decomposition, and Fourier analysis with machine learning to capture complex patterns like trends and seasonality. Finally, you developed a mental map for navigating the ecosystem of

automated feature engineering, enabling you to understand the trade-offs and integration patterns of key libraries.

The techniques in this chapter underpin most production forecasting systems we've worked on. Before reaching for deep learning, the feature-based gradient-boosted-tree approach should be the default. The same engineered features feed any other regressor too. In later chapters, the same feature engineering and time-aware validation principles carry over to neural networks; the workflow does not change.

In the next chapter, we will move beyond individual time series to explore multivariate and hierarchical forecasting, learning how to model relationships across multiple variables and reconcile forecasts across different levels of a hierarchy.

Get this book's PDF version and more

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

6

Multivariate and Hierarchical Forecasting

In previous chapters, we focused on univariate forecasting: predicting one time series from its own historical data. In *Chapter 3*, we used ARIMA to forecast Google Trends data based solely on its past values. In *Chapter 5*, we built features from each M5 item-store pair's sales history to predict its future sales. While we trained one model across all 30,490 series, we treated each item-store as conditionally independent, assuming one product's sales don't directly influence another's orders.

That approach, known as **univariate forecasting**, is the foundation for everything we have done so far, but it relies on a simplifying assumption that breaks down in two scenarios: when variables actively influence each other and when forecasts must aggregate consistently across levels.

Real-world data is rarely so neatly isolated. In practice, most business, and operational systems are deeply interconnected. The sales of one product may depend on the promotions or pricing of another. Regional sales roll up into national totals, which in turn influence corporate forecasts and production planning. Demand in

one sector may drive supply in another. When these relationships are ignored, valuable predictive signals are lost.

This chapter introduces two approaches that capture the structure and relationships in real-world business systems. **Multivariate forecasting** models how multiple variables influence each other over time. When advertising spend, sales, and inventory form a feedback loop, we need methods that capture these cross-variable dependencies explicitly. We'll explore **Vector Autoregression** (**VAR**) and its extension to include **external variables** (**VARMAX**). Alongside these, we'll cover global cross-learning with the `mlforecast` library, where a single tree model trains across thousands of series at once. Cross-learning is a different idea from multivariate forecasting: it scales by sharing parameters across series rather than by modeling cross-series feedback loops directly. We make that distinction explicit when we get there, because conflating the two is one of the most common pitfalls in applied forecasting.

Hierarchical forecasting addresses data organized in aggregation structures. When you forecast sales at the store level, regional level, and national level, these forecasts must be coherent in the sense that the regional forecasts must sum to the national total. We'll explore reconciliation methods, particularly the modern MinT approach, that ensure mathematical consistency while preserving forecast accuracy.

The Australian Tourism case study will show how reconciliation improves both forecast accuracy and operational consistency.

In this chapter, we will explore:

- Multivariate models
- Hierarchical models

- Case study: Forecasting Australian tourism

The two approaches, multivariate models and hierarchical models, are complementary. Together, they form a toolkit for building forecasts that better reflect the interconnected, hierarchical, and uncertain nature of real-world data systems.

By the end of this chapter you will recognize when univariate approaches break down and identify when relationships between variables or aggregation constraints must be modeled explicitly. You will also learn how to apply multivariate forecasting techniques to capture interactions between variables where appropriate and understand hierarchical forecasting systems, including when coherence matters and how reconciliation methods improve both consistency and accuracy. Finally you will be able to choose the right modeling strategy for real-world systems, balancing interpretability, scalability, and robustness across multivariate, hierarchical, and global approaches.

We begin with multivariate forecasting.

Multivariate models

In December 2011, Best Buy canceled thousands of online holiday orders hours before Christmas. The company blamed *overwhelming demand* from Black Friday promotions. The real culprit was a forecasting system that treated sales as an isolated time series, unable to connect planned promotions to inventory needs. This single failure mode, ignoring correlations between variables, costs global retailers an estimated $1.8 trillion annually in stockouts and lost sales.

The energy sector faces an equally dangerous version of this problem. In August 2020, California's grid operators ordered rolling blackouts affecting millions of residents. Their forecasting models had overestimated solar generation during evening peak demand. As the sun set during a heatwave, solar output plummeted while air conditioning demand remained high. The models failed to capture the critical negative correlation between solar availability and evening load during extreme weather events.

Both these failures share a commonality. Univariate models see only historical patterns in a single series. They cannot anticipate demand spikes from promotions that haven't happened yet. They cannot predict supply shortfalls from weather conditions that differ from historical norms. When the world operates as a system of interacting variables, forecasting one variable in isolation is a recipe for failure.

In previous chapters, we focused on univariate forecasting. We used a variable's own past to predict its future. In many business and scientific systems, though, variables are interdependent. Ignoring these relationships means leaving valuable predictive information on the table. Multivariate time series analysis allows us to model these interdependencies explicitly. This section explores when these methods are necessary, how classical approaches work, their fundamental limitations, and the modern library ecosystem that has emerged to solve these problems at scale.

What are multivariate forecasts?

Multivariate methods are designed to forecast several time series simultaneously. Unlike univariate models that rely solely on a single

series' past, multivariate models allow one time series to inform and improve the forecasts of another.

The term multivariate forecasting is often used inconsistently in the literature, so let's clarify it precisely.

Multivariate time series forecasting refers to forecasting multiple time series simultaneously, where each series may influence the others.

Sometimes multivariate is mistakenly used to mean forecasting one time series using multiple predictors. This is more correctly called a **univariate forecast with exogenous variables** (or **multivariate input, univariate output**). For example, predicting sales from its own history plus advertising and price data.

Formally, in multivariate forecasting, you have several time-dependent variables observed over time (for example, sales, advertising spend, and inventory). The goal is to model their **joint dynamics** (the mathematical description of how multiple variables change together), which is how each variable evolves and how they interact.

So, in a multivariate model, the target is a vector of time series, not a single one.

For example:

$$y_t = [y_{\{1,t\}}, y_{\{2,t\}}, \ldots, y_{\{k,t\}}]$$

And we model (y_t') as a function of its past values and possibly the past values of all other variables:

$$y_t = f(y_{t\{t-1\}}, y_{\{t-2\}}, \ldots, y_{\{t-p\}}) + \varepsilon_t$$

Let's look at some scenarios where multivariate forecasting is essential:

- **Leading indicators**: Series A changes before Series B responds. Building permits are the canonical example. A developer pulls a permit months before any concrete is poured, so the count of permits issued in March directly informs the count of construction workers hired between June and September. A VAR model can quantify the relationship explicitly, asking how much a 10% jump in permits in month t shifts employment in month $t + 4$. A Granger causality test then checks whether that lead-lag effect is statistically real or an artifact of the training period.
- **Correlated series**: Variables move together because they share a hidden driver. Electricity demand in Phoenix, Tucson, and Mesa rises and falls almost in lockstep, not because the cities affect each other but because they sit under the same desert heat dome. Modeling them jointly lets a sudden uptick in Tucson's load contribute to Phoenix's forecast within minutes, rather than waiting for that signal to reach Phoenix through its own meter readings. This shared-driver pattern is also why VAR's coefficients can look sensible even when there is no direct causal link, a point we return to when we discuss Granger causality.
- **System dynamics**: Variables form feedback loops where each influences the others. The bullwhip effect, expanded on shortly, is the textbook case. A small spike in retail demand pushes the wholesaler to over-order, which pushes the manufacturer to over-produce, which propagates back as an inventory glut a quarter later. A univariate forecast of the manufacturer's output

series cannot see the retail signal that started the cascade. A VAR model that includes all three tiers can.

While identifying these relationships provides a more complete picture of a system's dynamics than a univariate forecast will, there is an important constraint we must remember: to use a variable as a predictor, you must have its future values at the time of the forecast.

Consider these examples, and notice that only some of them are multivariate in the strict sense:

- **Economics** (**joint multivariate**): GDP, interest rates, inflation, and unemployment co-evolve. A central bank moving rates affects unemployment, which feeds back into inflation expectations and back into GDP growth. None of these is a clean "input"; each is itself a target that influences the others. This is exactly what VAR was designed for.
- **Supply chain** (**joint multivariate**): Retailer, wholesaler, and manufacturer orders form the bullwhip feedback loop discussed above. Forecasting any one of them in isolation misses the cascade.
- **Retail** (**univariate with exogenous inputs**): Ice cream sales depend on weather, but the retailer does not forecast weather; they consume someone else's meteorological forecast. Sales is the only target. This fits the univariate-with-exogenous pattern that **Vector Autoregression Moving Average with eXogenous variables** (**VARMAX**), covered shortly, addresses.
- **Energy** (**univariate with exogenous inputs**): Forecasting electricity demand uses past demand plus temperature,

humidity, and wind speed, again sourced from a meteorological model rather than co-modeled.

The line between true multivariate and univariate-with-exogenous is whether the side variables are themselves targets or inputs you already have forecasts for. Tooling differs accordingly: VAR for joint forecasting where every series is a target; VARMAX, or any regression with exogenous regressors, when the side variables come from elsewhere. Both bring valuable predictive information into the model; what changes is how many forecasts you produce and how the parameters are estimated.

The energy sector's struggle reveals another lesson. Legacy forecasting models, often simple ARIMA or linear regressions against temperature, were designed for an era of centralized power generation. These univariate or minimally multivariate models are failing under the complexity of the modern grid.

In this case, several critical correlations are systematically underestimated:

- **Extreme weather events**. Models trained on historical data cannot predict unprecedented conditions. During Christmas 2022, Duke Energy implemented rolling blackouts when a cold snap proved far more severe than forecasts predicted. Heating demand spiked 6-10% above predictions while extreme cold caused equipment failures at multiple power plants. The model failed on both the demand side and the supply side.
- **New demand sources**. Electric vehicle adoption and AI data centers create massive new loads with consumption patterns not reflected in historical data. Traditional models trained on

residential and industrial usage cannot forecast the scale or timing of this new demand.

To understand why system-level forecasting matters, let's understand something called the **bullwhip effect** demonstrates. Consider a three-tier supply chain: retailer to wholesaler to manufacturer. When the retailer experiences a demand spike, they place an emergency order with the wholesaler. The wholesaler sees this large order and interprets it as a signal about future demand, ordering even more from the manufacturer while building safety stock. The manufacturer, seeing amplified demand signals, ramps up production. When retail demand returns to normal, the entire chain has excess inventory.

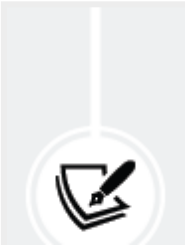
This is fundamentally a multivariate problem: three interconnected time series (retailer orders, wholesaler orders, manufacturer orders) where each series influences the next. A VAR model (discussed in the next section) can capture these feedback loops explicitly:

$$Retailer\ Orders_t = f(Retailer\ Orders_{t-1}, Wholesaler\ Response_{t-1})$$

$$Wholesaler\ Orders_t = f(Retailer\ Orders_{t-1}, Wholesailer\ Orders_{t-1})$$

$$Manufacturer\ Production_t = f(Wholesaler\ Orders_{t-1}, Manufacturer\ Production_{t-1})$$

Treating these as independent univariate forecasts ignores the systemic amplification that causes the bullwhip. A multivariate model captures how shocks propagate through the system.



Note on promotions: If the retailer's demand spike was caused by a planned promotion, that promotion should be included as an exogenous variable (covered in the

next section on VARMAX), not as an additional forecast target.

Several methods exist for multivariate forecasting:

- **Vector Autoregression** (**VAR**)
- **Vector Error Correction Models** (**VECM**)
- Dynamic factor models
- Multivariate RNNs or transformers that predict multiple correlated time series

Let's look at VAR and VARMAX in depth now.

VAR and VARMAX

VAR is a standard method for multivariate forecasting. It is a direct extension of the **Autoregressive** (**AR**) models from *Chapter 3*.

An AR model explains a variable using its own past values. A VAR model extends this logic to multiple variables. In a VAR system, each variable is modeled as a linear function of its own past values and the past values of all other variables in the system.

Consider a simple two-variable system: advertising spend y_1 and sales y_1. A VAR(1) model (with one lag) would have two equations:

$$y_{1,t} = c_1 + \phi_{11,1} y_{1,t-1} + \phi_{12,1} y_{2,t-1} + \epsilon_{1,t}$$

$$y_{2,t} = c_2 + \phi_{21,1} y_{1,t-1} + \phi_{22,1} y_{2,t-1} + \epsilon_{2,t}$$

The current value of advertising spend depends on its own past value and on past sales. Sales depend on both past sales and past advertising. This captures bidirectional feedback. High advertising

may drive sales, but high sales may also signal the need for more aggressive advertising to maintain momentum.

A simple VAR model

The `statsmodels` library in Python provides an implementation of VAR. The process involves checking for stationarity, fitting the model, and generating forecasts.

```
import pandas as pd
import numpy as np
from statsmodels.tsa.api import VAR
from statsmodels.tsa.stattools import adfuller
from sklearn.metrics import mean_squared_error
# A small synthetic example of how advertising drives next-day sales
np.random.seed(42)
dates = pd.date_range('2022-01-01', periods=200, freq='D')
advertising = 100 + np.random.normal(0, 5, 200).cumsum() * 0.1
sales = 50 + 0.4 * np.roll(advertising, 1) + np.random.normal(0, 3, 20
data = pd.DataFrame({'advertising': advertising, 'sales': sales},
                    index=dates)
# Step 1: Check for stationarity (required for VAR)
for col in data.columns:
    result = adfuller(data[col])
    print(f'ADF Statistic for {col}: {result[0]:.4f}')
    print(f'p-value: {result[1]:.4f}')
    # If p-value > 0.05, series is likely non-stationary
    # Apply differencing to achieve stationarity

# For this example, assume data is stationary or differenced once
# data_diff = data.diff().dropna()
# Step 2: Fit the VAR model with automatic lag selection
model = VAR(data)
results = model.fit(maxlags=15, ic='aic')
print(results.summary())
# Step 3: Generate forecasts
lag_order = results.k_ar
forecast_input = data.values[-lag_order:]
# Forecast 5 steps ahead
fc = results.forecast(y=forecast_input, steps=5)
```

```
# Create a DataFrame with forecasts
forecast_df = pd.DataFrame(
    fc,
    columns=data.columns,
    index=pd.date_range(start=data.index[-1], periods=6, freq='D')[1:]
)
print(forecast_df)
```

Running this on our synthetic dataset prints something like:

```
            advertising       sales
2022-07-20    99.842184   90.119027
2022-07-21   100.011327   90.262304
2022-07-22   100.094582   90.348517
2022-07-23   100.135427   90.398443
2022-07-24   100.155483   90.426924
```

The values are on the original scale here because we did not difference the data. If you do difference (which you usually will, since real sales and advertising data are non-stationary), the forecast you get back is in differences too, and you have to undo that step to recover levels. Concretely:

```
# If you fitted VAR on data_diff = data.diff().dropna(),
# the forecast 'fc' contains differences, not levels.
fc_diff = results.forecast(y=data_diff.values[-lag_order:], steps=5)
# Reverse the differencing: cumulative-sum the forecast deltas, then
# add the last observed level to anchor the path.
last_level = data.iloc[-1].values
fc_levels = last_level + np.cumsum(fc_diff, axis=0)
forecast_df = pd.DataFrame(
    fc_levels,
    columns=data.columns,
    index=pd.date_range(start=data.index[-1], periods=6, freq='D')[1:]
)
```

The cumulative sum walks the forecast forward in differences, and adding the last observed value pins the path to where the series actually was at the end of training. If you differenced twice, you cumulate twice; if you took log-differences, you also need to exponentiate at the end.

The code first checks if each series is stationary using the Augmented Dickey-Fuller test. Similar to what we discussed in chapter 3 for classical models, VAR assumes stationarity, and we have to test for stationarity. If the p-value from the ADF test exceeds 0.05, you must difference the data. The model then selects the optimal lag order using the **Akaike Information Criterion** (**AIC**). Finally, it generates multi-step-ahead forecasts and, if the data was differenced, undoes the transformation to return to the original scale.

In practice, not every variable meaningfully influences every other variable. While VAR assumes a fully connected system, some relationships may be weak, redundant, or entirely spurious. Identifying which variables truly matter is critical not only for interpretation but also for building more efficient and reliable models.

This is where Granger causality becomes a very important diagnostic tool.

Granger causality

A useful feature of VAR models is the ability to test for Granger causality. If past values of γ_0 significantly improve predictions of ρ_k beyond what $x_2{}'$'s own history provides, we say γ_0 Granger-causes ρ_k

. This doesn't prove true causation, but it does identify predictive relationships.

```
from statsmodels.tsa.stattools import grangercausalitytests
# Test if advertising Granger-causes sales
# maxlag=4 tests lags from 1 to 4
results = grangercausalitytests(data[['sales', 'advertising']], maxlag
# If p-values are consistently below 0.05, advertising helps predict s
```

Column order matters here: the second column is tested as a cause of the first. The function prints four test statistics per lag (`ssr_ftest`, `ssr_chi2test`, `lrtest`, `params_ftest`); they usually agree, so you can read the p-value of whichever one you prefer. If $p < 0.05$ at a given lag, reject the null of "no Granger causality" and conclude that advertising's past values help predict sales beyond what sales' own past already provides. If p-values stay above 0.05 across all lags, you cannot reject the null and advertising does not appear to Granger-cause sales in this dataset.

This is particularly valuable in exploratory analysis. It provides statistical evidence about which variables in a system actually contain predictive information about others.

VAR models only use the past values of the variables within the system. Many real-world problems require incorporating external information that isn't part of the forecasted variables. This is where VARMAX becomes essential.

VARMAX

Consider the retail stockout problem. You want to forecast sales, but you also know the dates of planned promotions. Promotions aren't part of the sales series itself but rather external interventions. VARMAX allows you to include a binary `is_promotion` variable as an exogenous input.

```
from statsmodels.tsa.statespace.varmax import VARMAX
# Assume 'data' has columns ['sales_product_a', 'sales_product_b']
# 'exog_data' has a binary 'is_promotion' column
# Split into train/test
train_data = data[:'2023-12-31']
test_data = data['2024-01-01':]
train_exog = exog_data.loc[train_data.index]
test_exog = exog_data.loc[test_data.index]
# Fit VARMAX model
model = VARMAX(train_data, exog=train_exog, order=(2, 0))
results = model.fit(disp=False)
# Forecast with known future exogenous variables
fc = results.forecast(steps=len(test_data), exog=test_exog)
print(fc)
```

Please note that `order=(2, 0)` represents an Autoregressive order of 2 and a Moving Average order of 0. The critical requirement is that you must provide the exogenous variables for the forecast period. If you're forecasting 10 days ahead, you need to know whether promotions are planned for each of those 10 days. This aligns perfectly with business planning processes where marketing campaigns are scheduled in advance.

For the energy grid problem, weather forecasts serve as exogenous variables. You can include predicted temperature, cloud cover, and wind speed from meteorological models to improve load forecasts.

With VAR or VARMAX, a modest system with 10 time series and 2 lags requires estimating over 200 parameters. A realistic business scenario with 50 products and 2 lags needs over 5,000 parameters. This quickly becomes computationally infeasible and statistically unreliable. You need massive amounts of data to estimate that many parameters accurately, and the model becomes extremely slow to train and forecast.

Further, VAR assumes that the time series are stationary, their statistical properties don't change over time. Real-world business data violates this constantly. Sales have trends. Markets have structural breaks. Achieving stationarity typically requires differencing, which makes model interpretation more difficult and can obscure long-run relationships.

When VAR is the wrong tool, in practice, four signs tell you to stop fighting VAR and pick something else:

- You have more than roughly ten series and a short history. Parameter count grows quadratically in the number of series, so a VAR(2) on 50 products needs over 5,000 coefficients to estimate. With anything short of huge datasets those estimates are dominated by noise.
- The Augmented Dickey-Fuller test still rejects stationarity after one or two rounds of differencing. Persistent non-stationarity often points to cointegration, long-run equilibrium relationships between series, that VAR cannot represent cleanly.
- Coefficients change sign or magnitude across rolling-window refits. That is a structural break or regime shift, and the linear time-invariant assumption baked into VAR no longer holds.

- On out-of-sample data, the VAR forecast errors are no smaller, or are larger, than independent univariate baselines. The cross-series signal you hoped to capture is either too weak or too unstable to pay for the extra parameters.

Each sign points to a different alternative:

- **Vector Error Correction Models (VECM)**: When series are cointegrated (oil and gas prices drift apart in the short run but revert to a long-run equilibrium), VECM is the principled extension of VAR. statsmodels supports it directly.
- **Regularized or Bayesian VAR**: LASSO-VAR shrinks irrelevant cross-coefficients to zero; Bayesian VAR with Minnesota-style priors does similar work. Both make VAR usable in the 10-50 series range where ordinary least squares collapses.
- **Dynamic Factor Models**: When many series are driven by a small number of common shocks (100 stock returns driven by 3-5 macro factors), factor models extract those latent drivers and forecast them, sidestepping the parameter explosion.
- **Global tree models with engineered cross-series features**: This is the `mlforecast` direction we cover next. You give up VAR's principled feedback loops but gain the ability to fit thousands of series at once, and you can recover some cross-series signal by hand-engineering lagged regional aggregates as features.
- **Neural multivariate models**: Architectures like Temporal Fusion Transformer or PatchTST (*Chapter 7*) learn cross-series interactions through attention, with the regularization that comes from training on large pooled datasets.

These alternatives ultimately reflect a broader modeling choice: whether to explicitly learn interactions between series or to treat each series independently.

Channel-Mixing Models (like VAR) actively learn correlations and dependencies between different time series. When these correlations are stable and meaningful, this approach can dramatically improve forecasts by using information from related series. VAR's Granger causality tests explicitly identify these relationships.

This approach is also brittle. If the learned correlations are spurious—correlations that exist in the training data but don't represent true causal relationships—the model will fail on new data. More dangerously, if the relationships between variables change over time (concept drift), the model's performance can degrade catastrophically.

The California energy grid failure is a perfect example. A model trained on historical data learned correlations between solar output and evening demand under normal conditions. When extreme weather created unprecedented conditions, these correlations changed. Solar output dropped faster than expected, and temperature-driven cooling demand remained higher than historical patterns suggested. The channel-mixing model, optimized for historical correlations, failed.

Channel-Independent Models treat each time series variable independently, forecasting each with its own univariate model. This discards cross-variable information entirely. The obvious drawback is that you lose potentially valuable predictive signals from related series.

The surprising finding from recent research is that channel-independent models often outperform complex channel-mixing models on real-world benchmarks. Because the models never learn inter-variable correlations, they're immune to changes in those correlations. They're simpler, more stable, and less prone to overfitting on spurious patterns.

This means you must carefully evaluate the trade-off. You can start with independent models as a strong baseline. Only move to multivariate approaches when you can demonstrate clear, sustained improvement on rigorous out-of-sample validation. Table 6.1 summarizes the choice.

Aspect	**Channel-mixing (e.g., VAR)**	**Channel-independent**
What it learns	Joint dynamics, including how each series feeds back into the others	Each series in isolation, ignoring inter-series signals
Best when	Cross-series relationships are stable and informative (advertising leads to sales, weather leads to load)	Series are weakly related, or the cross-series structure shifts over time
Strengths	Captures leading indicators and feedback loops; supports Granger causality and other interpretations	Simple, fast, robust; immune to changes in cross-series correlations

Weaknesses	Parameter explosion (10 series × 2 lags ≈ 200 parameters); brittle under regime change	Throws away potentially useful predictive signals from related series
Failure mode	Spurious correlations baked in at training time degrade live forecasts	Misses obvious feedback effects that a multivariate model would catch

Table 6.1:Channel-mixing vs. channel-independent

You can use `statsmodels`, when you have 2-10 series with clear relationships. You need statistical inference (p-values, confidence intervals, causality tests) and interpretability is critical, such as when you are doing academic research or regulatory work requiring formal tests. For anything above that, I'd recommend using other approaches and not using VAR/VARMAX.

Global multi-series models

Before we look at the code, a clarification that often trips readers up. There are two distinct ideas hiding behind the phrase "one model, many series":

- **Cross-learning (global univariate)**. One model is trained on data stacked from every series, so it learns patterns shared across them: weekly seasonality, promotion responses, calendar effects. At prediction time, however, the forecast for Store A still uses only Store A's own history (plus any exogenous variables you

supply). There is no direct path from Store B's lagged sales into Store A's forecast.

- **Cross-series dependencies (multivariate)**. The forecast for Store A explicitly uses lagged values of Store B, Store C, and so on. VAR is the canonical example: every series enters every equation.

`mlforecast`, as we are about to use it, is global univariate. Stacking the data lets a single model share parameters across stores, but it does not, on its own, recreate the cross-series feedback loops VAR captures. If you need those feedback loops inside a global model, you have to engineer them: add columns that hold lagged sales of related stores, regional aggregates, or a learned embedding per series. The library scales beautifully and learns shared general patterns, but treating it as a drop-in replacement for VAR will produce forecasts that quietly ignore the very interactions that motivated the multivariate framing in the first place.

With that distinction in hand, let's demonstrate the global modeling paradigm using `mlforecast` with a tree-based model. This approach scales to thousands of series while naturally handling exogenous variables.

First, we will create the dataset in a *panel* or *long* format. This is the data shape every Nixtla library expects, so it is worth pausing on. Each row is one observation of one series at one timestamp, and three columns carry the load:

- `unique_id`: a label that identifies the series the row belongs to (a store, a SKU, a region, a sensor). Rows with the same `unique_id` form one time series.
- `ds`: the timestamp for that observation.

- `y`: the value to forecast.

If you have ever used a *wide* pivot table with one column per series, panel format is the opposite: instead of widening the table as you add series, you stack them on top of each other and tag each row with its `unique_id`. The advantage is uniform structure, a million series and two series share the same schema, which is exactly what a single global model wants to consume. We will also include an external variable, `is_promotion`, which the model can use to improve its forecasts.

```
import numpy as np
import pandas as pd
from mlforecast import MLForecast
from mlforecast.lag_transforms import RollingMean
from lightgbm import LGBMRegressor
# Create panel data (long format)
# Required columns: 'unique_id', 'ds' (datetime), 'y' (target)
np.random.seed(0)
stores = ['A_store1', 'A_store2', 'B_store1', 'B_store2']
dates = pd.date_range('2023-01-01', periods=30, freq='D')
rows = []
for store in stores:
    base = np.random.randint(80, 220)
    for d in dates:
        is_promo = int(np.random.rand() < 0.2)
        y = (
            base + 5 * np.sin(2 * np.pi * d.dayofweek / 7) +
            15 * is_promo + np.random.normal(0, 3)
        )
        rows.append({
            'unique_id': store, 'ds': d,
            'y': y, 'is_promotion': is_promo})
data = pd.DataFrame(rows)
```

We created a `pandas` DataFrame containing 30 days of synthetic sales data for four distinct entities (`A_store1`, `A_store2`, and so on). The key is

that all series are in one DataFrame, identified by the `unique_id` column. This panel data structure allows a single machine learning model to be trained on the combined data from all series, which can help it learn more general patterns. Thirty days is enough history for the lag features we will configure shortly to be well-defined.

Next, we will initialize the `MLForecast` object. A major advantage of this library is its ability to automatically engineer features from the time series data. We will configure it to create lag features, rolling window statistics, and date-based features. Then, we will train a single LightGBM regression model on the entire dataset.

```
# Initialize MLForecast with automatic feature engineering
model = MLForecast(
    models=LGBMRegressor(n_estimators=100, random_state=42),
    freq='D',
    lags=[1, 7, 14],  # Use values from 1, 7, and 14 days ago
    lag_transforms={
           1:[RollingMean(window_size=7)],  # 7-day rolling mean of
                                            # the 1-day lag
    },
    date_features=['dayofweek', 'month'],
    num_threads=4
)
# Fit ONE model across ALL series
#  static_features=[] tells mlforecast there are no per-series static
# is_promotion will be picked up automatically as a dynamic feature.
model.fit(data, static_features=[])
```

In this step, we defined our forecasting model. `MLForecast` automatically created several new features for the `LGBMRegressor` model to use. For example, `lags=[1, 7, 14]` uses the sales value from `1`, `7`, and `14` days ago as predictors. `date_features` extracts useful calendar information. The `model.fit(data)` command then trained a single

model on this automatically enriched dataset, learning patterns from all four time series at once.

`mlforecast` distinguishes between two kinds of features, and the distinction shapes what you have to supply at prediction time. Static features are constants per series, things like a store cluster, product category, or region, that never change. You declare them with `static_features=[...]` in `fit()`, and the model uses them without needing future values. Dynamic features vary over time (`is_promotion, is_holiday, planned discount`). They are inferred from any panel column that isn't static, and you must provide their future values to `predict()`. The `is_promotion` column in our dataset varies day to day, so it is dynamic.

A point worth restating here: the lags in `lags=[1, 7, 14]` are computed within each `unique_id`. The row for `A_store1` on a given date carries `A_store1`'s own values from `1, 7,` and `14` days back, never `B_store2`'s. The model learns from all four series jointly through shared parameters, but each prediction is still a function of that series' own past plus the exogenous columns. To let `A_store1`'s forecast react to `B_store2`'s recent sales you would have to add cross-series features explicitly, for example a column with the lagged regional total or each store's lagged sales joined onto the panel.

Finally, we will use our trained model to forecast the next seven days. Because our model implicitly learned from the `is_promotion` column, we must provide a DataFrame containing the known or planned values for this variable for the future dates we want to forecast.

```
# Must provide future values for dynamic external variables
future_dates = pd.date_range(dates[-1] + pd.Timedelta(days=1),
                             periods=7, freq='D')
```

```
future_exog = pd.DataFrame({
    'unique_id':  np.repeat(stores, 7),
    'ds':  np.tile(future_dates, len(stores)),
    'is_promotion': [0, 0, 1, 0, 0, 0, 0] *  len(stores)  # Known futu
                                                          # promotions
})
forecasts = model.predict(h=7, X_df=future_exog)
print(forecasts.head())
```

Here, we first created the `future_exog` DataFrame to specify the promotion schedule for the next seven days across all product-store combinations. We then passed this information to the `model.predict()` method. The model used its learned patterns, along with the future promotion data, to generate forecasts for the next `7` days `(h=7)` for every `unique_id`. The result is a single DataFrame containing the predictions for all series.

We now have three lenses on a forecasting system, and it helps to see them as parts of one picture rather than three competing options. A real production stack often uses all three:

- **Multivariate models (VAR, VARMAX, VECM)** capture explicit cross-series feedback. Reach for them when you have a small system, ten or fewer series, where each series is genuinely a target and the dependencies between them carry real predictive value. Good for promotion leading to sales resulting in inventory loops, supply-chain bullwhip dynamics, or jointly forecasting GDP, inflation, and unemployment.
- **Global cross-learning models (mlforecast and friends)** scale to thousands of series by training one model on stacked panel data. They share parameters across series instead of modeling cross-series feedback; cross-series effects only enter if you engineer

features for them. Good for retail SKUs across stores, sensors across factories, or any setting where you have many similar series and limited history per series.

- **Hierarchical reconciliation (covered next)** does not produce forecasts on its own. It takes whatever base forecasts your other models produced and adjusts them so that the lower levels sum correctly to the upper levels, while at the same time often improving accuracy at every level.

The three roles compose naturally. A typical workflow looks like this: train a global mlforecast model to produce base forecasts at the SKU-store level for tens of thousands of series, fit a small VAR or VARMAX on a handful of high-value aggregate categories where promotion dynamics matter, and then run hierarchical reconciliation across the whole tree so that the SKU-store forecasts, the category forecasts, and the chain-wide totals all agree. The choice is rarely *VAR vs mlforecast vs hierarchical*. The choice is which combination earns its keep at each level of the business.

With that picture in mind, hierarchical reconciliation is the next thing to understand.

Hierarchical models

Hierarchical forecasting is the process of forecasting time series that are organized in a hierarchy of aggregation levels, ensuring that forecasts remain coherent across those levels. In other words, you're dealing with data that can be aggregated or disaggregated along dimensions such as geography, product line, or organizational structure.

For example: Total Sales, by Region, by Store. Each level in this hierarchy contains related series. This contrasts with univariate or multivariate forecasting, where each time series may be modeled independently or jointly without regard to such structural relationships. Multivariate models and hierarchical models can be combined. For example, multivariate models can generate base forecasts that are then reconciled hierarchically.

Some common applications include:

- **Retail and consumer goods**:
- Forecasting sales at the product, category, and store levels.
- Ensuring store-level forecasts sum to regional or national totals.
- **Supply chain and inventory**: Forecasting demand by SKU, warehouse, and region to plan procurement and logistics.
- **Finance and planning**: Budgeting and revenue forecasting across departments, subsidiaries, or countries.
- Energy and utilities: Forecasting consumption at the household, city, and grid levels.
- **Public policy**: Forecasting employment, population, or health outcomes at district, state, and national levels.

If you forecast each level of the hierarchy independently, you'll usually get inconsistent results, for example the forecast for total sales might not equal the sum of regional forecasts. Hierarchical forecasting provides reconciliation techniques which adjust forecasts so they are consistent across all levels while preserving as much of the original predictive accuracy as possible.

What are hierarchies?

A hierarchical time series is a collection of series that follow a tree-like aggregation structure. A common retail example illustrates this well.

- **Level 0:** Total Company Sales
- **Level 1:** Sales by Category (e.g., Electronics, Apparel, Groceries)
- **Level 2:** Sales by Product (e.g., TV, Laptop, T-Shirt, Jeans)

In this structure, the sum of sales for all categories must equal the total company sales. Likewise, the sum of sales for all products within the Electronics category must equal the total sales for Electronics.

This structure can be visualized as a tree.

```
Total Sales (Level 0)
    ├── Electronics
    │   ├── TV
    │   └── Laptop
    ├── Apparel
    │   ├── T-Shirt
    │   └── Jeans
    └── Groceries
        ├── Bread                          (Level 2)
        └── Milk                           (Level 2)
```

Energy systems follow similar hierarchical patterns. The electricity grid is another tree:

```
National Grid Interconnection              (Level 0)
    ├── ERCOT (Texas)                      (Level 1)
    │   ├── Austin Energy                  (Level 2)
    │   │   ├── Substation A1              (Level 3)
    │   │   └── Substation A2              (Level 3)
    │   └── CenterPoint                    (Level 2)
    │         └── Substation C1            (Level 3)
```

```
    └── PJM (Mid-Atlantic)                  (Level 1)
        └── PECO                            (Level 2)
            └── Substation P1               (Level 3)
```

Read the levels as: `Level 0` is the national interconnection, `Level 1` the Regional Transmission Organizations (ERCOT, PJM, etc.), `Level 2` the Local Utility Service Areas, and `Level 3` the individual substations at the bottom.

These hierarchies reflect how decisions get made. Finance teams budget at the category level. Logistics plans inventory at the SKU level. Regional grid operators balance supply and demand for their territory. When forecasts at these different levels conflict, teams make decisions based on incompatible numbers.

A related concept is **grouped time series**. In a grouped structure, a series can belong to multiple non-nested groups. For example, a product could be grouped by category (*Electronics*) and by brand (*Brand A*). The forecasting principles are similar. This chapter focuses on the more common hierarchical case.

When forecasting a hierarchy, there is a fundamental constraint. The forecasts must be **coherent**, also known as **aggregation consistent**. This means the forecasts for the lower-level series must sum up to the forecasts for the higher-level series. The forecast for TV sales plus the forecast for Laptop sales must equal the forecast for Electronics sales.

This constraint is routinely violated in practice with model forecasts. When you train independent models for each series, their forecasts almost never add up correctly. A model forecasting Total Sales learns patterns in the aggregate data. A model forecasting TV sales learns

patterns specific to that product. Their errors are independent. The aggregation constraint gets broken.

Consider this concrete example. You forecast:

- **Total Sales**: $1,000,000
- **Electronics**: $450,000
- **Apparel**: $380,000
- **Groceries**: $200,000

The three categories sum to $1,030,000, not $1,000,000, which means that you have incoherent forecasts.

But why does this matter? Because different teams use forecasts from different levels. Let's see an example:

- The CFO sets quarterly budgets using the Total Sales forecast
- The procurement team orders inventory using category-level forecasts
- Store managers plan staffing using product-level forecasts

If these numbers don't align, the procurement team might order inventory worth $1,030,000 while the CFO only budgeted $1,000,000. The mismatch creates operational friction and erodes trust in the forecasting system.

If you train one model for Total Sales, another for Electronics, and a third for TV sales, their forecasts are almost guaranteed *not* to add up correctly. Their errors are independent. The aggregation constraint is violated. We need methods that enforce coherence.

Not every forecasting problem requires hierarchical methods. Use the checklist in Table 6.1 for reference.

Use Hierarchical Forecasting When...	Avoid Hierarchical Forecasting When...
Multiple organizational levels exist.	Only one decision level exists.
Coherence matters for decisions.	Series are completely independent.
You have 3+ hierarchy levels.	Data is too sparse at lower levels.
Series are related but have individual patterns.	Computational resources are extremely limited.

Table 6.2: When to use hierarchical forecasting.

We briefly mentioned reconciliation approaches before, let's look at them in depth now.

Reconciliation approaches

Mathematically, we can formalize the problem we discuss in this section using a summing matrix k. Before the linear algebra, here is the intuition. Imagine you only know the sales of every individual product. To recover Total Sales, you add them all up. To recover **Electronics**, you add up just the products in that category. To recover any individual product, you take its own value as is. The summing matrix k is simply a recipe card: each row is one series in the

hierarchy, and the entries in that row tell you which bottom-level products to add to get it. A 1 means "include this product", a **0** means "skip it." Multiplying `S` by the vector of bottom-level values, Sb_t, just executes every recipe at once.

Let γ_0 be a vector containing all bottom-level series at time t. Let y_t' be a vector containing all series across all levels. The relationship is:

$$y_t = Sb_t$$

The summing matrix S encodes the aggregation rules. For our 3-category, 6-product retail example, the bottom-level vector γ_0 holds (**TV**, **Laptop**, **T-Shirt**, **Jeans**, **Bread**, **Milk**) in that order, and $y_t = Sb_t$ recovers every series in the hierarchy. The full matrix has one row for the **Total**, three for the categories, and six for the bottom-level products themselves (so the bottom rows just copy each product through unchanged):

	TV	**Laptop**	**T-Shirt**	**Jeans**	**Bread**	**Milk**
Total	1	1	1	1	1	1
Electronics	1	1	0	0	0	0
Apparel	0	0	1	1	0	0
Groceries	0	0	0	0	1	1
TV	1	0	0	0	0	0

Laptop	0	1	0	0	0	0
T-Shirt	0	0	1	0	0	0
Jeans	0	0	0	1	0	0
Bread	0	0	0	0	1	0
Milk	0	0	0	0	0	1

Table 6.3: Summing Matrix for the 3-category, 6-product retail example. Rows correspond to series in the hierarchy; columns correspond to bottom-level products

The first row sums all six products to get **Total Sales**. The next three rows sum the products in each category. The last six rows form an identity block: each bottom-level product is just itself. That identity block at the bottom is what makes the matrix dimensions add up to "every series in the hierarchy", aggregates and bottom alike, in one S.

Our goal is to produce reconciled forecasts $\widetilde{y_h}$ for horizon h that satisfy:

$$\tilde{y}_h = S\tilde{b}_h$$

The tilde notation, $\widetilde{y_h}$, denotes reconciled (adjusted, coherent) forecasts. This distinguishes them from:

- Base forecasts (denoted with a hat: $\widehat{y_h}$), the initial, incoherent forecasts from your models
- Historical values (plain notation: y_t'), the observed data

All modern reconciliation methods build on this linear algebraic framework.

Hierarchical forecasting involves two key stages:

1. **Base forecasting**: You first produce forecasts for all series at all levels, often using standard time series models (ARIMA, exponential smoothing, machine learning, and similar). These are sometimes called base forecasts.
2. **Forecast reconciliation**: Then you apply a reconciliation method to make sure the forecasts are coherent (aggregate correctly)

There are four main reconciliation strategies, and we'll work through each in turn: bottom-up, top-down, middle-out, and optimal reconciliation (the modern approach, of which MinT is the workhorse). The first three differ mainly in which level of the hierarchy they treat as the source of truth, then propagate that information up or down. Optimal reconciliation refuses to pick a single source of truth and instead combines information from all levels at once.

The earliest methods for hierarchical forecasting enforce coherence through simple, intuitive rules. They involve making forecasts at one level of the hierarchy and then using aggregation or disaggregation to derive the rest.

These traditional methods represent a forced choice. They compel the forecaster to decide which level of the hierarchy contains the source of truth. This choice determines which information is trusted and which is discarded.

The bottom-up approach

This is the most straightforward approach.

In this approach, we forecast each series at the most detailed level (the "bottom" of the hierarchy). In our example, this means forecasting each individual product. We aggregate these forecasts upwards to get forecasts for all higher levels. For example, the forecast for **Electronics** is the sum of the forecasts for **TVs** and **Laptops**.

Of course this method comes with its pros and cons. The pro is that this method preserves all information at the lowest level. No data is lost to aggregation. It ensures the forecasts are coherent by construction.

The con is that low-level data is often very noisy. The signal-to-noise ratio can be poor. Summing many noisy forecasts can lead to inaccurate aggregate forecasts. This approach also completely ignores the potentially strong and stable patterns that may only be visible at higher levels of aggregation. It trusts the noisy details and discards the stable aggregate signal.

The top-down approach

This method works in the opposite direction of bottom-up. In this approach, we only forecast the top-level series. In our example, this means forecasting **Total Sales.**

We disaggregate this single forecast downwards to produce forecasts for all lower levels. This is typically done using historical proportions. If **Electronics** accounted for 40% of total sales over the last year, its forecast is set to 40% of the total sales forecast.

The pro of this approach is that top-level series is usually the most stable and has the highest signal-to-noise ratio. It is often easier to model accurately.

The con is that this approach assumes that historical proportions will remain constant in the future. This is a strong and often incorrect assumption. It cannot capture individual trends in lower-level series. If a new product is gaining popularity, this method will not reflect its changing share of the total. It trusts the stable aggregate signal but discards all information about the dynamics of the individual components.

The middle-out approach

This is a hybrid method that tries to balance the trade-offs.

We choose an intermediate level of the hierarchy to forecast. For example, forecast at the **Category** level.

We aggregate these forecasts upwards to get the **Total Sales** forecast. We then disaggregate them downwards (using proportions) to get the **Product** level forecasts.

This is a compromise. It attempts to find a middle ground between the noisy bottom level and the overly simplistic top level. However, it still forces a choice about which level's information to prioritize and which to discard. All these methods are built on a one-way flow of information, leading to a loss of potentially valuable signals from other levels.

Modern reconciliation: optimal combinations

The limitations of traditional methods led to a new perspective. Why must we choose one level as the source of truth? Every level of the hierarchy contains some information. The goal should be to combine all of it in a statistically optimal way.

Modern reconciliation techniques operate on a different principle.

1. Generate independent "base" forecasts for *every* series at *every* level of the hierarchy.
2. Recognize that these forecasts are incoherent.
3. Use a mathematical procedure to adjust them, producing a new set of forecasts that are both coherent and more accurate than the original base forecasts.

This approach combines the information. The key is to find the optimal combination.

Coherence here is more than a bookkeeping rule. Enforcing the aggregation constraint filters noise and reduces forecast variance, so what looks like a business requirement also pays a statistical dividend at every level of the tree.

The most prominent modern reconciliation method is **Minimum Trace**, or **MinT**. It treats the reconciliation process as a regression problem. It aims to find a set of coherent forecasts that are as close as possible to the initial, unbiased base forecasts.

The "minimum trace" name comes from its statistical objective: it minimizes the trace of the covariance matrix of the reconciled forecast errors. In simpler terms, it minimizes the total forecast variance across all series in the hierarchy.

The intuition is worth unpacking. The covariance matrix of the reconciled errors is a square table whose diagonal entries are the variances of the forecast errors at each series, Total, Electronics, TV, Laptop, and so on. Its trace is just the sum of those diagonal entries: total forecast uncertainty across the whole hierarchy. MinT searches over all reconciled forecasts that satisfy the aggregation constraints (every coherent forecast can be written as $\tilde{y} = Sb$ for some vector of bottom-level values), and picks the one with the smallest total uncertainty. Because the sum of variances is what you minimize, every level shares in the gain: a noisy region's error tends to get pulled toward the more stable state-level signal, and the more stable level returns the favor by absorbing some of the bottom-up information about local trends. That two-way exchange is what the older bottom-up, top-down, and middle-out methods give up by committing to a single direction of information flow.

MinT learns the best way to combine the base forecasts. The full covariance-based variants (`mint_cov` and `mint_shrink`) do this by estimating the empirical covariance of the historical forecast errors, which lets them recognize, for example, that an over-forecast for TV sales is often accompanied by an under-forecast for Laptop sales, and adjust both. The simpler variants take shortcuts: `ols` assumes the covariance is the identity and ignores empirical error structure entirely; `wls_struct` uses a diagonal weighting that depends only on where each series sits in the hierarchy, not on its observed errors; `wls_var` uses the in-sample error variances on the diagonal but still ignores cross-series error correlations. Each shortcut trades a bit of statistical efficiency for a lot of computational simplicity. We come back to this trade-off below.

The MinT workflow follows a clear sequence.

1. **Generate Base Forecasts.** Use any reliable forecasting model (like ARIMA, ETS, or a machine learning model) to create an independent forecast for every series in the hierarchy. These forecasts will be incoherent.
2. **Choose or estimate the weighting matrix** W. The simplest variant (`ols`) skips this step and uses $W = I$. The structural variant (`wls_struct`) builds W directly from the shape of the hierarchy with no model fitting. The covariance-based variants (`mint_cov`, `mint_shrink`) actually estimate W from the historical in-sample errors of the base models, capturing how forecast errors are correlated across the hierarchy.
3. **Apply Reconciliation Matrix.** A summing matrix, which mathematically defines the hierarchy's structure, is used along with the covariance matrix to compute the MinT reconciliation matrix. This matrix contains the optimal weights for combining the base forecasts.
4. **Produce Coherent Forecasts.** Multiply the vector of base forecasts by this reconciliation matrix. The result is a new set of forecasts that are fully coherent and have, on average, lower error than the original base forecasts.

There are several variations of MinT, differing mainly in how the error covariance matrix is estimated or simplified. The underlying principle, however, remains the same: use the entire set of base forecasts and the structure of the hierarchy to produce optimally combined, coherent predictions.

MinT's statistical efficiency comes at a computational cost. The general implementation requires inverting the matrix $S^T W^{-1} S$ where k is the summing matrix that encodes the hierarchy and W is the weighting matrix discussed above. For a hierarchy with T series, that inversion is $O(n^3)$ in the worst case, but the actual cost depends heavily on the variant. `ols` and `wls_struct` reduce nicely because W is the identity or a simple diagonal, so $S^T W^{-1} S$ stays sparse and structured and the inversion is much cheaper than the worst case. wls_var is similarly cheap. `mint_cov` and `mint_shrink` are the expensive ones: estimating and inverting a dense $n \times n$ covariance is what drives the $O(n^3)$ cost and the matching memory blowup.

This creates a practical barrier. For a retail business with:

- 1,000 stores
- 10,000 SKUs per store
- Total: 10 million bottom-level series

For small hierarchies (under a few hundred series) every MinT variant works fine and the choice is purely statistical. For hierarchies up to a few thousand series, sparse implementations of `ols` and `wls_struct` keep MinT comfortably feasible because the matrices involved stay structured. The trick is to build k as a sparse matrix instead of a dense one. With 50,000 bottom-level series, a dense k already costs roughly 20 GB of memory, while the sparse form holds only the handful of non-zeros per row:

```
from scipy.sparse import csr_matrix
# S_df is the dense summing matrix returned by HierarchicalForecast ut
S_sparse = csr_matrix(S_df.drop(columns='unique_id').values)
# Pass S_sparse to MinTrace; sparse linear algebra avoids the dense bl
reconciler = MinTrace(method='wls_struct')
```

For 10,000-100,000 series, the full-covariance variants become impractical and the standard moves are: stick to `ols` or `wls_struct`, run `mint_shrink` only on the smaller upper levels of the hierarchy, or fall back to bottom-up reconciliation for the deepest sub-trees. Beyond 100,000 series you typically partition the hierarchy (reconcile products within store, then stores within region) or move to neural approaches that learn coherent forecasts directly. Concretely, "distributed computing" here usually means partitioning the tree across workers rather than running MinT on the whole thing at once:

```
# Pseudocode for partitioned reconciliation across many stores
from joblib import Parallel, delayed
def reconcile_one_store(store_id):
    sub_S, sub_tags = build_subtree(store_id) # products within this s
    sub_y_hat = y_hat[y_hat['store'] == store_id]
    return hrec.reconcile(Y_hat_df=sub_y_hat, S_df=sub_S, tags=sub_tag
# Reconcile each store's product subtree in parallel (Spark, Dask, or
store_results = Parallel(n_jobs=-1)(
    delayed(reconcile_one_store)(s) for s in store_ids
)
# Then run a small MinT across the store-level totals to glue subtrees
```

This two-stage pattern (reconcile within each store independently, then reconcile the store totals against the regional and chain-wide aggregates) keeps every individual MinT call to a manageable size and parallelizes naturally across a Spark or Dask cluster.

`sktime` provides a wrapper to add hierarchical reconciliation to its forecasting pipelines. This wrapper works with any forecaster that is compatible with `sktime`. darts offers transformers that perform

hierarchical reconciliation. These operate on the library's own `TimeSeries` objects and work well with its deep learning models.

Nixtla's `HierarchicalForecast` library is built for performance. It can handle thousands of series. For millions of series, the library can be configured to use different methods. You can use bottom-up aggregation for the largest groups. You can also apply more complex reconciliation only at higher levels. Finally, you can partition the problem, for instance, by reconciling products within stores before reconciling across all stores.

Recent research has developed end-to-end neural network methods that learn to produce coherent forecasts directly without a separate reconciliation step. These methods show promise for very large hierarchies. However, they require significant expertise to implement and tune. For most practical applications, start with statistical reconciliation methods. They're proven, well-understood, and widely implemented. We'll discuss neural approaches in *Chapter 7*.

Let's go through a practical example.

First, we need to set up our environment and create the necessary data structures. This involves generating synthetic time series data for several individual products and defining how they are related in a hierarchy. This setup mimics a common business scenario, like products rolling up into categories.

```
import pandas as pd
import numpy as np
# Create sample data: 100 time periods, 4 products
np.random.seed(42)
dates = pd.date_range('2020-01-01', periods=100, freq='D')
# Bottom-level series 4 products, each tagged with its category.
```

```
# Note we use 'item_id' (not 'unique_id')
product_map = {
    'TV': 'Electronics', 'Laptop': 'Electronics',
    'TShirt': 'Apparel', 'Jeans': 'Apparel'
}
rows = []
for product, category in product_map.items():
    base = np.random.randint(50, 150)
    trend = np.linspace(0, 20, 100)
    seasonal = 10 * np.sin(np.linspace(0, 8*np.pi, 100))
    noise = np.random.normal(0, 5, 100)
    sales = base + trend + seasonal + noise

    for d, y in zip(dates, sales):
        rows.append(
            {'item_id': product, 'category': category,
             'ds': d, 'y': float(y)}
        )
df_long = pd.DataFrame(rows)
```

A plain dictionary like `{'Electronics': ['TV', 'Laptop'], ...}` is a fine way to think about the hierarchy on paper, but it is not the format reconciliation libraries consume. The standard move is to attach the hierarchy as categorical columns on the data and let `hierarchicalforecast.utils.aggregate` build the summing matrix and tags for you:

```
from hierarchicalforecast.utils import aggregate
# Levels of the hierarchy, listed from coarsest to finest:
# ['category'] gives the category aggregates; ['category', 'item_id']
# gives the bottom-level products.
spec = [['category'], ['category', 'item_id']]
hier_df, S_df, tags = aggregate(df=df_long, spec=spec)
```

`hier_df` now contains long-format rows for every series in the hierarchy (the implicit total, the categories, and the products), `S_df` is

the summing matrix discussed earlier, and tags is the level to `unique_id` mapping that `HierarchicalReconciliation` expects. The same aggregate helper handles arbitrarily deep hierarchies; you just add more columns to spec.

With our data prepared, the next step is to generate base forecasts for every series in the hierarchy (totals, categories, and products), not just the bottom level. These forecasts will not be coherent on their own; the sum of the product forecasts will rarely match the category forecast they roll up to. Reconciliation will fix that. We use `AutoETS` from `StatsForecast` for the base step.

```
from statsforecast import StatsForecast
from statsforecast.models import AutoETS
# Step 1: Generate base forecasts using ETS
sf = StatsForecast(
    models=[AutoETS(season_length=7)],
    freq='D',
    n_jobs=1
)
base_forecasts = sf.forecast( df=hier_df, h=12)
```

Here, we instantiated the `StatsForecast` object, passing it our DataFrame and specifying the `AutoETS` model, which automatically finds the best Error, Trend, and Seasonality model for each series. The `sf.forecast()` method then efficiently computed predictions for the next 12 days (`h=12`) for each of the four products. The result, `base_forecasts`, is a DataFrame containing these initial, unadjusted predictions.

To better understand the motivation behind modern reconciliation, it's useful to contrast it with traditional approaches. The traditional methods (bottom-up, top-down, and middle-out) for hierarchical

forecasting enforce coherence through simple, intuitive rules. They involve making forecasts at one level of the hierarchy and then using aggregation or disaggregation to derive the rest. These traditional methods represent a forced choice. They compel the forecaster to decide which level of the hierarchy contains the source of truth. This choice determines which information is trusted and which is discarded.

Case study: Forecasting tourism in Australia

This section provides a practical, hands-on application of hierarchical forecasting. We will use a real-world dataset to compare the performance of a traditional bottom-up approach against modern optimal reconciliation.

We will use the Australian domestic tourism dataset (https://www.kaggle.com/datasets/luisblanche/quarterly-tourism-in-australia), a standard benchmark in the hierarchical forecasting literature. This dataset contains the number of overnight trips taken by Australian residents from 1998 to 2016.

The data has a clear, three-level geographical hierarchy.

- **Level 1 (Total):** Total visitor nights for all of Australia.
- **Level 2 (State):** Visitor nights for 7 Australian states and territories.
- **Level 3 (Zone):** Visitor nights for 27 tourism zones within the states.

- **Level 4 (Region):** Visitor nights for 76 tourism regions within the zones.

This structure is ideal for testing hierarchical methods. The sum of region forecasts must equal the zone forecast, which must sum to the state forecast, and so on.

The Nixtla stack ships a ready-made loader for this dataset, so we get three artifacts in one call.



Before running any of this, install the relevant libraries:

```
pip install statsforecast hierarchicalforecast datasets
```

Note that you'd already have this installed with the default environment for the book (see the GitHub companion repository at https://github.com/PacktPublishing/Machine-Learning-for-Time-Series-with-Python-Second-Edition).

The three artifacts are:

- `Y_df`: a long-format DataFrame with `unique_id`, `ds`, and `y columns` covering every series in the hierarchy (total, states, zones, regions), all stacked together.
- `S_df`: the summing matrix that encodes the aggregation structure, with one row per series and one column per bottom-level region.
- `tags`: a dictionary mapping each hierarchy level (Country, State, Zone, Region) to the list of `unique_id` values living at that level.

The reconciliation library uses this to know which rows are bottom level and which are aggregates.

```
import pandas as pd
from statsforecast import StatsForecast
from statsforecast.models import AutoETS
from hierarchicalforecast.core import HierarchicalReconciliation
from hierarchicalforecast.methods import BottomUp, MinTrace
from datasetsforecast.hierarchical import HierarchicalData
# 1. Load the benchmark dataset (Y_df, S_df, and tags arrive together)
Y_df, S_df, tags = HierarchicalData.load('./data', 'TourismSmall')
S_df = S_df.reset_index(names='unique_id')
# Force quarterly timestamps to the start of the quarter and unique_id
Y_df['ds'] = pd.to_datetime(Y_df['ds']).dt.to_period('Q').dt.to_timest
Y_df['unique_id'] = Y_df['unique_id'].astype(str)
# Hold out the last 8 quarters as a test set
test_size = 8
test_df = Y_df.groupby('unique_id').tail(test_size)
train_df = Y_df.drop(test_df.index)
```

The key components, then, are the time series values themselves (`train_df`), the summing matrix (`S_df`) that defines the aggregation relationships, and tags, which identify which level each series belongs to.

We will now walk through a complete hierarchical forecasting project.

1. **Generate base forecasts**: The first step is to generate independent forecasts for every series in the hierarchy, from the total down to the individual regions. We will use **Exponential Smoothing (ETS)** for this task. These “base” forecasts will not be coherent.

```
# AutoETS picks Error/Trend/Seasonality terms automatically
# season_length=4 is for quarterly data
sf = StatsForecast(
    models=[AutoETS(season_length=4)],
    freq='QS',
    n_jobs=1
)
```

Generate base forecasts for the next 8 quarters at every level

```
y_hat = sf.forecast(df=train_df, h=test_size)
```

2. **Implement bottom-up baseline**: The `hierarchicalforecast` library applies any reconciliation method on top of the base forecasts. We pass two reconcilers in one call: a `BottomUp` baseline and MinTrace configured with the `wls_struct` variant of MinT (the variant that uses a diagonal W derived from the hierarchy structure, see the section above on choosing W).

```
reconcilers = [
    BottomUp(),
    MinTrace(method='wls_struct')
]
hrec = HierarchicalReconciliation(reconcilers=reconcilers)
y_rec = hrec.reconcile(Y_hat_df=y_hat, S_df=S_df, tags=tags
```

The reconciled DataFrame `y_rec` contains one column per (model, reconciler) pair, for example `AutoETS/BottomUp` and `AutoETS/MinTrace_method-wls_struct`. Each row's values now satisfy the aggregation constraints exactly: the four region forecasts add up to the zone, the zones to the state, the states to the national total.

A covariance-based variant such as `mint_shrink` needs one extra ingredient: the in-sample fitted values, from which it computes residuals to estimate W. Pass `fitted=True` to `forecast()` so `StatsForecast` caches the fitted values, retrieve them with `forecast_fitted_values()`, and pass the result as `Y_df`; the reconciler does the residual subtraction internally:

```
from statsforecast.reconciliation import MinT
# Re-run forecast with fitted=True to cache the in-sample fitted value
y_hat = sf.forecast(df=train_df, h=test_size, fitted=True)
fitted = sf.forecast_fitted_values()  # columns: unique_id, ds, y, Aut
reconcilers = [MinTrace(method='mint_shrink')]
hrec = HierarchicalReconciliation(reconcilers=reconcilers)
y_rec_shrink = hrec.reconcile(
    Y_hat_df=y_hat, Y_df=fitted, S_df=S_df, tags=tags
)
```

`BottomUp`, `MinTrace('ols')`, and `MinTrace('wls_struct')` ignore `Y_df` because they do not need empirical covariance information; `mint_shrink` and `mint_cov` are the variants that use it.

Evaluating and comparing results

To determine which method is better, we must evaluate their accuracy. It is important to measure performance at *all levels* of the hierarchy. A method might be good at the top level but poor at the bottom. We will use the **Mean Absolute Scaled Error** (**MASE**), a standard metric for comparing forecast accuracy across series with different scales. A lower MASE is better. MinT lowers MASE by reducing forecast variance across the hierarchy. This coherence ensures departments rely on identical figures, aligning inventory orders with corporate budgets.

The results of this comparison are summarized in the table:

Hierarchy Level	Base Forecasts (Incoherent)	Bottom-Up	MinT (Optimal Reconciliation)
Total (Level 1)	0.65	0.88	0.62
State (Level 2)	0.79	0.91	0.75
Zone (Level 3)	1.02	1.05	0.97
Region (Level 4)	1.21	1.21	1.22

Table 6.4: Forecast Accuracy Comparison (MASE) for Australian Tourism

The results are clear and compelling.

- The incoherent base forecasts perform reasonably well but are not usable in practice.
- The Bottom-Up method is significantly *worse* than the base forecasts at the aggregate levels (Total, State, Zone). This demonstrates its primary weakness: aggregating many noisy forecasts from the bottom level can destroy accuracy at the top.
- The MinT method is the clear winner. It produces forecasts that are more accurate than the original base forecasts at every aggregate level. It successfully combines information from across

the hierarchy to reduce error. While its performance at the very bottom level is marginally worse than the base forecasts, its substantial gains at the upper levels make it the superior overall approach.

This case study provides quantitative evidence for the claims made earlier. Optimal reconciliation does more than make the numbers add up and improves forecast accuracy at every aggregate level.

Summary

This chapter equipped you to move beyond univariate forecasting. Where previous chapters treated each time series as an independent entity, you now understand how to model the relationships and hierarchies that define real-world business operations.

You've mastered three distinct approaches to modeling interconnected series. For small systems of two to ten tightly coupled variables, you can implement VAR models and use Granger causality tests to identify genuine leading indicators. For hierarchical business structures where forecasts must aggregate consistently, you can apply MinT reconciliation to ensure that regional predictions sum correctly to corporate totals without sacrificing forecast accuracy. And for large-scale production systems with hundreds or thousands of series, you've learned to use `mlforecast`'s global modeling paradigm, training a single tree-based model that shares parameters across all series while respecting their individual patterns through carefully engineered features.

The techniques matter less than the judgment to pick the right one. You know that VAR scales poorly past ten series because of

parameter explosion. You know that channel-mixing models, useful when correlations are stable, turn brittle under regime changes and concept drift.

You've seen that hierarchical reconciliation, despite its theoretical elegance, requires significant computational resources; the general MinT implementation scales as $O(n^3)$. However, the projection step for the specific OLS method often scales closer to $O(m^3)$ m is the number of bottom-level series. Additionally, global models sacrifice some series-specific nuance in exchange for the ability to learn cross-series patterns that would be invisible to thousands of independent univariate models.

Yet all these techniques share a fundamental limitation. VAR models require you to specify lag orders. Reconciliation methods need you to provide the summing matrix that defines the hierarchy. The `mlforecast` approach depends entirely on the lag features, rolling statistics, and calendar effects you engineer by hand. Your models know about weekly seasonality because you told them to look seven days back. They understand hierarchical structure because you encoded it explicitly.

In *Chapter 7*, we abandon this feature-engineering paradigm entirely. Neural networks learn temporal patterns, cross-series relationships, and even hierarchical structure automatically from raw sequences. LSTMs discover which past time steps matter through their gating mechanisms. Transformers use attention to weigh the importance of every historical point without you specifying a window size. Architectures like N-BEATS perform interpretable decomposition into trend and seasonality internally, mimicking the

classical methods we studied but learning the decomposition from data rather than applying fixed rules.

Further Reading

There are different methods for multivariate forecasting. While they are out of the scope of this chapter, they would make for an interesting read:

- Vector Error Correction Models (VECM): https://www.statsmodels.org/dev/release/version0.9.html#vector-error-correction-model-vecm
- Dynamic Factor Models: https://www.statsmodels.org/dev/statespace.html#dynamic-factor-models
- Multivariate RNNs or Transformers that predict multiple correlated time series: https://arxiv.org/abs/2601.12039
- Best Buy Cancels Some Orders Ahead of Christmas, CNBC, https://www.cnbc.com/2011/12/23/best-buy-cancels-some-orders-ahead-of-christmas.html
- Why California has blackouts: A look at the power grid, ABC News, https://abcnews.com/US/california-blackouts-power-grid/story?id=89460998

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

https://discord.gg/VXpz9YPUTW



7

Practical Deep Learning for Time Series

One of this book's recurring threads is that time-series methods are moving past the classical statistical models that dominated the field for decades. Deep learning used to be a promising-but-peripheral technology, and that is changing fast. Modern deep architectures now drive production forecasting in tasks with many series, many covariates, multivariate dynamics, and long horizons, and in many of these settings they beat classical models outright.

Since the first wave of LSTMs and early Transformers, two trends have shaped the field. The first is the *Transformer* revolution, rebuilt for time series. Direct ports from **Natural Language Processing** (**NLP**) ran into quadratic attention costs and the wrong inductive biases. New ways to tokenize temporal data (patches, inverted axes, 2D reshapes) fixed both problems and made attention competitive for forecasting and classification. The second trend is a resurgence of simplicity. A class of models built entirely on **Multi-Layer Perceptrons** (**MLPs**) and even single linear layers has shown that on many long-horizon benchmarks, complexity is not a prerequisite for state-of-the-art accuracy. DLinear is the emblematic case: a single-layer linear model that beats Transformer stacks on standard benchmarks, and that forced everyone to re-examine what was actually doing the work in the deeper architectures.

In this chapter, you will learn how to implement foundational architectures like LSTMs and TCNs for time series, why simple linear models (DLinear) and MLP-based architectures (N-BEATS, TSMixer) have emerged to challenge more complex models, applications of modern Transformer-based models, from interpretable hybrids (TFT) to patch-based (PatchTST) and channel-dependent (iTransformer) approaches, and finally how to pick your way through the *model zoo* and understand the key architectural trends dominating modern time series research.



Model zoo

The deep learning forecasting ecosystem in Python is split across several mature libraries (`darts`, `neuralforecast`, `sktime`, plus model-specific packages like `crossformer`). Each has its own strengths, and reference implementations of new architectures appear first in different places. The examples in this chapter deliberately rotate through these libraries so you can see each model in its native habitat and learn to read across the ecosystem. Focus on the concepts behind each architecture; the surface API is whatever the maintainers chose this month.

In this chapter, you will learn:

- Why deep learning?
- Foundational models

- MLP-based models
- Transformer models
- 2D and decomposition models
- Model selection

Let's get into it!

Technical requirements

In this chapter, we will work mainly with the `AirPassengersDataset` dataset (https://www.kaggle.com/datasets/rakannimer/air-passengers), and additionally with the `sktime's` solar-generation dataset (loaded via `load_solar`), the ETTh1 electricity-transformer dataset (bundled with `darts`), and the LongHorizon Traffic dataset (loaded via `datasetsforecast`). All of these load from the libraries themselves, so no manual download is required.

Why deep learning?

Deep learning is a subfield of machine learning concerned with algorithms relating to neural networks, a technique that got its name because of the loose association with biological neural networks in the human brain. Nowadays, deep neural networks in the guise of transformers are revolutionizing the field of language generation, be it in commercial products such as ChatGPT or in open-source **Large Language Models** (**LLMs**) like the Qwen series.

But even before these products and models, deep learning had been setting new performance records across many application domains, mostly for unstructured datasets such as text, images, video, and audio. Even though tabular datasets and time-series have so far shown themselves to be less amenable to deep learning, there's also been a large amount of progress. Deep neural networks are now the go-to algorithm for domains such as energy forecasting.

What distinguishes deep learning from classical statistical approaches and traditional machine learning lies in how patterns are represented and learned. Statistical models, such as ARIMA or Exponential Smoothing, rely on explicit assumptions about the structure of the time series, for example linear trends, seasonality, or autocorrelation, and are often interpretable and data-efficient. Machine learning models such as Random Forests or gradient-boosted trees learn patterns automatically but still generally require carefully engineered features, like lagged variables, rolling averages, or external covariates. Deep learning, by contrast, can automatically learn hierarchical and non-linear representations from raw data, often across multiple time series simultaneously. This allows it to capture complex temporal dependencies, long-range interactions, and cross-series patterns that are difficult or impossible to encode manually. However, these advantages typically come at the cost of higher computational requirements, larger data needs, and reduced interpretability, which can limit the benefit of deep learning in simpler or highly structured time-series settings. The following table summarizes the key differences across these three paradigms.

Feature	Statistical	Machine	Deep Learning (DL)

	Methods	**Learning (ML)**	
Examples	ARIMA, ETS, Theta	RF, XGBoost, SVR	RNN, TCN, Transformer
Best for	Short-term, stable, seasonal/linear patterns	Short–medium term, tabular covariates, moderate nonlinearity	Long-term dependencies, complex multivariate patterns
Data requirements	Works well with small datasets	Needs more data than statistical but is still efficient	Often requires large multi-series datasets
Computational cost	Very low	Moderate	High (GPU/TPU training common)
Feature engineering	Domain-driven, highly manual	Important but flexible (lags, holidays, metadata)	Architecture extracts features automatically
Seasonality and trend	Best-in-class for explicit seasonality/trend modeling	Can learn seasonality with engineered features	Needs help (decomposition, embeddings) unless the architecture incorporates seasonal priors
Handling exogenous variables	Limited	Strong	Strong (especially multivariate models)
Interpretability	High (white-box)	Medium (varies by model)	Low (black-box), with exceptions: decomposition models like DLinear expose interpretable trend/seasonal weights
Accuracy	Strong baseline widely used in industry	Good accuracy with strong feature engineering	Can be state-of-the-art when lots of data and complexity justify

Robustness to distribution shift	Good for mild shifts	Mixed	Often brittle without retraining or drift handling
Complexity/maintainability	Simple to deploy	Moderate	Complex models, higher ops burden
Hybridization	Very common (e.g., ETS + ML)	Often backbone for hybrids	Increasingly used as a component in hybrids (e.g., ES-RNN)

Table 7.1: Comparison of statistical models, ML models, and DL models

Rather than viewing one approach as universally superior, it is more useful to treat them as complementary tools, each suited to different data regimes, problem structures, and operational constraints.

In this chapter, we'll chart the course of the field's evolution, moving from early hybrid systems to the current ecosystem of specialized deep learning architectures that now define the cutting edge. In contrast, in *Chapter 9*, we'll talk about large-scale, general-purpose foundation models, which are an extension of the techniques presented here.

Deep learning models, while promising in theory, have struggled to consistently outperform traditional approaches such as ARIMA, Exponential Smoothing or their hybrid counterparts in many standard forecasting benchmarks. In univariate time-series forecasting (especially in benchmark settings such as the *M4 Competition*, which comprised 100,000 time series from diverse domains), pure deep learning or standard machine-learning methods did not decisively outperform classical statistical approaches. For example, among the top 17 methods in M4, the majority were combinations of statistical techniques, and the winning entry was a hybrid architecture combining classical statistical methods with a recurrent neural network (LSTM). Because many of the series lacked rich covariate/feature information, de-seasonalization and embedding of classical components (level/trend/seasonality) helped substantially; pure ML or vanilla neural networks fared poorly. The victor of M4, developed by Slawek Smyl, was not a pure deep learning model but rather a novel hybrid architecture (ES-RNN) that merged Exponential Smoothing with a recurrent neural net, illustrating that at that time deep learning was not yet mature enough to replace the proven principles of statistical forecasting. The ES-RNN used statistical methods for de-seasonalization and normalization, making the time series more digestible for the neural component, which then focused on capturing complex non-linear residual behavior. The key takeaway from M4 is that the state-of-the-art was a blend, where deep learning augmented rather than replaced statistical methods. *Figure 7.1* sketches the recipe so you can adapt it to your own stack.

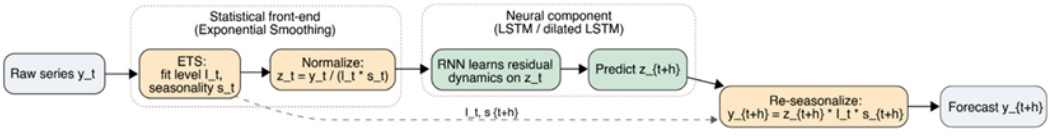


Figure 7.1: ES-RNN-style hybrid template

An ETS front-end estimates level and seasonal components, normalizes the series, and the neural component learns the residual dynamics on the cleaned signal. At forecast time the statistical components are re-applied to put predictions back on the original scale.

You can generalize the same pattern by swapping ETS for STL, Theta, or any decomposition you trust, and swapping the LSTM for whatever neural architecture suits the problem (TCN, Transformer, even a small MLP). The valuable property is that the statistical front-end gives the neural model a stationarized, normalized input it has a chance of learning, instead of forcing it to discover trend and seasonality from scratch.

Two years later, the *M5 Competition* (2020) introduced a fundamental shift by focusing on a massive hierarchical retail sales dataset from Walmart, including 42,840 highly correlated time series along with a rich set of exogenous features such as prices, promotions, holidays, and special events. This transformed the forecasting problem from univariate extrapolation to a large-scale, feature-rich tabular regression problem. The winning solutions were pure machine learning methods, with *LightGBM* emerging as the dominant algorithm. LightGBM's ability to perform global modeling, learning patterns across all series simultaneously, allowed it to generalize promotions, price changes, and seasonal effects across products and stores, outperforming series-by-series methods and statistical or DL hybrids.

The numbers back the claim that this was the dominating algorithm because 7 of the top 10 entries in the M5 Accuracy track were pure LightGBM solutions; the winning entry (*YJ_KAGGLE*) cut the weighted RMSSE by roughly 23% over the competition benchmark, and no statistical-only submission cracked the top 50. The M5 Uncertainty track, which asked for nine quantiles per series, was also won by gradient-boosted trees with quantile loss. M5's lesson is the inverse of M4's: when you have many covariates and correlated series, global gradient boosting wins decisively, and the statistical front-ends that powered M4 hybrids no longer carry their weight.

The M6 Competition (2022–2023) changed the question entirely. Instead of retail demand, M6 ran on 100 traded financial assets (50 US stocks + 50 international ETFs) over 12 four-week submission periods. Participants forecast the probability of each asset landing in each of five quintiles of returns and, in a paired investment track, built portfolios optimizing the information ratio.

The outcomes were sobering. The median forecasting team barely beat the naive uniform benchmark (assign 0.2 probability to every quintile); fewer than half of all teams beat it consistently. In the investment track, most strategies underperformed an equal-weight passive portfolio. The winners across both tracks did not rely on heavy deep learning, instead favoring simple ensembles and judicious feature engineering. M6 reinforced the lesson that domain matters: in noisy, near-efficient financial markets, neither classical statistical methods nor deep learning holds a clear edge, and complex models can actively hurt by overfitting to noise. For an excellent post-mortem, see Makridakis et al., *The M6 forecasting competition* (https://arxiv.org/abs/2310.13357).

As forecasting research has progressed, a new generation of architectures has emerged. For example N-BEATS showed that a purely deep learning architecture (with no explicit time-series-specific components) could achieve compelling performance across univariate benchmarks.

More recently, foundation-model style approaches for time-series, such as TimeGPT, have been proposed: pre-trained on hundreds of billions of data points, these models aim to generalize zero-shot to unseen series and domains.

Recent surveys caution that even these advanced models face serious challenges: distribution shift, domain heterogeneity, over-complexity, and the fact that simpler linear layers sometimes outperform full Transformer stacks in certain forecasting tasks. Thus, rather than asking whether deep learning *beats* classical methods, many practitioners now ask when and how deep learning adds value, what data/feature contexts justify it, and how to build hybrids that use the strengths of both worlds.

We will start by revisiting LSTMs and **Temporal Convolutional Networks** (**TCNs**) to set up the building blocks of sequence modeling, then turn to the modern Transformer family and the architectural choices that made attention work for time series. We will follow that with the linear and MLP-based models that have re-set expectations on what "minimum viable" looks like for a forecasting model. Each architecture is paired with a hands-on example. The examples rotate through the main Python libraries in the ecosystem: `sktime` for classification and benchmarking, darts for general-purpose forecasting, and Nixtla's `NeuralForecast` for high-throughput multi-series workloads.

Finally, we will distill these concepts into a practical model selection framework, so you can weigh the architectural trade-offs and pick the right deep learning tool for the problem in front of you.

Foundational models

We will begin this section with the foundational models that dominated sequence modeling for years. You have likely heard of LSTMs; they were the standard for a long time and even powered the winning M4 model.

While these *old guard* models are now often outperformed by newer architectures, they are worth understanding. They introduce the core concepts, like handling sequential memory (LSTMs) and parallel processing (TCNs), that every other model in this chapter builds upon or reacts against.

Long short-term memory

LSTMs (*Long Short-Term Memory* by Sepp Hochreiter and Jürgen Schmidhuber, 1997) are a type of **Recurrent Neural Networks** (**RNNs**) which were the standard for sequence modeling for many years because they were created to solve the vanishing gradient problem in simple RNNs. This allows them to learn dependencies over long periods.

The core idea of an LSTM is its cell state. The LSTM can add or remove information from this cell state using three structures called **gates**. It has three primary gates:

- **Forget gate**: This gate looks at the new input and the previous hidden state to decide what information (if any) to discard from the cell state
- **Input gate**: This gate decides what new information to store in the cell state
- **Output gate**: This gate decides what information from the cell state to output as the next hidden state

Figure 7.2 shows how these pieces fit together inside a single LSTM cell. Read it left-to-right: the previous hidden state and current input feed all three gates; the forget gate scales the previous cell state down (deciding what to keep), the input gate adds new candidate information, and the output gate decides how much of the resulting cell state is exposed as the new hidden state.

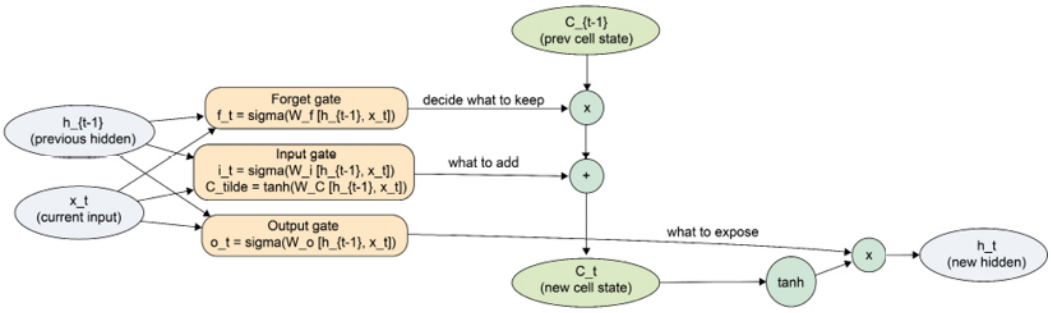


Figure 7.2: Inside an LSTM cell

The cell state **C** runs along the top as a horizontal pipe. The forget, input, and output gates each combine the previous hidden state and current input through sigmoid layers; pointwise multiplications and additions (the **+** and **x** signs) update the cell state and produce the new hidden state.

This gated mechanism makes LSTMs good at modeling stateful relationships. Their main weakness is that they are sequential: they must process the sequence one time step at a time, which makes them difficult to parallelize on modern hardware like GPUs and leads to long training times.

The `darts` library provides a simple wrapper for LSTMs (and other RNNs) built on PyTorch. In this example, we train an LSTM on the classic `AirPassengers` dataset.

```
import pandas as pd
from darts.models import RNNModel
from darts.datasets import AirPassengersDataset
# 1. Load and prepare data
series = AirPassengersDataset().load().astype("float32")
train, val = series.split_before(pd.Timestamp(19580101))
# 2. Define the LSTM model
#    We specify the model_type as LSTM
model_lstm = RNNModel(
    model=LSTM,
    hidden_dim=20,       # Number of units in the LSTM layer
    n_rnn_layers=1,      # Number of LSTM layers
    input_chunk_length=12, # How many time steps to look back
    training_length=16,  # Total length of a training sample
    n_epochs=100,
    model_name=lstm_air,
    force_reset=True,
    random_state=42,
    pl_trainer_kwargs={accelerator: "cpu"}  # swap to "gpu" if available
)
# 3. Train the model
model_lstm.fit(train, val_series=val, verbose=True)
# 4. Make a forecast
pred = model_lstm.predict(n=36) # Forecast 36 months into the future
pred.plot()
```

Figure 7.3 shows the result.

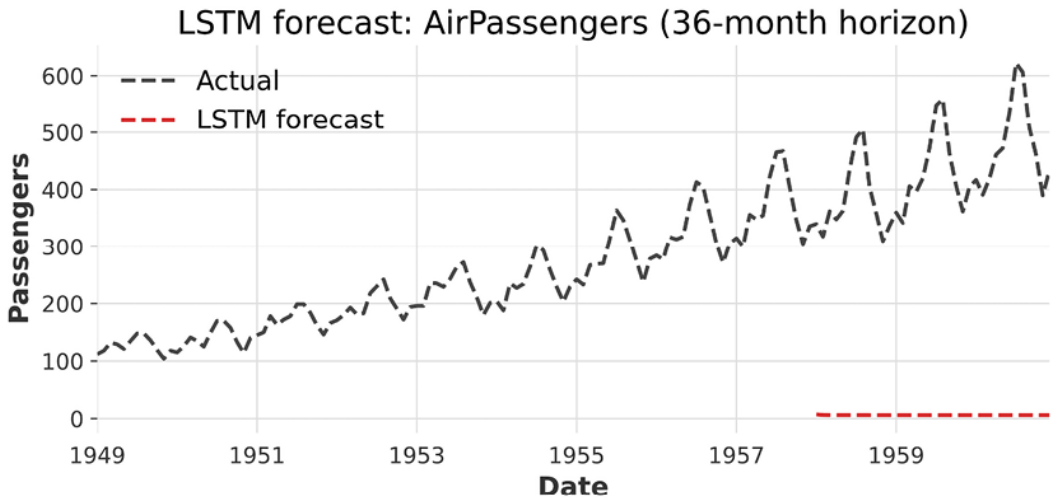


Figure 7.3: LSTM forecast on AirPassengers

The line in the middle of the plot is the historical series (*1949-1960*); the line at the bottom right is the LSTM's 36-month forecast extending past the validation cut-off. The model captures the upward trend and the annual seasonality but underestimates the amplitude of the seasonal swings, which is typical of a small-capacity LSTM with limited training data.

While LSTMs improve on traditional RNNs, their sequential nature limits scalability. This motivates the need for architectures that can better leverage parallel computation.

Gated Recurrent Units (**GRUs**) offer a simpler alternative to LSTMs by using update and reset gates to regulate the flow of information through the network. Although GRUs retain the sequential structure of recurrent models, their reduced complexity and fewer parameters can improve computational efficiency while maintaining effective modeling of temporal dependencies.

One such alternative is **Temporal Convolutional Networks** (**TCNs**).

Temporal Convolutional Networks

TCNs are an alternative to RNNs that apply convolutional architectures to sequence data.

Before explaining TCNs, we must answer two very important questions:

- What is a **convolution**? A convolution uses a small filter (or kernel) that slides across the input data. At each position, it looks at a small, local patch of the sequence. This is good at finding specific local patterns (e.g., a 3-day spike).
- What is a **receptive field**? The receptive field is the total span of input data that the model uses to calculate a single output value. In a simple one-layer model, the receptive field is just the size of its filter. To *see* further back, you must stack many layers.

TCNs are built on two key principles:

- **Causal convolutions**: This is a strict rule that ensures the model cannot *see* into the future. The output at any time step t can only be convolved with inputs from time t and earlier. This is necessary for valid forecasting.
- **Dilated convolutions**: This technique allows the TCN to have a very large receptive field (how far back in the past it can see) without needing a huge number of layers. A dilated convolution applies its filter by skipping input values with a set step. By stacking layers with exponentially increasing dilation factors (e.g., 1, 2, 4, 8...), the network can efficiently capture very long-range dependencies.

A major advantage of TCNs is that their convolutional operations can be performed in parallel. This makes them much faster to train than LSTMs and they often serve as a very strong baseline.

To understand how this works in practice, we will reuse the same `AirPassengers` dataset from the previous section. This allows us to directly compare the behavior of a TCN with the LSTM model we just trained.

In this example, we define a TCN model using the `darts` library, same as before. Pay particular attention to the `input_chunk_length`, which determines how much historical context the model sees, and the `dilation_base`, which controls how the receptive field expands across time.

```
import pandas as pd
from darts.models import TCNModel
from darts.datasets import AirPassengersDataset
# 1. Load and prepare data (same as before)
series = AirPassengersDataset().load().astype("float32")
train, val = series.split_before(pd.Timestamp(19580101))
# 2. Define the TCN model
#    The input_chunk_length is crucial; it must be long
#    enough for the dilations to cover the desired history.
model_tcn = TCNModel(
    input_chunk_length=24, # Lookback window
    output_chunk_length=12, # Forecast horizon
    n_epochs=100,
    kernel_size=3,
    num_filters=6,
    dilation_base=2, # Dilation factors will be 1, 2, 4, 8...
    model_name=tcn_air,
    force_reset=True,
    random_state=42,
    pl_trainer_kwargs={accelerator: "cpu"}  # swap to "gpu" if available
)
# 3. Train the model
model_tcn.fit(train, val_series=val, verbose=True)
# 4. Make a forecast
#    Note: TCN is a direct forecaster, so it predicts
#    the output_chunk_length (12 steps) at once and is rolled forward to reach n=36.
pred = model_tcn.predict(n=36)pred.plot()
```

Figure 7.4 shows the result.

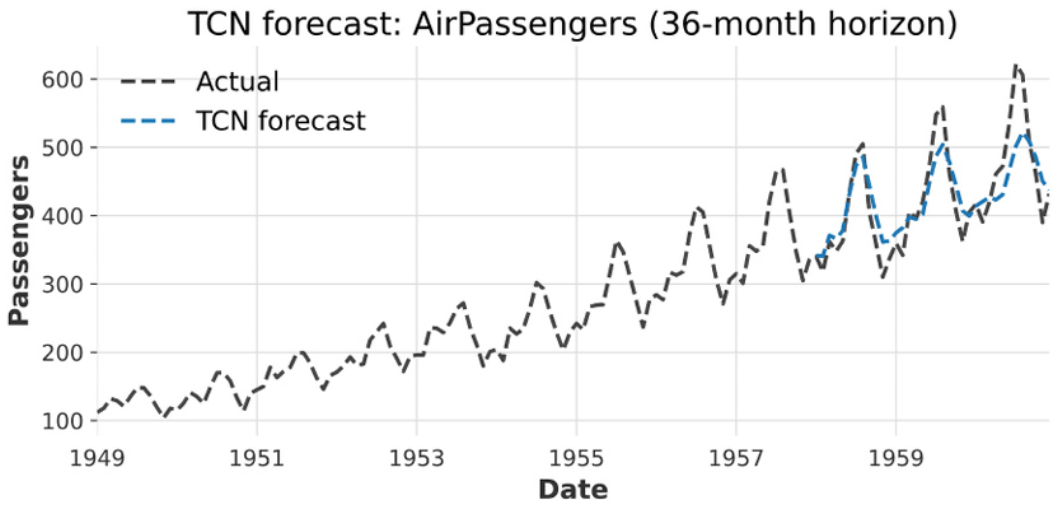


Figure 7.4: TCN forecast on AirPassengers

Same data and horizon as *Figure 7.3* so the two architectures can be compared directly. The TCN's dilated convolutions give it a wider effective receptive field for the same number of layers, and the forecast tracks the seasonal amplitude more closely than the LSTM does, with the trend slightly steeper.

After seeing the complex gates of an LSTM and the dilated convolutions of a TCN, it is natural to assume the path to better forecasts requires more complexity. For years, the field was in an arms race to build deeper, more intricate models.

Then, researchers asked a simple, almost heretical question: *Do we actually need all this complexity?* In my experience, this is one of the most important and practical trends for any practitioner. These models are fast and effective. Treat them as your new default baseline. Before you ever try a complex Transformer, you should start here.

Linear and MLP-based models

Forecasting research is a field often characterized by a relentless pursuit of architectural complexity. But lately, a surprising wave of simple models that can hold their own has been gaining ground.

Two approaches are the heart of this change: pure linear models, which are exactly what they sound like (a single linear layer mapping look-back window to forecast), and **Multi-Layer Perceptron (MLP)** models, which stack several fully connected layers with non-linear activations.

The trend for simplicity was largely catalyzed by the 2022 paper *Are Transformers Effective for Time Series Forecasting?*, which demonstrated that a one-layer linear model (DLinear) could outperform many Transformer-based architectures on standard long-term forecasting benchmarks. That result reopened the door for deeper, but still attention-free, MLP architectures, of which N-BEATS and TSMixer are the most widely cited examples.

This success demonstrates the importance of **inductive bias** (the set of assumptions a model makes to generalize from training data to unseen data). A generic Transformer, designed for NLP, has a weak

and often mismatched bias for time series; it has to learn fundamental concepts like trend and seasonality from scratch. In contrast, DLinear has a strong and correct bias—that many time series can be decomposed into trend and seasonal components—built directly into its architecture, while N-BEATS hard-codes a basis-expansion view of the same decomposition.

Why does this work so often in practice? Four reasons specific to the time-series setting:

- The signal is mostly trend and seasonality. On the standard long-horizon benchmarks (ETT, Electricity, Weather, Traffic, ILI), most of the predictable variance is captured by a smooth trend plus one or two seasonal cycles. A linear projection from the look-back window to the forecast horizon can express both perfectly. The complex, non-linear interactions a Transformer was built to model are often noise in disguise; learning them just hurts.
- Useful temporal autocorrelation decays fast. Long-range cross-time interactions, the thing self-attention is purpose-built for, rarely carry much marginal predictive information once short-range autocorrelation and seasonality are accounted for. A model that ignores everything beyond, say, two seasonal cycles is not throwing away much signal.
- A single linear layer mapping `look_back` to horizon has very few parameters compared to a multi-block Transformer. With limited training data, the linear model converges to a stable estimate of the trend/seasonal mapping; the Transformer overfits, especially on the long horizons that exaggerate any inductive-bias mismatch.
- When the test distribution drifts (level shifts, regime changes, COVID-style breaks), simpler models degrade more gracefully. A linear projection cannot invent confident, wildly wrong non-linearities; a deep model often does.

These are general tendencies that are certainly not written in stone. When the series has exogenous covariates, strong cross-series correlations, or genuinely non-linear dynamics (e.g., demand spikes triggered by promotion combinations), deeper architectures do earn their keep. The practical implication is the same one we will keep coming back to: start with a linear baseline, then prove a more complex model is worth its cost.

DLinear

The DLinear model is an example of the *less is more* philosophy in time-series forecasting. It belongs to the LTSF-Linear family, a set of models designed for long-term forecasting that use a single linear layer mapping look-back window to forecast horizon, with no hidden layers and no non-linearities. Traditional Transformers use self-attention, a mechanism that compares all points in a sequence to determine which are most relevant. While self-attention works well for text and language tasks, it is permutation-invariant, meaning it does not inherently respect the order of data. This can be a disadvantage for ordered time series, where the sequence of events is critical for making accurate predictions.

Standard Transformers patch this with positional encoding (vectors added to each token to mark its position), but DLinear sidesteps the issue entirely. Instead of relying on attention, DLinear uses classical decomposition: it splits the time series into trend and residual components, models each with

a simple linear approach, and then combines them to make forecasts. By focusing on the structured parts of the data, DLinear reduces model complexity while improving performance.

DLinear's architecture is simple, consisting of two main stages: decomposition and linear modeling.

1. **Decomposition**: The model first decomposes the input time series into two distinct components: a trend component and a seasonal (or remainder) component. This is achieved using a simple moving average kernel applied to the input series. The smoothed output of the moving average constitutes the trend component, while the seasonal component is obtained by subtracting the trend from the original series. As we've seen in *Chapters 2* and *3*, this step is a classic statistical technique that simplifies the forecasting problem by separating the long-term trajectory from the periodic fluctuations.
2. **Linear layers**: Following decomposition, two separate single linear layers are applied (no hidden layers, no activations: literally one weight matrix each). One maps the trend component from look-back window to forecast horizon, the other does the same for the seasonal component. The final prediction is the sum of the two layer outputs. Because each layer is a single matrix, the trained weights are directly inspectable: you can plot the trend weights to see which past time steps contribute most to each future step, which is the property that earns DLinear an interpretability footnote in *Table 7.1*.

The effectiveness of this architecture stems from isolating trend and seasonality, avoiding that the model only needs to learn simple linear mappings for each, making it tolerant of noise, computationally efficient, and less prone to overfitting.

DLinear's combination of efficiency and accuracy makes it a strong baseline to consider at the start of any forecasting project. It trains orders of magnitude faster than Transformer models, which makes it useful for rapid iteration and for setting a benchmark that more complex models must justify surpassing. The model is well-suited for domains where trend and seasonality are dominant drivers, including economics, energy consumption, and weather prediction.

The `nixtla` ecosystem provides a convenient and user-friendly implementation of DLinear via the `NeuralForecast` library. This tutorial walks through how to train a DLinear model on a standard time series dataset and generate future forecasts.

Make sure you have the `neuralforecast` library installed. You can check the book's companion repository for detailed and up-to-date information.

We'll use the classic `AirPassengers` dataset, pre-formatted for `NeuralForecast`.

```
import pandas as pd
from neuralforecast.utils import AirPassengersPanel
Y_df = AirPassengersPanel
# Inspect the first few rows
Y_df.head()
```

That prints:

```
  unique_id         ds      y  trend  y_[lag12]
0  Airline1 1949-01-31  112.0      0       112.0
```

```
1  Airline1    1949-02-28     118.0        1         118.0
2  Airline1    1949-03-31     132.0        2         132.0
3  Airline1    1949-04-30     129.0        3         129.0
4  Airline1    1949-05-31     121.0        4         121.0
```

This dataset is monthly airline passenger counts and is a standard benchmark for forecasting. The columns follow NeuralForecast's standard panel (*long*) layout. Every model in the library expects this shape, so it's worth understanding the contract:



Stabilizing variance

Because DLinear relies on simple linear mappings for its separated trend and seasonal components, it expects additive data. When working with series like the passenger counts, where seasonal amplitude expands over time, practitioners should apply a natural logarithm transformation prior to fitting the model. You must invert this transformation on the output to return forecasts to their original scale.

The DataFrame must include:

- `unique_id`: an identifier for the series. The panel here only has one series, `Airline1`, but the same column would carry hundreds of IDs for a multi-series dataset (e.g., one per store or sensor).
- Ds: the timestamp column (datetime).
- Y: the target variable (values to forecast).
- `trend` and `y_[lag12]` are extra columns that ship with this particular dataset; NeuralForecast will ignore them unless you pass them as `hist_exog_list` or `futr_exog_list`.
- If you bring your own data, reshape it into exactly this `unique_id`/`ds`/`y schema` before passing it to `nf.fit`.

Now we'll instantiate a `DLinear` model with the appropriate horizon and look-back window:

```python
from neuralforecast import NeuralForecast
from neuralforecast.models import DLinear
# Forecast horizon (months ahead)
horizon = 12
# Look-back window (months)
input_size = 24
# Instantiate the DLinear model
models = [
    DLinear(
        h=horizon,
        input_size=input_size,
        moving_avg_window=12,
        max_steps=1000,
        val_check_steps=50
    )
]
```

We used the following parameters of `DLinear` in the previous code block:

- H: forecast horizon (how far into the future to predict)
- `input_size`: number of past observations used as input

- `moving_avg_window`: size of the window for the decomposition moving average
- `max_steps`: total number of training steps
- `val_check_steps`: how often to validate during training

Let's train the model now. The `NeuralForecast` class handles the training and prediction pipeline:

```
nf = NeuralForecast(models=models, freq='M')
# Fit the model to the dataset
nf.fit(df=Y_df)
# Generate future forecasts
Y_hat_df = nf.predict()
```

After running this code, `Y_hat_df` contains the predicted values for the next 12 months.

Walking back through the code makes the *simple and classical* claim concrete. The *classical* part is the moving-average step set by `moving_avg_window=12`: a textbook decomposition that splits the series into a smoothed trend and a seasonal residual, the same recipe ETS and STL have used for decades (we covered this in *Chapters 2* and *3*). The *simple* part is the model that sits on top: two single linear layers, one per component, mapping `input_size=24` past observations to `h=12` future ones. There are no hidden layers, no activations, no attention, no recurrence. With `max_steps=1000` the whole thing trains in seconds on a CPU. That is the entire model, and on long-horizon benchmarks it often beats Transformers many times its size, which is exactly the result that re-opened the door for the rest of the architectures in this section.

N-BEATS

DLinear pushes simplicity to its extreme. N-BEATS sits one step deeper on the complexity ladder, while still being a pure MLP architecture, with no recurrence, convolution, or attention. It is one of the most widely cited deep models in forecasting because it was the first pure neural architecture to convincingly beat the M4 statistical benchmarks, and it remains a strong baseline on univariate series with structured trend and seasonality.

The architecture has three load-bearing ideas:

- **Stacked blocks with double residuals**. N-BEATS chains together many fully-connected blocks. Each block reads the look-back window and emits two outputs: a backcast (its best reconstruction of the input) and a forecast (its contribution to the prediction). The block's backcast is subtracted from the input before the next block sees it, so each subsequent block works on the residual the previous ones could not explain. The final forecast is the sum of every block's forecast. This double-residual mechanism is what makes deep N-BEATS stacks trainable without exploding gradients.
- **Basis expansion**. Each block does not predict raw values directly. It predicts a small set of expansion coefficients, which are then multiplied by a fixed basis to produce the backcast and forecast. In the interpretable variant, the bases are hand-crafted: a polynomial basis for trend, and a Fourier basis for seasonality. In the generic variant, the bases are themselves learned linear layers. Interpretable N-BEATS therefore gives you a built-in trend/seasonality decomposition you can inspect, similar in spirit to DLinear but more expressive.

- **Ensembling**. The published results rely on bagging hundreds of models trained with different look-back windows and random seeds. A single N-BEATS model is decent; the published numbers are an ensemble. If you only run one model, do not expect M4-leaderboard performance.

N-BEATS shines on univariate series where trend and seasonality dominate, the typical M-competition setting. It does not natively use exogenous covariates (a successor, N-BEATSx, adds that). For multivariate forecasting with cross-series interactions, reach for TSMixer or a Transformer-family model. N-BEATS's strengths are speed, decomposability of the interpretable variant, and a strong univariate baseline that often matches or beats more complex Transformer architectures.

The darts library wraps N-BEATS as `NBEATSModel`. The following example trains the generic variant on the `AirPassengers` dataset, the same series we used for the LSTM and TCN examples, so you can compare like with like.

```
import pandas as pd
from darts.models import NBEATSModel
from darts.datasets import AirPassengersDataset
from darts.dataprocessing.transformers import Scaler
# 1. Load and split
series = AirPassengersDataset().load().astype("float32")
train, val = series.split_before(pd.Timestamp("19580101"))
# 2. Scale (N-BEATS trains more stably on [0, 1] inputs)
scaler = Scaler()
train_s = scaler.fit_transform(train)
val_s = scaler.transform(val)
# 3. Define a small generic N-BEATS
model_nbeats = NBEATSModel(
    input_chunk_length=24, # 2-year look-back
    output_chunk_length=12, # 1-year horizon
    generic_architecture=True,
    num_stacks=10,
    num_blocks=1,
    num_layers=4,
    layer_widths=128,
    n_epochs=200,
    random_state=42,
)
# 4. Fit
model_nbeats.fit(train_s, val_series=val_s, verbose=False)
# 5. Forecast 36 months and invert the scaling
pred_s = model_nbeats.predict(n=36)
pred = scaler.inverse_transform(pred_s)
pred.plot()
```

Two parameters control the capacity-vs-data trade-off: `num_stacks` (how many residual blocks to chain) and `layer_widths` (how wide the MLPs inside each block are). On small datasets like `AirPassengers`, the values above are deliberately modest; on M4-scale problems, the original paper used 30 stacks of width 512 and ensembled the results. To get the interpretable variant, set `generic_architecture=False`, in which case darts splits the stacks into a trend block (polynomial basis controlled by `trend_polynomial_degree`) and a seasonality block (Fourier basis), and you can read off the decomposition from the trained model.

N-BEATS is a sensible second model to try after DLinear: still fast, still a pure MLP, but with the depth and basis expansion needed to capture more intricate trend-and-seasonality patterns. If it gives a

noticeable lift over DLinear on your data, you have evidence that a deeper architecture is justified before you reach for Transformers.

TSMixer

TSMixer represents another branch of the *simplicity is effective* movement, drawing inspiration from the MLP-Mixer architecture in computer vision. It is an all-MLP model that completely eschews recurrence, convolution, and self-attention, yet has demonstrated state-of-the-art performance, particularly on large-scale multivariate datasets.

TSMixer mixes information across the temporal and feature dimensions using standard MLP blocks. This separation is central to the design of TSMixer. Rather than learning everything at once, the model alternates between two distinct operations, each focusing on a specific type of dependency:

- **Time-mixing**: In this stage, MLPs are applied across the time dimension. Crucially, the same MLP is applied to each feature (channel) independently, with shared weights. This operation allows the model to learn temporal patterns and dependencies (for example trends, seasonality) within each individual time series.
- **Feature-mixing**: In the subsequent stage, MLPs are applied across the feature dimension. Here, the MLP is applied to the vector of features at each time step, again with shared weights across all time steps. This operation enables the model to learn the complex correlations and interdependencies between the different variables (channels) at each point in time.

A TSMixer block consists of a time-mixing MLP followed by a feature-mixing MLP, with layer normalization and residual connections applied throughout. Stacking these blocks builds up representations that integrate both temporal dynamics and cross-variate relationships. This design is also efficient, as it relies only on matrix multiplications, so it is computationally cheaper than attention-based models.

The Darts library makes it easy to train and evaluate deep learning models for time-series forecasting. In this tutorial, we demonstrate how to use TSMixerModel for multivariate long-term forecasting on the `ETTh1` dataset (electricity transformer temperature data).

We'll import the necessary packages:

```
import pandas as pd
import torch
from darts import TimeSeries
from darts.models import TSMixerModel
from darts.datasets import ETTh1Dataset
from darts.dataprocessing.transformers import Scaler
from pytorch_lightning.callbacks import EarlyStopping
```

We begin by loading the `ETTh1` dataset. `ETTh1` is a seven-column multivariate series (HUFL, HULL, MUFL, MULL, LUFL, LULL, OT) recording load and oil-temperature readings from an electricity transformer at hourly resolution. By convention, OT (Oil Temperature) is the forecasting target and the other six are past covariates that may help predict it. Darts represents time series using TimeSeries objects, which simplify handling and preprocessing. For this example, we forecast all seven columns

jointly and pass the same series in as past covariates, so each channel can attend to every other channel's recent history (including its own; passing the target through `past_covariates` is intentional, not a bug). We then split the data into training and validation sets, keeping 70% of the data for training and 30% for validation. Neural networks perform better when the inputs are scaled, so we apply a Scaler to both the target series and the past covariates.

```
series = ETTh1Dataset().load()
target_series = series
past_covariates = series
# Split into training and validation sets (70% training, 30% validation)
train, val = target_series.split_after(0.7)
train_past_cov, val_past_cov = past_covariates.split_after(0.7)
# Scale the data for better training performance
scaler_target = Scaler()
scaler_past_cov = Scaler()
train_scaled = scaler_target.fit_transform(train)
val_scaled = scaler_target.transform(val)
train_past_cov_scaled = scaler_past_cov.fit_transform(train_past_cov)
val_past_cov_scaled = scaler_past_cov.transform(val_past_cov)
```

Next, we define the TSMixer model. We specify the length of the historical window, the forecast horizon, and a few training settings. To prevent overfitting, we use an early stopping callback that monitors the validation loss. The TSMixer model is then instantiated with layer normalization and the specified training callbacks. The historical window determines how many past time steps the model uses for prediction, while the forecast horizon sets how far into the future we aim to predict.

```
horizon = 96             # Forecast horizon (steps ahead)
input_chunk_length = 512 # Historical look-back window
early_stopper = EarlyStopping(
    monitor=val_loss,
    patience=5,
    min_delta=1e-4,
    mode='min',
)
model_tsm = TSMixerModel(
    input_chunk_length=input_chunk_length,
    output_chunk_length=horizon,
    norm_type='LayerNorm',  # Layer normalization in mixer blocks
    pl_trainer_kwargs={callbacks: [early_stopper]}
)
```

We will break down some elements of this code:

- `input_chunk_length`: how many past time steps the model uses as input.
- `output_chunk_length`: how far ahead we want to forecast.
- `norm_type='LayerNorm'`: which normalization to apply inside the mixer blocks. LayerNorm normalizes across features within each time step and is a stability standard in modern deep architectures; the alternative `'BatchNorm'` normalizes across the batch dimension and tends to be less stable for sequence models with variable-length inputs.
- `pl_trainer_kwargs`: a `dict` that gets passed straight through to the underlying `pytorch_lightning.Trainer`. Anything Lightning's trainer accepts goes here: callbacks (we use one for

early stopping), `accelerator='gpu'`, `devices=[0]`, `max_epochs`, `precision='16-mixed'` for mixed-precision training, `gradient_clip_val`, and so on. Treating the underlying Lightning trainer as configurable is a darts pattern you'll see across all of its torch-based models.

With the model defined, we proceed to train it. The `fit` method takes the scaled training series and past covariates and uses the validation series for early stopping. The model learns temporal dependencies across all series while exploiting cross-series relationships captured in the past covariates.

```
model_tsm.fit(
    series=train_scaled,
    past_covariates=train_past_cov_scaled,
    val_series=val_scaled,
    val_past_covariates=val_past_cov_scaled,
    epochs=50,
    verbose=True
)
```



A quick professional's tip is that early stopping prevents the model from overfitting by monitoring the validation loss.

After training, we generate forecasts for the validation set. The predictions are initially in the scaled space, so we apply the inverse transformation to return them to the original values. Finally, we can plot or inspect the forecasts alongside the actual series.

```
pred_scaled = model_tsm.predict(
    n=horizon,
    series=train_scaled,
    past_covariates=train_past_cov_scaled
)
# Inverse transform to original scale
pred_unscaled = scaler_target.inverse_transform(pred_scaled)
We can visualize the forecasts like this:
val.plot(label='Actual')
pred_unscaled.plot(label='Forecast')
```

Figure 7.5 shows the result for the OT (Oil Temperature) channel, the canonical target column in ETTh1.

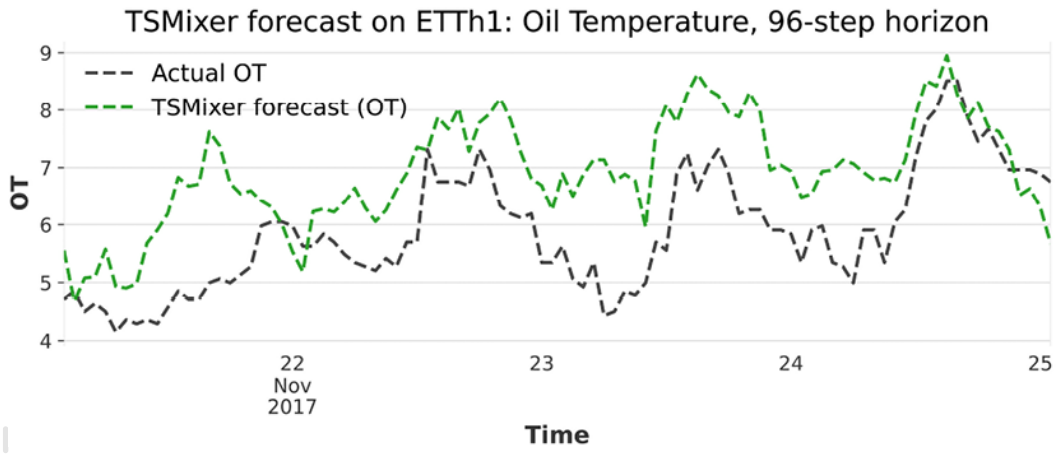


Figure 7.5: TSMixer 96-step forecast on the OT channel of ETTh1

The dark line is the held-out actuals; the green line is the TSMixer forecast generated from the last 512 observations of training data. The model tracks the slow drift of oil temperature well; sharp short-term spikes are smoothed out, which is typical of MLP-based long-horizon models that prioritize the dominant trend over high-frequency noise.

This tutorial illustrates how TSMixer can be applied for long-term multivariate forecasting. By combining historical patterns across multiple series and exploiting past covariates, TSMixer captures complex temporal dependencies efficiently. The use of scaling and early stopping ensures stable training, while the model's architecture makes it well-suited for datasets with long look-back windows and multiple correlated variables.

This tutorial illustrates how TSMixer can be applied for long-term multivariate forecasting. By combining historical patterns across multiple series and leveraging past covariates, TSMixer is able to capture complex temporal dependencies efficiently. The use of scaling and early stopping ensures stable training, while the model's architecture makes it well-suited for datasets with long look-back windows and multiple correlated variables.

Now that we have a fast baseline with DLinear and TSMixer, we can earn our complexity. We now ask: *what can a more complex model do?*

The Transformer architecture, with its self-attention mechanism, is designed to find complex, non-linear relationships between any two points in a sequence, no matter how far apart. This capability is exactly what we need for time series... in theory.

In practice, applying Transformers directly from language processing was a failure. This part explores the modern Transformers that actually work, and we'll group them by the most important practical decision you'll make: whether to treat your time series as independent or related.

Transformer models

While DLinear demonstrated that simpler linear approaches can outperform complex Transformers on many long-term forecasting tasks, the search for Transformer architectures that can efficiently handle long-range dependencies in time series has continued.

Transformers excel at capturing long-range dependencies through their self-attention mechanism, which allows the model to weigh the importance of different past time points when making a prediction.

Directly applying the original Transformer architecture from NLP to time series is challenging because:

- **Quadratic complexity**: Self-attention compares every time step with every other one, so computation and memory usage grow as the square of the sequence length, *O(L2)*. Long time series can therefore become very expensive to process.
- **Permutation-invariance**: Self-attention does not inherently respect the order of inputs. Unlike text, where context can sometimes be flexible, time series are ordered: the sequence of events matters. Standard Transformers address this with positional encoding, sinusoidal or learned vectors added

to each input token to inject ordering information. For time series, positional encoding alone is often not enough, and the architectures discussed below add further temporal inductive biases on top of it.

A first wave of architectures attacked these challenges by hybridizing attention with sequence models that already had the right inductive biases, and by adding components that explicitly handled covariates and interpretability. The most influential of these is the Temporal Fusion Transformer.

Temporal fusion transformer (TFT)

TFT is a hybrid LSTM-Transformer designed around two practical needs that pure attention models did not address: handling heterogeneous covariates (past observed, known future, and static) and producing forecasts you can actually inspect. It became a default choice for industrial forecasting precisely because it slots into the kind of data analysts already work with: panel datasets where each series has metadata (store ID, region, product family), a target with past values, and known-future inputs (calendar, promotions, prices).

The architecture stitches together four pieces:

- **Variable selection networks**. Separate gated networks, one each for static covariates, past inputs, and known-future inputs, learn which variables matter at each time step. The output is a soft attention over input variables, so for any forecast you can read off which covariates the model leaned on. This is the part that gives TFT its reputation for interpretability.
- **Static covariate encoders.** Static features (e.g., store size, product category) are projected into context vectors that condition every other component, including the encoder, decoder, and variable selection networks. This is how a single global TFT model can specialize its behavior per series.
- **LSTM encoder-decoder for local processing.** Past inputs go through an LSTM encoder, future known inputs through an LSTM decoder. The LSTMs handle short-term, order-sensitive patterns, exactly where pure self-attention is weakest. Their outputs become the queries, keys, and values for the next stage.
- **Interpretable multi-head attention for long-range patterns**. A modified multi-head self-attention layer mixes information across all decoder time steps. The *interpretable* modification is that all heads share value projections, so attention weights can be averaged across heads to produce a single, readable map of which past time steps the forecast attended to.

TFT outputs quantile forecasts by default (10th, 50th, 90th percentiles, etc.) trained with a quantile loss. You get prediction intervals out of the box, which is rare among the architectures in this chapter and useful in settings where the cost of being wrong is asymmetric (inventory, capacity planning, energy dispatch). TFT shines when you have known-future inputs and want both accuracy and the ability to explain a forecast to a stakeholder. It is heavier and slower than `PatchTST` or `DLinear`, so on plain univariate series with no covariates it is rarely the right pick.

The `darts` library wraps TFT as TFTModel. The example below trains a small TFT on AirPassengers, using the time index itself as a known-future covariate so the model has something to attend to in the decoder.

```
import pandas as pd
from darts.models import TFTModel
from darts.datasets import AirPassengersDataset
from darts.dataprocessing.transformers import Scaler
from darts.utils.timeseries_generation import(
    datetime_attribute_timeseries)
# 1. Load and split
series = AirPassengersDataset().load().astype("float32")
train, val = series.split_before(pd.Timestamp("19580101"))
# 2. Scale the target
scaler = Scaler()
train_s = scaler.fit_transform(train)
val_s = scaler.transform(val)
# 3. Build a known-future covariate (month-of-year, scaled)
covariates = datetime_attribute_timeseries(
    series, attribute="month", one_hot=False
).astype("float32")
cov_scaler = Scaler()
covariates_s = cov_scaler.fit_transform(covariates)
# 4. Define TFT
model_tft = TFTModel(
    input_chunk_length=24,       # 2-year look-back
    output_chunk_length=12,      # 1-year horizon
    hidden_size=16,
    lstm_layers=1,
    num_attention_heads=4,
    dropout=0.1,
    n_epochs=200,
    add_relative_index=False,    # we provide future_covariates explicitly
    random_state=42,
)
# 5. Fit; future_covariates carries the known-future signal (month index)
model_tft.fit(
    series=train_s,
    future_covariates=covariates_s,
    val_series=val_s,
    val_future_covariates=covariates_s,
    verbose=False,
)
# 6. Forecast 36 months
pred_s = model_tft.predict(n=36, future_covariates=covariates_s)
pred = scaler.inverse_transform(pred_s)
pred.plot()
```

Two parameters dominate the capacity dial: `hidden_size` (the dimension of every internal representation, kept tiny here for speed) and `num_attention_heads` (parallel attention slices). For probabilistic forecasts, pass a likelihood such as `darts.utils.likelihood_models.QuantileRegression(quantiles=[0.1, 0.5, 0.9])`; predict then returns a stochastic TimeSeries you can sample or summarize into intervals.

To inspect the attention weights, call `model_tft.predict(..., num_samples=1)` and use the model's explain utilities; the original paper's interpretability story is one of TFT's main selling points.

If you have a panel dataset with calendar features, holidays, prices, or other known-future inputs and the stakeholder wants to know why a particular forecast moved, start with TFT. If you only have a univariate series, skip ahead to the next subsection: PatchTST is faster and usually as accurate.

PatchTST

PatchTST (**Patch Time Series Transformer**), introduced a new paradigm for processing temporal data. PatchTST builds on two core ideas: patching and channel independence.

- **Patching**: The core innovation of PatchTST is to move away from point-wise attention, where each time step is a token. While, in NLP, a word is a natural token carrying semantic meaning, a single time point in a series has very little semantic value on its own. PatchTST segments time series into overlapping sub-series of a fixed length, called **patches**.

  These patches, which retain local semantic information like trends and seasonal shapes, are then linearly projected to form the input tokens for the Transformer encoder. Concretely, with patch length `P=4` and stride `S=2`, a length-12 series `[x1, x2, ..., x12]` becomes five overlapping patches: `[x1..x4], [x3..x6], [x5..x8], [x7..x10], [x9..x12]`. The Transformer then attends over five patch tokens instead of twelve point tokens, cutting sequence length while each token carries a richer local signal.

- **Channel Independence (CI)**: The model processes each channel through the same Transformer backbone, with shared weights. After forecasting, the predictions for each channel are concatenated to form the final multivariate forecast. This approach is a strategic choice to avoid noise. It prevents the model from being confused by spurious correlations from irrelevant channels and makes the architecture scale to datasets with many variables. A useful mental model: CI is like having one expert who looks at each series in isolation, while **channel-dependent** (**CD**) approaches like iTransformer are more like a team meeting where every series gets to talk to every other one. CI wins when the "other voices" in the room are mostly noise; CD wins when those voices carry real signal (e.g., neighbouring traffic sensors, electricity loads on the same grid).

The combination of these two principles makes PatchTST both efficient and effective. It can capture long-range dependencies by attending over a long history (enabled by patching) without being overwhelmed by noisy cross-channel interactions (due to channel independence).

PatchTST is the go-to model for long-term forecasting of both univariate and multivariate time series. The model has been successfully applied across many domains: energy, weather, traffic, and medical data. It has established itself as a top-performing model for long-term forecasting, outperforming previous Transformer-based models and even strong linear baselines like DLinear on many benchmarks. Its performance is particularly notable on datasets with strong seasonal patterns, such as the Electricity dataset.

The `sktime` library provides a unified, scikit-learn-like interface for time series tasks, including an implementation of PatchTST. In this tutorial, we show how to forecast half-hourly UK solar generation using PatchTST. The dataset, available via `sktime.datasets.load_solar`, contains 5,905 observations and a strong 48-step daily cycle: exactly the kind of long, periodic series PatchTST was built for. PatchTST is hungry for data; running it on a 16-row series like Longley would not give the patching mechanism enough training windows to learn anything meaningful.

`PatchTSTForecaster` ships behind sktime's optional deep-learning extra. Install with:

```
pip install "sktime[deep_learning]"
```

You'll also need transformers (Hugging Face), which is what `sktime` delegates to under the hood. Then we can start by importing the necessary packages:

We'll then start by importing the necessary packages:

```
import os
os.environ["WANDB_DISABLED"] = "true"  # silence wandb prompts
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sktime.forecasting.patch_tst import PatchTSTForecaster
from sktime.datasets import load_solar
```

We then load the solar dataset and reserve the last 48 observations (one full day) for testing. Neural networks tend to perform better when the input is scaled, so we standardize the target with `StandardScaler` and convert the result back into a pandas DataFrame to keep the index and column name.

```
# Load the dataset
y = load_solar().asfreq("30min").to_frame(name="solar")
# Split into training and test sets
horizon = 48
y_train, y_test = y.iloc[:-horizon], y.iloc[-horizon:]
# Scale the target for better model performance
scaler_y = StandardScaler()
y_train_scaled = pd.DataFrame(
    scaler_y.fit_transform(y_train),
    index=y_train.index,
    columns=y_train.columns
)
```

Next, we define the PatchTST model. The current sktime wrapper takes two dictionaries: a config that describes the Transformer (context length, prediction length, layers, attention heads, embedding dimensions, patch length, and stride) and a `training_args` dict passed through to the underlying Hugging Face Trainer. The patching mechanism divides the time series into overlapping segments, turning a long sequence of points into a shorter sequence of richer tokens. The context length sets how many past observations the model attends to for each forecast.

```
# Define the model configuration
config = {
    "context_length": 192,        # Look-back window
                                  # (4 days of half-hourly obs)
    "prediction_length": horizon, # Forecast horizon (1 day)
    "patch_length": 8,            # Length of each patch (4 hours)
    "patch_stride": 4,            # Stride between patches
    "num_hidden_layers": 2,
    "num_attention_heads": 4,
    "d_model": 32,
    "ffn_dim": 64,
    "activation_function": "gelu", # 'gelu' is the default;
                                   # 'relu' also supported
    "ff_dropout": 0.1,
    "attention_dropout": 0.1,
```

```
    "head_dropout": 0.0,
    "norm_type": "batchnorm",     # or 'layernorm'
    "scaling": "std",             # per-window standardization built in
}
training_args = {
    "num_train_epochs": 5,
    "output_dir": "patchtst_results",
    "report_to": "none",
}
forecaster = PatchTSTForecaster(
    config=config,
    training_args=training_args,
)
```

In this code:

- `patch_length` and `patch_stride` implement the patching mechanism, reducing the number of tokens the Transformer must process. With `patch_length=8` and `patch_stride=4`, a 192-step look-back becomes 47 overlapping patch tokens.
- `context_length` is the effective historical window used for each forecast.

We then fit the model. Since we want to predict the next 48 half-hourly steps, we pass a forecasting horizon array from 1 to 48.

```
fh = np.arange(1, horizon + 1)
forecaster.fit(y=y_train_scaled, fh=fh)
```

Once training is complete, we generate forecasts for the held-out day. The predictions come back in the scaled space, so we apply the inverse transformation to return them to the original units (solar generation in MW).

```
y_pred_scaled = forecaster.predict()
y_pred = pd.DataFrame(
    scaler_y.inverse_transform(y_pred_scaled),
    index=y_pred_scaled.index,
    columns=y_pred_scaled.columns
)
print(Forecast head:)
print(y_pred.head())
```

This workflow shows how PatchTST handles a long, periodic series in a clean, intuitive way. By segmenting the input into patches and treating each variable independently, the model captures long-range dependencies efficiently. The same pattern extends to multivariate inputs: pass a multi-column DataFrame and PatchTST applies the shared backbone to each channel. For longer horizons or larger multivariate datasets, scale up `context_length`, `d_model`, and `num_hidden_layers`; the companion repository tracks an updated configuration as the API evolves.

Where does PatchTST hurt? Three drawbacks worth knowing before you commit to it on a real project:

- **No native support for exogenous covariates.** PatchTST is built around the target series (or set of channels). If your forecast genuinely depends on holidays, prices, weather forecasts, or other

known-future inputs, you have to bolt them on as extra channels and hope cross-attention picks up the relationship, or switch to a model designed for it (TFT, NeuralForecast's `NHITS-x`).

- **Channel independence is a feature and a limitation.** The shared backbone with no cross-channel attention is what makes PatchTST scale to hundreds of series. It's also why the model cannot learn that, say, traffic in one location causes traffic in another. On datasets where that signal matters (Electricity, Traffic, financial baskets), iTransformer or Crossformer often beat PatchTST by a non-trivial margin.
- **Sensitive to patch_length and patch_stride.** These two hyperparameters set both the token semantics and the sequence length the Transformer sees. Too short and you reproduce point-wise attention's noise; too long and you blur over the local patterns that gave PatchTST its edge. The defaults from the paper (`patch_length=16`, `patch_stride=8` on hourly data) are a reasonable starting point, but expect to tune them per dataset.

iTransformer

While PatchTST demonstrated that ignoring cross-channel correlations works well for certain datasets, the iTransformer takes the opposite approach. It is designed around the idea that in many multivariate systems, the correlations between series are not noise but the primary source of predictive information. By prioritizing cross-variate relationships, iTransformer explicitly models the interactions between variables, making it well-suited to high-dimensional datasets.

The core innovation of iTransformer lies in its inverted architecture, which reinterprets how the standard Transformer processes time series data. Rather than tokenizing the time dimension into individual steps or patches, iTransformer treats each entire time series for a single variate as a token. Concretely, if your data has shape (`lookback_window, num_variates`), it is transposed to (`num_variates, lookback_window`) before being fed into the Transformer. Each row then becomes a variate token, embedding the full temporal information for that variable.

Once tokenized this way, the Transformer's components are applied differently than in a standard Transformer. The multi-head self-attention mechanism now operates across variates, allowing the model to explicitly learn correlations between every pair of variables over the entire look-back window. Meanwhile, the **Feed-Forward Network** (**FFN**) is applied along the temporal dimension within each variate token, enabling the model to capture complex, non-linear temporal patterns individually for each series. This structure embodies a pure channel-dependent strategy, directly addressing the challenge of modeling multivariate dependencies—a known limitation of channel-independent architectures like PatchTST.

iTransformer has achieved great results on high-dimensional multivariate forecasting benchmarks, including the Traffic dataset with 862 series and the Electricity dataset with 321 series. A key advantage of the architecture is that performance generally improves with longer look-back windows, as more historical context allows self-attention to better capture cross-variate patterns.

The model is particularly suited for applications where strong correlations exist between series. Typical use cases include traffic flow prediction, electricity consumption forecasting, and financial market

analysis, where the predictive signal lies in the interactions between multiple variables rather than solely in individual temporal patterns.

The `NeuralForecast` library provides a convenient implementation of iTransformer, allowing you to efficiently train it on large multivariate datasets.

We'll need an extra package for the dataset loader. Install with:

```
pip install datasetsforecast
```

Then we'll import the necessary libraries:

```
import pandas as pd
from neuralforecast import NeuralForecast
from neuralforecast.models import iTransformer
from datasetsforecast.long_horizon import LongHorizon
```

We begin by loading a multivariate dataset. For this tutorial, we use the `Traffic` dataset, which contains 862 separate time series of hourly road-occupancy rates. `LongHorizon.load` will download the dataset on first call and cache it under a directory; subsequent calls reuse the cache. If you cannot download (e.g., offline or firewalled), substitute any wide multivariate CSV reshaped into long format (one row per (`unique_id, ds`) pair).

```
Y_df, _, _ = LongHorizon.load(directory='./data', group='Traffic')
Y_df['ds'] = pd.to_datetime(Y_df['ds'])
# Split into training and testing sets
train_df = Y_df[Y_df['ds'] < '2008-06-15']
test_df = Y_df[Y_df['ds'] >= '2008-06-15']
```

`Y_df` arrives in NeuralForecast's standard panel layout (the long format we introduced for DLinear): `unique_id` identifies each series, `ds` is the timestamp, `y` is the value. So `Y_df['unique_id'].nunique()` counts how many distinct series are in the panel: 862 for the full Traffic dataset. `iTransformer` treats each of these series as one token in the attention mechanism, so the model needs to know the count up front.

Next, we define the model parameters, including the forecast horizon, look-back window, and the number of series.

```
horizon = 96          # Forecast horizon (hours ahead)
input_size = 336      # Historical window length
num_series = Y_df['unique_id'].nunique()
models = [
    iTransformer(
        h=horizon,
        input_size=input_size,
        n_series=num_series,
        d_model=64,          # Embedding dimension
        n_heads=4,
        n_layers=3,
        d_ff=256,
        dropout=0.2,
        random_state=42
```

```
    )
]
```

We then create a `NeuralForecast` object, which handles the multivariate training process. The library trains a single global model across all series, sharing patterns and interactions across them.

```
nf = NeuralForecast(models=models, freq='H')
# Fit the model on the training data
nf.fit(df=train_df)
```

After training, we generate forecasts for the test period. The predictions are returned in a `pandas` DataFrame, ready for analysis or visualization.

```
# Generate predictions
Y_hat_df = nf.predict()
# Inspect the first few forecasted values
print(Y_hat_df.head())
```

This workflow illustrates how iTransformer captures cross-series dependencies in high-dimensional datasets. By inverting the Transformer architecture and treating each variate as a token, it directly models interactions between variables while maintaining the ability to learn complex temporal patterns within each series. This makes it a strong choice for multivariate forecasting where cross-variate correlations carry the predictive signal.

Crossformer

Crossformer offers a middle ground between channel-independent and channel-dependent architectures. While models like PatchTST ignore cross-channel correlations and iTransformer fully prioritizes them, Crossformer is designed to explicitly capture both temporal patterns and inter-variable dependencies in a structured, hierarchical manner. Its core innovation is the **Two-Stage Attention** (**TSA**) layer, which disentangles temporal and cross-dimensional correlations while staying computationally tractable.

Crossformer begins with a **Dimension-Segment-Wise** (**DSW**) embedding, which segments the time series into short patches similar to PatchTST. Instead of treating these patches purely along the time axis, it organizes them into a 2D array of tokens with shape (`num_segments`, `num_variates`). This design preserves both local temporal patterns within each segment and the alignment of variables across time.

The heart of Crossformer is the **Two-Stage Attention** (**TSA**) layer. In the first stage, called the cross-time stage, multi-head self-attention is applied along the temporal dimension for each variate independently. This allows the model to capture temporal dependencies within each series. In the second stage, the cross-dimension stage, attention is applied across the variate axis for each segment. To make this tractable for high-dimensional datasets, Crossformer introduces a small set of router tokens that aggregate information from all variables and broadcast it back, effectively approximating full all-to-all attention with linear complexity instead of quadratic.

Multiple TSA layers are stacked in a hierarchical encoder-decoder structure. The encoder merges adjacent segments at each layer to create coarser temporal representations, while the decoder generates forecasts at each scale. These outputs are then combined to produce the final prediction, allowing the model to capture both long-term trends from coarse segments and fine-grained details from shorter segments.

Crossformer has demonstrated state-of-the-art performance on datasets with strong and dynamic inter-variable correlations, such as Electricity and Traffic. Its hierarchical structure helps long-horizon forecasting in particular: forecast error grows more slowly with horizon than in simpler architectures.

The model is best suited for multivariate forecasting tasks where relationships between series are complex and time-varying. It excels when it is important to model both individual series dynamics and their interactions in a structured way. Crossformer has been applied successfully in domains such as traffic forecasting, energy management, and weather prediction.

The `crossformer` package, built on PyTorch Lightning, provides a ready-to-use implementation of the model. We will now illustrate how to train and use Crossformer with a sample multivariate dataset.

We start by creating a sample multivariate dataset. This is a teaching stub: the data is pure random noise, so the model will not learn anything meaningful from it. The point is to walk through the API surface end-to-end. For real usage, replace `sample_df` with a wide DataFrame whose columns are your variates (one column per channel) and whose rows are evenly-spaced time steps.

```
import pandas as pd
import numpy as np
import torch
num_features = 8
num_timesteps = 400
sample_df = pd.DataFrame(np.random.rand(num_timesteps, num_features))
```

To use real data, the conversion is mechanical. Crossformer's `DataInterface` expects exactly this wide-DataFrame shape (timesteps, features) with no datetime index and no target/covariate distinction. If your data is currently in a long ("`panel`") format like `NeuralForecast` uses, pivot it first; if it's a wide DataFrame with a `DatetimeIndex`, drop the index. For example, to use the ETTh1 dataset:

```
# Long to wide pivot (if your data is in long format)
# wide_df = panel_df.pivot(index='ds', columns='unique_id', values='y')
# Wide DataFrame with datetime index → reset the index
from darts.datasets import ETTh1Dataset
ett = ETTh1Dataset().load().pd_dataframe() # wide DataFrame
                                           # indexed by timestamp
sample_df = ett.reset_index(drop=True)     # Crossformer expects
                                           # a plain RangeIndex
num_features = sample_df.shape[1]          # 7 channels for ETTh1
num_timesteps = len(sample_df)             # ~17,420 hourly observations
```

Now `sample_df` carries real signal and the same downstream code (`config`, `DataInterface`, `CrossFormer`, `Trainer`) trains a useful model.

Next, we define the model configuration, specifying the number of features, the look-back window length (`in_len`), forecast horizon (`out_len`), segment length, and Transformer architecture parameters

such as the number of layers, attention heads, embedding dimensions, and dropout. We also specify training parameters for PyTorch Lightning:

```
cfg = {
    data_dim: num_features,
    in_len: 96,           # Look-back window
    out_len: 96,          # Forecast horizon
    seg_len: 6,           # Length of each segment/patch
    d_model: 64,          # Embedding dimension
    n_heads: 4,           # Number of attention heads
    e_layers: 2,          # Encoder layers
    d_layers: 1,          # Decoder layers
    factor: 10,           # Router factor for cross-dimension attention
    dropout: 0.1,
    batch_size: 32,
    accelerator: cpu,     # or gpu
    precision: 32,
    max_epochs: 10
}
```

The `DataInterface` class handles batching and data preparation for the model:

```
from crossformer.data_tools import DataInterface
dm = DataInterface(sample_df, **cfg)
```

We then initialize the Crossformer model with the configuration and create a PyTorch Lightning Trainer to handle the training loop:

```
from crossformer.model.crossformer import CrossFormer
from lightning.pytorch import Trainer
model = CrossFormer(**cfg)
trainer = Trainer(
    accelerator=cfg[accelerator],
    precision=cfg[precision],
    max_epochs=cfg[max_epochs],
)
```

Training is performed by calling the trainer's `fit` method:

```
trainer.fit(model, dm)
```

Once the model is trained, we can generate forecasts. For demonstration, we pass a random tensor representing a batch of input sequences through the model. In practice, you would provide real input data:

```
input_tensor = torch.randn(1, cfg['in_len'], cfg['data_dim'])
model.eval()
with torch.no_grad():
    forecast = model(input_tensor)
print(Forecast shape:, forecast.shape)
```

This example shows how Crossformer captures both temporal patterns and cross-dimensional correlations in multivariate datasets. By combining the Dimension-Segment-Wise embedding, Two-Stage Attention, and a hierarchical encoder-decoder, it produces accurate forecasts across multiple variables while remaining computationally efficient.

All the models we have explored so far, from LSTMs to Transformers, share a common assumption: they treat a time series as a 1D sequence. They operate on the data as a one-directional flow of information, whether by using recurrent memory, 1D convolutions, or self-attention.

But what if this 1D view is limiting us? What if the key to capturing complex patterns, especially seasonality, isn't a new 1D architecture but a new data representation?

2D and decomposition models

The shift is motivated by an observation: in time series forecasting, innovation is happening at the data representation layer as much as at the model layer. By reshaping time series into structured 2D tensors, we can borrow proven architectures from computer vision, such as **Convolutional Neural Networks** (**CNNs**), to detect patterns that are difficult for 1D models to learn.

Many real-world time series are multi-scale and periodic, with overlapping daily, weekly, or yearly cycles. TimesNet explicitly encodes these periodic structures in the input, effectively turning a time series into an *image* where repeating patterns are aligned in a way that CNNs can easily process. By embedding this strong inductive bias directly into the input, TimesNet accelerates learning and improves performance.

TimesNet is built around the insight that multi-periodicity, the presence of multiple overlapping seasonal patterns, is common in real-world series. Modeling such complexity in 1D sequences is challenging, so TimesNet transforms the data into a 2D representation and applies 2D CNNs to extract meaningful features.

1. The first step is period discovery. TimesNet analyzes the input series in the frequency domain using a **Fast Fourier Transform** (**FFT**). By examining the power spectrum, the model identifies the dominant frequencies, which correspond to the most important periods in the series. The top-k periods are then selected for further processing.
2. Next, each 1D time series is reshaped into a 2D tensor for each selected period. If a series has length T and a dominant period Y_t, it is reshaped into a tensor of size $p_i f_i$, where $f_i = T / p_i$. (Mathematically, TimesNet can pad the sequence so that its length is divisible by the selected period; this allows the reshaping to produce a complete rectangular tensor.) Each column captures the data points within a single period (intra-period variation), while each row aligns data points across consecutive periods (inter-period trends). To make this concrete, take hourly electricity load over T = 168 hours (one week) with a discovered period pi = 24. The series is reshaped into a 24 × 7 matrix: each column is one full day (intra-day shape: morning peak, afternoon dip, evening spike), each row is the same hour-of-day across all seven days (inter-day trend at, say, 18:00). A 2D convolution kernel that sweeps this matrix can pick up *evenings get progressively higher through the week* along rows and *this is a typical Tuesday shape* along columns at the same time. This

transformation effectively turns temporal patterns into a structured 2D image, making repeating patterns easier to detect.

3. After reshaping, TimesNet applies a parameter-efficient 2D Inception block for feature extraction. The Inception architecture, borrowed from computer vision, can simultaneously capture features at multiple scales. In TimesNet, the 2D kernels of the Inception block process both vertical (intra-period) and horizontal (inter-period) patterns. A single shared block is applied to all k period tensors, keeping the parameter count low.
4. Finally, the feature maps from the top-k periods are aggregated adaptively. The weighted sum uses the amplitudes of the corresponding frequencies to emphasize dominant patterns. This process is encapsulated in a TimesBlock, and multiple TimesBlocks can be stacked to form a deep network, allowing hierarchical feature extraction across periods.

TimesNet has achieved results across many time series tasks, including long- and short-term forecasting, imputation, classification, and anomaly detection. Its multi-period representation makes it particularly effective for series with strong seasonal patterns. TimesNet is versatile enough to serve as a general-purpose backbone for time series analysis, enabling a single model to perform multiple tasks across domains such as electricity load forecasting, weather prediction, financial analysis, and industrial monitoring.

The `NeuralForecast` library provides an easy-to-use implementation of TimesNet, enabling straightforward training and prediction.

We begin by loading a dataset with strong seasonality, such as the `AirPassengersPanel` dataset, which records monthly international airline passengers:

```
from neuralforecast.utils import AirPassengersPanel
Y_df = AirPassengersPanel
Print(Y_df.head())
```

Next, we define the model parameters. The forecast horizon specifies how many future steps we want to predict, and the look-back window determines how many historical time steps the model observes. TimesNet also requires the number of top periods to extract (`top_k`) for the 2D transformation and the number of kernels in the Inception block.

```
from neuralforecast.models import TimesNet
horizon = 12
input_size = 36
models = [
    TimesNet(
        h=horizon,
        input_size=input_size,
        top_k=3,
        num_kernels=16, # Number of Inception kernels
        d_model=64, # Model dimension
        n_layers=2,
        dropout=0.2,
        random_state=42
    )
]
```

In practical terms, `top_k=3` tells TimesNet to run an FFT on the input window, sort the discovered periods by spectral amplitude, and keep the three strongest. For an hourly retail series you would typically see something like 24 (daily cycle), 168 (weekly cycle), and 24×30=720 (monthly cycle). For a monthly macroeconomic series with a long history, the FFT might surface 12 (annual), 6 (semi-annual), and 3 (quarterly). Each of these periods becomes its own 2D reshape that the Inception block then convolves over, and the final forecast is a weighted combination across the three. Higher `top_k` lets the model express more overlapping cycles at the cost of compute; setting it too high mostly captures noise frequencies, so the paper's `top_k=2` to 5 is a reasonable range to sweep.

We then create a `NeuralForecast` object to manage the training and prediction pipeline:

```
from neuralforecast import NeuralForecast
nf = NeuralForecast(models=models, freq='M')
```

Training the model is straightforward. The `fit` method learns from the historical data:

```
nf.fit(df=Y_df)
```

Once the model is trained, we can generate forecasts.

```
Y_hat_df = nf.predict()
print(Y_hat_df.head())
```

This workflow shows how TimesNet converts time series into structured 2D representations, applies CNN-based feature extraction, and generates accurate forecasts. By explicitly modeling multi-periodicity, TimesNet excels at capturing complex seasonal patterns and trends that are difficult for traditional 1D models to learn.

Deep-learning forecasting moves fast. The models in this chapter, from linear DLinear to 2D TimesNet, are the current state of the art, but the field is far from settled. Two emerging classes are pushing the frontier: State Space Models, which scale to very long contexts at near-linear cost, and Diffusion Models, which give well-calibrated probabilistic forecasts. Both are likely to show up in the next round of benchmark papers.

Model selection

We've just covered eight different deep learning architectures, from LSTMs to Transformers to TCNs. This is the model zoo that overwhelms many practitioners.

You are likely asking the most important question: *Which one do I choose?*

In my experience, this is where most teams get stuck. They see a model zoo of new architectures published every month. Patch-this, i-that, Crossformer, Times-something. The sheer number of models is overwhelming, and it's easy to get lost in the hype.

Do not make the mistake of trying all of them. In my experience, a practical workflow is about making a few smart, sequential decisions. My goal isn't to show you every model. It's to give you a framework

for choosing the right one. This is the process I use to pick a tool from the zoo that solves the problem (not just the newest shiny tool).

When evaluating any model, we can use three guiding principles:

- **Simplicity First**: Complexity does not always equal performance. In fact, deceptively simple models can outperform complex ones, especially when the time series has clear, strong patterns. DLinear, for example, just uses simple linear layers on decomposed data (trend and seasonality). It frequently gets excellent results, proving that advanced attention mechanisms are not always necessary. This is the most common trap. Teams build complex, unmaintainable models to chase a tiny metric gain. This is how you get a Zillow-scale failure: a black box no one understands, trusts, or can fix. Always, always start with a simple, strong baseline. You might be surprised.
- **Channel Strategy** (**CI vs. CD**): This is the most important technical choice for multivariate (many series) forecasting. *Channel* is just another name for a single time series. A *multivariate* dataset has many channels.
    - CI models like PatchTST process each time series separately. Think of it as building one model and sharing its weights across all your series. This is your default, safe choice. Use it when cross-series correlations are weak, noisy, or just unknown. It prevents the model from learning spurious relationships.
    - If you are transitioning from *Chapter 6* note a key distinction: learning cross-series correlations is distinct from enforcing hierarchical coherence. While channel-dependent models capture dependencies between variables, they do not guarantee mathematical aggregation consistency across levels. Enforcing such alignment still requires the reconciliation methods covered previously.
    - CD models like iTransformer or TSMixer explicitly model interactions between the series. Use this only when you have strong, known relationships. For example, traffic in one area causes traffic in another. Or, all your series are part of one physical system, like an electricity grid.
- **Time Series Representation**: This is about how the model *looks* at your data. This has been the biggest source of innovation.
    - **Point-wise**: Older models looked at one data point at a time. This is slow and misses bigger patterns.
    - **Patch-wise** (**PatchTST**): The model looks at chunks (patches) of time. This was a breakthrough. It helps the model see small local trends and is much more efficient.
    - **Series-wise** (**iTransformer**): The model treats an entire time series as a single 'token'. This is radical. It's powerful for finding high-level, abstract relationships between many series at once.
    - **2D Transformation** (**TimesNet**): The model turns your 1D time series line into a 2D 'image' (using a Fast Fourier Transform). This lets it use powerful image-recognition networks (CNNs) to find complex seasonal patterns, like daily and weekly cycles—at the same time.

The following table summarizes these ideas.

Model	Backbone	Channel	Key	Computational	Best-Fit Scenarios

		Strategy	Innovation	Cost	
DLinear	Linear (single layer)	CI	Decomposition into trend and seasonality	Very Low	Series with strong trends/seasonality; fast baseline
N-BEATS	MLP	CI	Stacked residual blocks with basis expansion	Low	Univariate forecasting with strong trend/seasonality; interpretable variant
TSMixer	MLP	CD	Alternating time-mixing and feature-mixing MLPs	Low $(O(LC^2 + L^2C))$	Moderate (scales with both lookback length and channel count)
TFT	LSTM + Attention	CD	Variable selection + interpretable attention; native quantile output	High	Panel data with many covariates and need for interpretability/intervals
PatchTST	Transformer	CI	Patching to reduce sequence length	Moderate (attention is quadratic in patch count, but patching shrinks the sequence)	Long-term forecasting, weak/noisy cross-channel correlations
iTransformer	Transformer	CD	Inverted attention (series-as-tokens)	High (attention is quadratic in channel count)	High-dimensional multivariate forecasting with strong inter-series dependencies
Crossformer	Transformer	CD	Two-stage attention with router	Moderate	Multivariate forecasting with complex, dynamic inter-series relationships
TimesNet	CNN	CD	1D-to-2D FFT-	Moderate	General-purpose, multi-

	(Inception)		based transformation		periodic series

Table 7.2: Deep Learning model zoo

Please note that in the Computational Cost column, k denotes the look-back window length and T denotes the number of channels (time series).

Figure 7.6 captures the workflow as a decision tree. Read it top-to-bottom: every fork is a question you can answer cheaply about your data before committing to a model.

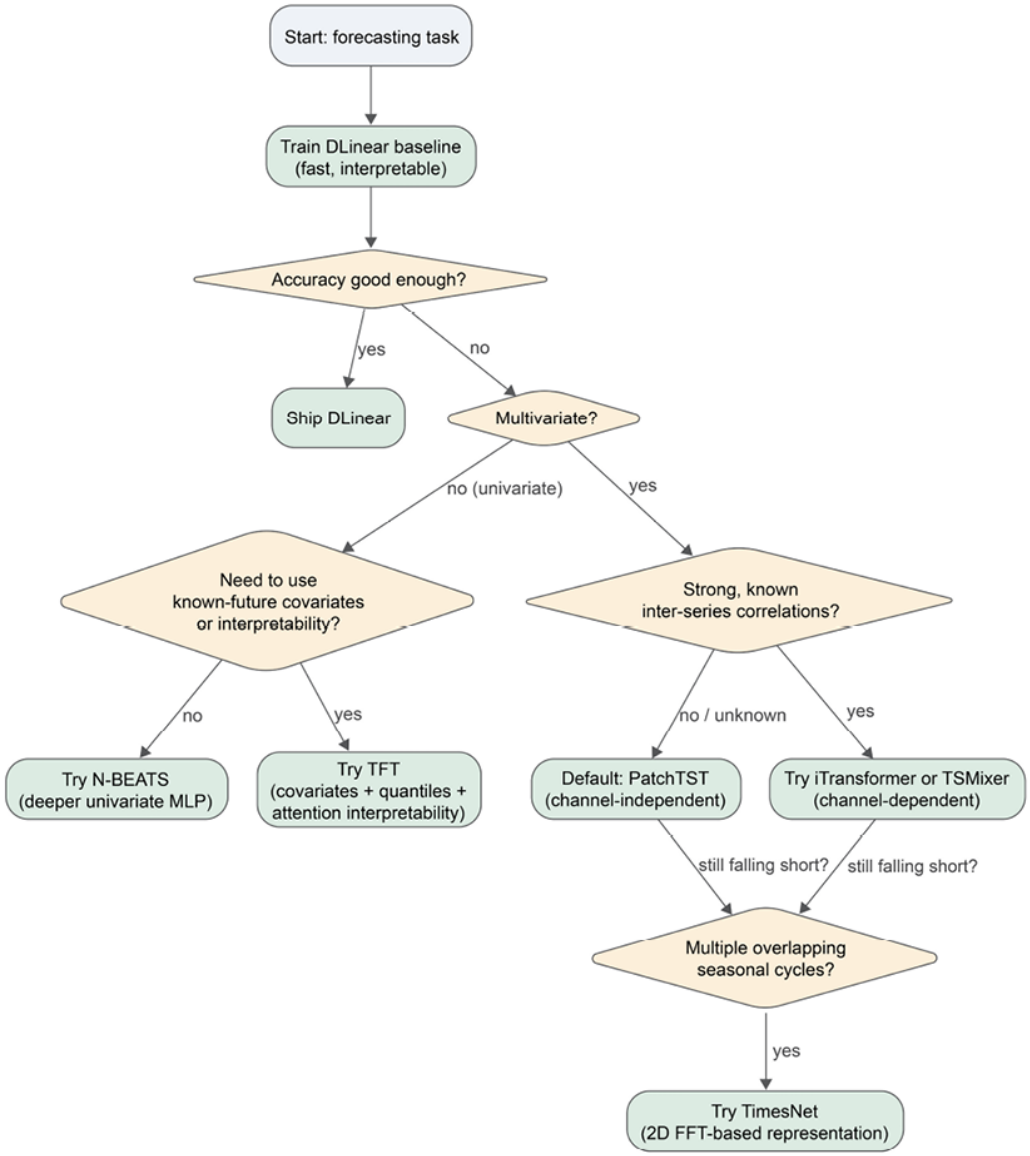


Figure 7.6: Model selection decision tree

Here is the process I follow when starting a new forecasting project. It's opinionated, but it's battle-tested and moves from the simplest, cheapest models to the most complex, forcing you to justify every

step.

My first model is DLinear. Always. It is fast to train, easy to interpret, and as research has shown, it often outperforms far more complex models. I use this as my benchmark. If a fancy, resource-hungry Transformer cannot beat DLinear on my data, I will not pay the computational or complexity price for it.

If DLinear leaves a noticeable gap and the series is univariate, my next stop is N-BEATS. It is still a pure MLP, still fast, but its stacked residual blocks and basis expansion let it model deeper non-linear patterns than a single linear layer can. The interpretable variant gives you a trend/seasonality decomposition you can show to a stakeholder.

This brings us to the most important decision point: handling multivariate data. My default is still PatchTST. Because it's channel-independent, it treats each series as its own problem. This is a safe starting point. It *prevents* the model from being confused by noisy or meaningless correlations a common problem in real-world data. Many large-scale systems fail this way; they learn spurious relationships that break in production. A CI model defends against that by construction.

As an advanced step, if I have strong evidence that my series are truly correlated (e.g., traffic sensors in the same city, or energy load from the same grid), then I will try a CD) model. In this case, iTransformer and TSMixer are my primary choices. I have to *prove* that the added complexity of modeling these interactions is worth it.

There is a separate axis I check before any of the above: do I have known-future covariates (calendar, prices, promotions, weather forecasts) and does someone need to understand the forecast? If yes to both, I reach for TFT. Its variable selection networks and interpretable attention give you a story to tell about each prediction, and it produces quantile forecasts natively, which matter when downstream decisions are asymmetric.

Finally, if the data is dominated by complex, overlapping cycles, for example, hourly, daily, *and* weekly patterns all at once, I'll test TimesNet. It was specifically designed for this. Its unique 2D representation allows it to find and model these multi-period patterns better than most 1D sequential models can.

You'll notice I didn't recommend LSTM or TCN in this workflow. This is intentional. We've covered them briefly so you recognize them but for most modern forecasting tasks, they are outperformed, slower, and more difficult to tune than the models above. Their main value today is educational (understanding the history) or as components inside more complex, hybrid systems.

Practical tips for training deep learning models

Choosing an architecture is half the work; getting it to train without burning a week of GPU credits or your patience is the other half. A few hard-won pieces of advice:

- Hardware: when does GPU actually pay off? For the small examples in this chapter (AirPassengers, 17k rows) train in 1-3 minutes on CPU but 10-20 seconds on a single consumer GPU; iTransformer on Traffic (862 series × 17k+ rows) genuinely needs a GPU and finishes in minutes there versus hours on CPU. The rule of thumb: until you exceed those thresholds, do not bother provisioning a GPU.

- Training time, ballpark per model. On a single hourly multivariate dataset with ~10k training rows and a forecast horizon around 100, expect (CPU / single GPU): DLinear ~ a few seconds / instant; N-BEATS ~ 1 min / 5 s; TSMixer ~ 2-5 min / 10-30 s; PatchTST ~ 2-5 min / 10-30 s; TFT ~ 5-10 min / 30-60 s; iTransformer (depends heavily on channel count) ~ 5-30 min / 30 s-2 min; Crossformer and TimesNet sit in the same range as PatchTST. These are rough, but the relative ordering is stable: linear and pure-MLP models are always order-of-magnitude faster than attention-based ones.
- Hyperparameter tuning. Two things matter most for deep forecasting models - the look-back window (`input_size` / `input_chunk_length` / `context_length`) and the model dimension (`d_model`, `hidden_size`, `layer_widths`). Sweep those before anything else. `NeuralForecast` integrates with `optuna` via AutoModel wrappers, for example:

```
from neuralforecast.auto import AutoNHITS
from neuralforecast import NeuralForecast
model = AutoNHITS(
    h=horizon,
    num_samples=20,        # trials
    backend="optuna",
)
nf = NeuralForecast(models=[model], freq="H")
nf.fit(df=Y_df)
```

Each `Auto*` class ships with a default search space curated by the maintainers; pass `config=...` to override. For darts, the equivalent is `gridsearch` for small spaces and an `optuna` integration shown in the darts docs for larger ones. Two pieces of advice: cap `num_samples` at 20-30 trials for an initial sweep (the marginal accuracy gain past that point is rarely worth the wall-clock), and always tune on a temporal validation split, never a random one.

Let's conclude!

Further Reading

- Bai, Shaojie, J. Zico Kolter, and Vladlen Koltun. 2018. "*An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling.*" arXiv. https://arxiv.org/abs/1803.01271.
- Chen, Si-An, and colleagues. 2023. "*TSMixer: An All-MLP Architecture for Time Series Forecasting.*" arXiv. https://arxiv.org/abs/2303.06053.
- Lim, Bryan, and colleagues. 2021. "*Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting.*" arXiv. https://arxiv.org/abs/1912.09363.
- Liu, Yong, and colleagues. 2023. "*iTransformer: Inverted Transformers Are Effective for Time Series Forecasting.*" arXiv. https://arxiv.org/abs/2310.06625.
- Makridakis, Spyros, and colleagues. "*The M6 Forecasting Competition.*" arXiv. https://arxiv.org/abs/2310.13357.
- Nie, Yuqi, and colleagues. 2022. "*A Time Series is Worth 64 Words: Long-term Forecasting with Transformers.*" arXiv. https://arxiv.org/abs/2211.14730.

- Oreshkin, Boris N., and colleagues. 2019. "*N-BEATS: Neural Basis Expansion Analysis for Interpretable Time Series Forecasting*." arXiv. https://arxiv.org/abs/1905.10437.
- Wu, Haixu, and colleagues. 2022. "*TimesNet: Temporal 2D-Variation Modeling for General Time Series Analysis*." arXiv. https://arxiv.org/abs/2210.02186.
- Zeng, Ailing, and colleagues. 2022. "*Are Transformers Effective for Time Series Forecasting?*" arXiv. https://arxiv.org/abs/2205.13504.
- Zhang, Yunhao, and Junchi Yan. 2023. "*Crossformer: Transformer Utilizing Cross-Dimension Dependency for Multivariate Time Series Forecasting*." OpenReview. https://openreview.net/forum?id=vSVLM2j9eie.

Summary

In this chapter, we covered modern deep learning for time series. We moved from foundational LSTMs and TCNs to the MLP-based models, DLinear and N-BEATS, that challenge our assumptions about complexity. We then walked through the Transformer revolution, from TFT's interpretable hybrid through the channel-independent (PatchTST) versus channel-dependent (iTransformer) distinction.

You now have a toolkit of strong architectures. Building a production-ready system requires more than just a good architecture. So far, all our models have produced point forecasts, a single *best guess* for the future.

In a real business, this is rarely enough. When a model predicts 10,000 units of demand, a manager needs to know: is that 10,000 ±100, or 10,000 ±8,000? The answer determines inventory, budget, and risk.

In *Chapter 8*, we will move beyond point forecasts. You'll learn practical guides to Conformal Prediction for generating reliable uncertainty estimates for any model. We will also tackle volatility modeling with GARCH models, a critical task for financial risk analysis. While deep learning models can produce accurate point forecasts, they often lack built-in uncertainty quantification.

Get this book's PDF version and more

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

8

Quantifying Time Series Uncertainty with Conformal Prediction

In the previous chapters, we built models to generate a single point forecast, our best guess for the future. In *Chapter 7*, we saw how complex deep learning models can capture intricate patterns to improve this guess.

However, a single-point guess is rarely enough for real-world decisions. It answers, "what do you think will happen? But not how confident are you? A forecast of 10,000 units means something very different if the likely range is 9,900-10,100 versus 2,000-18,000. While the first is a planning number the second is a guess.

This chapter tackles this second, very critical question. We will move from point forecasts to prediction intervals. You will learn a modern, reliable framework called conformal prediction that can wrap any model, from an ARIMA to an XGBoost to a Transformer, and provide a statistical guarantee on its uncertainty. We will then build adaptive intervals that respond to changing conditions (like promotions) and explore specialized models for handling financial volatility.

In this chapter:

- The problem with uncertainty
- The conformal prediction framework
- Fixed-width intervals with mapie
- Adaptive intervals with conformalized quantile regression (CQR)
- Recipes for the modern time series ecosystem

- Conformal prediction for non-exchangeable time series
- Production-ready uncertainty
- Modeling financial volatility
- Choosing your uncertainty method

The chapter has four sections of substantive methods, each matched to a distinct error pattern. Before we dive in, *Figure 8.1* maps the territory: diagnose your model's residuals once, then read the path that lands on the right method. Refer back as you work through the sections.

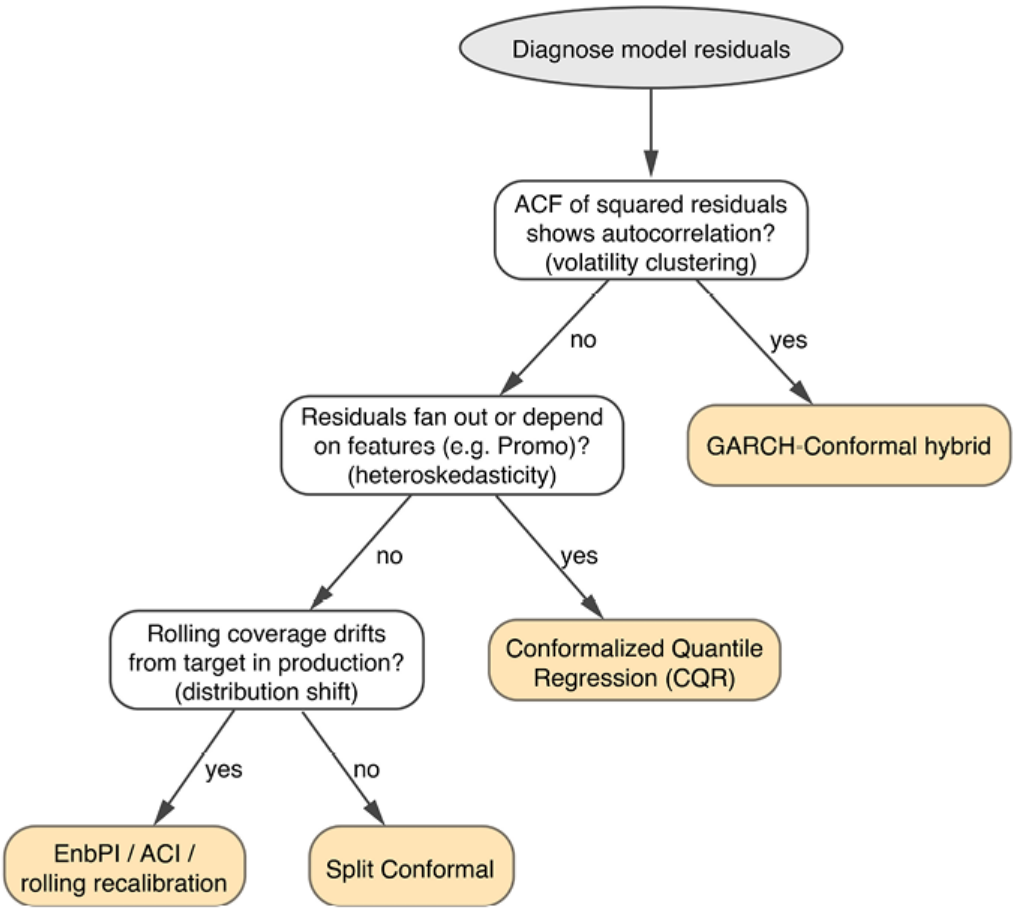


Figure 8.1: Decision tree for choosing an uncertainty method

Start at the top with residual diagnostics, follow the yes/no branches, and arrive at the method that matches your error structure. Each leaf corresponds to a section in this chapter.

The problem with uncertainty

In the preceding chapters, we constructed models for time series forecasting. We used tree-based models like XGBoost to predict M5 Walmart sales in *Chapter 5* and built deep learning models in *Chapter 7*. All these efforts produced a single point forecast, a prediction, for example, that sales tomorrow will be 10,000 units.

In a production system, a point forecast is insufficient. As we mentioned before, it doesn't answer the question: how confident are you?

This becomes clear when we look at common forecasting use cases:

- **Inventory:** As we saw with the retail inventory challenges in *Chapter 1*, a forecast of 10,000 units is useless for stocking. Should the manager stock 10,000 units, risking a stockout if demand is 11,000? Or stock 12,000, risking waste if demand is 9,000? The real business need is a prediction interval, a range of likely outcomes.
- **Capacity planning:** A web services team needs to know the peak likely user load, not the average, to provision servers. Similarly, for the capacity planning problems we introduced (like provisioning cloud servers in *Chapter 1*), a web services team needs to know the peak likely user load, not the average, to provision servers.
- **Anomaly detection:** An observation is only anomalous if it falls outside the range of expected values.

The challenge is that most traditional methods for generating these intervals are unreliable. They are built on assumptions that are often violated in practice, leading to silent failures that create operational risk.

Let us look at some common failure modes of traditional methods.

- **Parametric failures:** Classical statistical models, like the ARIMA and Exponential Smoothing methods discussed in *Chapter 3*, often produce prediction intervals. These intervals, however, are typically based on the assumption that the model's residuals (the forecast errors) are

independent, homoskedastic (have constant variance), and follow a normal distribution. In practice, especially with complex, real-world data like the M5 sales, these assumptions are almost always wrong. The residuals are often heavy-tailed and heteroskedastic (their variance changes over time). Relying on intervals from a model whose assumptions are broken is a recipe for over-confident, narrow predictions.

- **Bayesian failures:** Bayesian methods, such as **Bayesian Neural Networks**, are another approach to uncertainty. They estimate a full posterior distribution over model parameters. This captures what the model does not know (epistemic uncertainty) and inherent randomness (aleatoric uncertainty). However, Bayesian credible intervals have a major caveat. They are valid only if the chosen model architecture and the subjective prior distribution are correct. There is often no general procedure for choosing a prior that correctly reflects ignorance. If the model is wrong, the Bayesian framework cannot warn you. It simply updates its beliefs based on those wrong assumptions. This can lead to unstable results. The resulting intervals may not be well-calibrated, meaning a 95% interval may not contain the true value 95% of the time.

This chapter focuses on a different paradigm, one born from a critique of this very flaw. We are building production systems that must be reliable.

We need a framework that provides a guarantee of performance even if our model is imperfect. Instead of asking, How much do I believe in this range, given my model?, we will ask, How often will this range be correct, regardless of the model?.?

This is the promise of **conformal prediction**.

> **The Sources of Uncertainty**
>
> Before we build intervals, it helps to understand the different types of uncertainties.



- **Aleatoric uncertainty**: This is the irreducible, natural randomness in the data, such as unpredictable customer behavior.
- **Epistemic uncertainty**: This is uncertainty from the model itself. It includes using the wrong features, having a poor model architecture, or not having enough data. Conformal prediction is excellent at quantifying this.
- **Distributional uncertainty**: This is uncertainty caused by the data-generating process changing over time. This is also known as *distribution drift* and is a key challenge in production systems.

The conformal prediction framework

Conformal Prediction (**CP**) is a model-agnostic wrapper that can take any pre-trained model, whether an ARIMA, an XGBoost, or a deep neural network, and convert its heuristic predictions into statistically rigorous prediction intervals.

To understand how conformal prediction achieves this, we will break it down into two key components:

- The validity guarantee that underpins the method
- The practical algorithm used to construct prediction intervals

The conformal guarantee

The core of conformal prediction is a distribution-free validity guarantee. It states that for a chosen error rate T (e.g., $T = 0.1$ for 90% confidence), the resulting prediction interval $C(X_{n+1})$ for a new data point will contain the true outcome Y_{n+1} with a probability of at least $n \times n$.

Formally:

$$P(Y_{n+1} \in C(X_{n+1}) \geq 1 - \alpha$$

This guarantee is *marginal*, meaning it holds on average over all data points, and points, single-step: it is a statement about one new point X_{n+1} at a time. A multi-step forecast across *H* horizons gets *H* separate marginal guarantees rather than a joint guarantee; we return to the implications in multi-horizon coverage. Crucially, the guarantee depends on only one assumption: **exchangeability**. Exchangeability is a weaker condition than i.i.d. (independent and identically distributed); it means that the joint distribution of the data is invariant to permutation (the order doesn't matter).

This *order doesn't matter* assumption is a clear problem for time series, which are defined by their order. Methods that explicitly handle non-exchangeable streams (EnbPI, ACI, weighted conformal, block conformal) are an active research area; we point readers to the literature rather than cover them in depth here. For now, we will operate under a simplified assumption: that after a good model has captured the trends, seasonality, and other temporal patterns, the *residuals* of that model are (approximately) exchangeable. Concretely: with a strong XGBoost on the Rossmann data, what is left after the model captures day-of-week seasonality, holidays, and the Promo effect should look roughly i.i.d. (store-level noise without obvious time structure), and i.i.d. residuals are approximately exchangeable.

The split conformal algorithm

The most common and practical variant of CP is **split conformal prediction**, also known as inductive conformal prediction. It works by splitting the data, defining a *nonconformity score,* and finding a quantile of those scores.

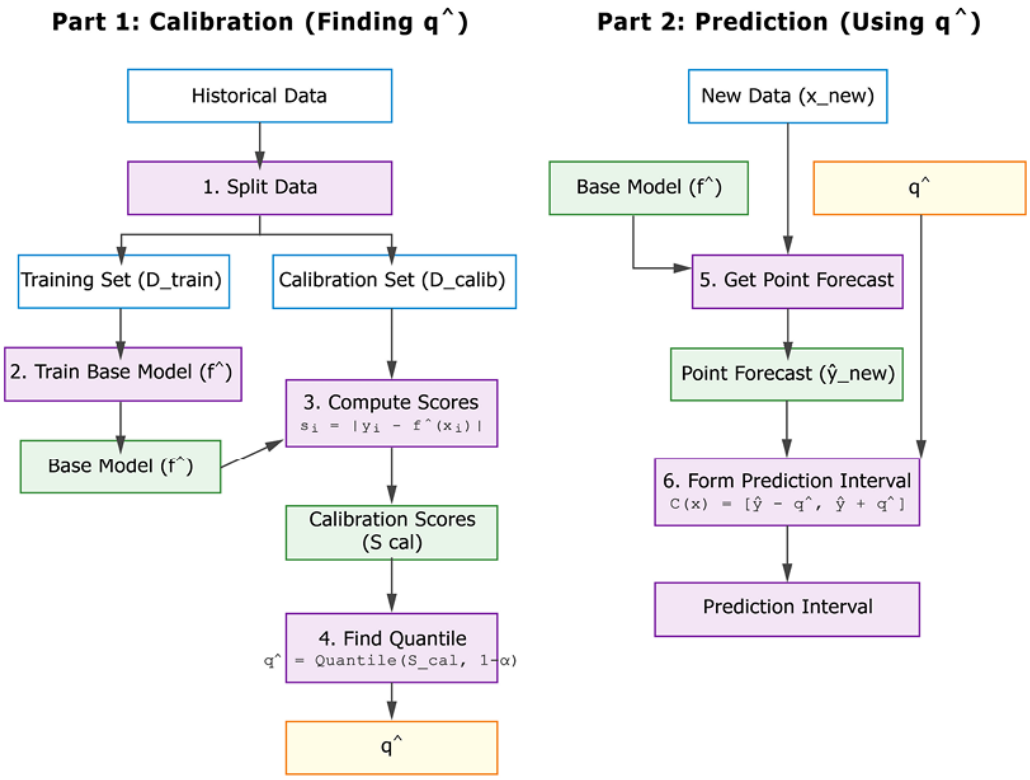


Figure 8.2: The split conformal prediction process

This diagram illustrates the two-part process of split conformal prediction. The first part, Calibration, shows how we use the held-out calibration set (D_{calib}) to compute a set of nonconformity scores, which are typically the absolute residuals. We then find a specific quantile of these scores to get a single, fixed value, p. The second part, Prediction, shows how this value is used. For any new point forecast ($\hat{y}_{new}$), we simply add and subtract this fixed p value to create the final, guaranteed prediction interval 5. The flowchart highlights that p is a single, fixed number used for all future predictions.

Here is the step-by-step algorithm, which is also visualized in *Figure 8.2*.

1. **Data split**: We cannot use the same data to train the model and calibrate its uncertainty. Doing so would lead to optimistic, over-fitted intervals. We must split our historical data into two disjoint sets:

- **Training Set** (p, d, q): Used to fit the base forecasting model (e.g., XGBRegressor).
- **Calibration Set** (p, d, q): Used to calibrate the uncertainty intervals. This set is held out and *never* seen by the model during training.

2. **Define a nonconformity score**: We need a nonconformity score $p, d, q)$, which is a function that measures how weird or non-conforming a new data point x, y is, given our model. A heuristic of uncertainty works well. For regression, the simplest and most common score is the absolute residual

$$s_i = |y_i - \hat{f}(x_i)|$$

Here, $\hat{f}$ is our model trained on p, d, q. A larger score Y_t means the model's prediction $\hat{f}(x_i)$ was *worse* or *less conforming* to the true value y_t' .

3. **Compute calibration scores**: We use our fitted model $\hat{f}$ to make predictions for every point Y_t in the calibration set D_{calib}. We then compute the nonconformity score (the absolute residual) for each of these n_{cal}points. This gives us a list of calibration scores

$$S_{cal} = \{s_1, s_2, \ldots s_{n_{cal}}\}$$

4. **Find the quantile**: This is the core calibration step. To achieve a $n \times n$ coverage (e.g., 90% coverage where $\alpha = 0.1$), we find the empirical quantile of the calibration scores. Due to a small correction for finite samples, we calculate the quantile at the $((1-\alpha)(n_{cal}+1)) / n_{cal}$

For a reasonably large calibration set, this is simply the $n \times n$ quantile of $|R_t|$. Let's call this quantile value p.

$$\hat{q} = Quantile(S_{cal}, 1-\alpha)_{\text{i}}$$

5. **Form prediction intervals**: For any new, unseen data point x_{new}, we first get its point forecast, $\hat{y}_{new} = \hat{f}(x_{new})$. The final, valid prediction interval

is constructed using p

$$C(x_{new}) = [\hat{y}_{new} - \hat{q}, \hat{y}_{new} + \hat{q}]$$

This simple procedure is guaranteed to achieve the desired coverage. The logic is that, by exchangeability, the (unknown) score of our new test point Y_{n+1} is just another draw from the same distribution as the scores in $|R_t|$. It has a $1/n_{cal} + 1$ chance of being in any rank position. Therefore, the probability that Y_{n+1} will be larger than p (the $n \times n$ quantile of the past scores) is at most T. This means the probability that $s_{new} \leq \hat{q}$ is at least $n \times n$, which is the definition of our coverage guarantee.

This provides the robust, model-agnostic, and distribution-free foundation for uncertainty that we need for reliable systems.

The following table summarizes the key differences between these frameworks, highlighting their underlying principles, types of uncertainty captured, guarantees, and practical trade-offs.

Property	**Parametric (e.g., ARIMA)**	**Bayesian (e.g., BNN)**	**Conformal Prediction**
Core Principle	Assumes residuals follow a specific, known distribution (e.g., normal).	Uses priors and likelihoods to compute a posterior distribution over parameters.	Calibrates empirical scores from a holdout set to guarantee coverage.
Uncertainty Type	Aleatoric (Data Noise) only.	Epistemic (Model) and Aleatoric (Data), decomposed.	Model-agnostic. Captures total predictive uncertainty (aleatoric + epistemic

			combined, not decomposed).
Guarantee	None. Fails if assumptions (e.g., normality) are violated.	None. Credible intervals are conditional on the model and prior being correct.	Guaranteed marginal coverage $P(Y \in C(X)) \geq 1-\alpha)$
Key flaw	Fails silently when assumptions are wrong.	Can be sensitive to *unjustified priors* and gives no alert if the model is misspecified.	The guarantee is marginal, not conditional. It may under-cover specific subgroups.
Best For	Simple, well-behaved processes.	Problems with strong domain knowledge that can be encoded in priors.	Production systems requiring robust, verifiable performance guarantees.

Table 8.1: Comparison of Uncertainty Frameworks

Fixed-width intervals with mapie

Before building complex adaptive intervals, we should establish a simple, fixed-width baseline. This serves as a *Model 1.0* and provides the foundation for all subsequent analysis. This module implements the split conformal algorithm discussed in the previous section. The process requires three distinct data splits, not the typical training and test splits.

- **Training set** (p, d, q): Used to fit the base forecasting model (an XGBRegressor).
- **Calibration set** (D_{calib}): Used only to calibrate the uncertainty. This set is held out and never seen by the model during its training phase. This separation is the core principle of split conformal prediction.
- **Test set**: Used to evaluate the final performance of the generated intervals.

The `mapie` library, a scikit-learn-compatible tool, is used for this implementation. Specifically, its `SplitConformalRegressor` class wraps an already-fit estimator and takes a separate calibration set via its `conformalize()` method. The `confidence_level` is set at construction (e.g., 0.9 for 90% coverage). Intervals come out of `predict_interval()`.

The *Chapter 5* prep pipeline ships `processed_train_m5.parquet` from the M5 Forecasting Accuracy competition (Zenodo, CC-BY 4.0). Run `download_m5.py` from the *Chapter05* folder once to fetch the raw files, then the prep notebook produces the parquet. It has 39 columns including engineered features: Promo (1 if today's `sell_price` is below 95% of its trailing 30-day rolling median, 0 otherwise), `snap_active` (SNAP food-benefit day in the item's state), `DayOfWeek` (0=Mon..6=Sun), `cat_id` (`FOODS` / `HOBBIES` / `HOUSEHOLD`), plus lag and rolling-window features. The head of the prediction-relevant subset looks like this:

id	Date	DayOfWeek	cat_id	store_id	snap_active	Promo	sales
`FOODS_1_052_`	2011-01-29	5	`FOODS`	`CA_1`	0	0	3

CA_1_validation							
FOODS_1_0522_CA_1_validation	2011-01-30	6	FOODS	CA_1	0	1	12
FOODS_1_0522_CA_1	2011-01-31	0	FOODS	CA_1	0	0	0

_validation							
FOODS_1_052_CA_1_validation	2011-02-01	1	FOODS	CA_1	1	0	1
FOODS_1_052_CA_1_val	2011-02-02	2	FOODS	CA_1	1	0	4

id							
at							
io							
n							

Table 8.2: The prediction-relevant subset

The following code walks through the full pipeline: splitting the data, training the model, calibrating the conformal wrapper, and generating prediction intervals on the test set. We separate the target (sales) and string-typed columns (Date, id) from the feature matrix so XGBoost sees only numeric features; the date and id are carried alongside so we can rejoin them against the test slice for grouping (e.g., by `Promo` or `cat_id`) and plotting.

```
import numpy as np
import pandas as pd
from xgboost import XGBRegressor
from mapie.regression import  SplitConformalRegressor
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
df = pd.read_parquet("processed_train_m5.parquet")
y = df["sales"]
ds = df["Date"]
ids = df["id"]
X = df.drop(columns=["sales", "Date", "id"])
# 1. Create Train, Calibration, and Test splits
# We need a 3-way split: 60% train, 20% calibration, 20% test
X_train, X_rem, y_train, y_rem = train_test_split(
    X, y, test_size=0.4, random_state=42
)
X_calib, X_test, y_calib, y_test = train_test_split(
    X_rem, y_rem, test_size=0.5, random_state=42
)
print(f"Train size: {len(X_train)}")
print(f"Calibration size: {len(X_calib)}")
print(f"Test size: {len(X_test)}")
# 2. Train the base model (as in Ch. 5)
estimator = XGBRegressor(random_state=42)
estimator.fit(X_train, y_train)
# 3. Wrap the prefit estimator and calibrate on the held-out set.
alpha = 0.1
mapie = SplitConformalRegressor(
```

```
    estimator=estimator, confidence_level=1 - alpha)
mapie.conformalize(X_calib, y_calib)
# 4.  Generate predictions and intervals on the test set.
y_pred, y_pis = mapie.predict_interval(X_test)
# 5. Generate predictions and intervals on the test set
alpha = 0.1   # For 90% coverage
print(f"Generating predictions with {1-alpha:.0%} intervals...")
y_pred, y_pis = mapie.predict(X_test, alpha=alpha)
# y_pis is a numpy array of shape (n_samples, 2, 1)
# [:, 0, 0] is the lower bound, [:, 1, 0] is the upper bound
y_lower = y_pis[:, 0, 0]
y_upper = y_pis[:, 1, 0]
print("Done.")
```

Please note that the `60/20/20` split is achieved through two sequential `train_test_split` calls. The first call carves out the 60% `X_train`, leaving the remaining 40% as `X_rem`. The second call splits this `X_rem` perfectly in half, creating a 20% `X_calib` and a 20% `X_test`. This three-way partition is the critical foundation of this method. Let's break down a few elements:

- `estimator.fit(X_train, y_train)`: This is the standard model training procedure. The `XGBRegressor` only ever *sees* the `D_train` data, ensuring its training is independent of calibration.
- `SplitConformalRegressor(estimator=estimator, confidence_level=0.9)`: The wrapper takes an already-fit estimator. The `confidence_level` sets the target coverage for the intervals (we pass `1 - alpha`).
  `mapie.conformalize(X_calib, y_calib)`: The calibration step. `mapie` runs `estimator.predict()` on the `X_calib` data and computes the absolute residuals $s_i = |y_i - \hat{f}(x_i)|$. These residuals are the *nonconformity scores*, It then finds the empirical $n \times n$ quantile of these scores to get a single, fixed value, p.
- `mapie.predict(X_test)`: This is the inference step. For each new point in `X_test`, the model generates a point forecast $\hat{y}_{new}$. It then applies the same fixed $\hat{y}_{new}$value (calculated during calibration) to create the interval: $C(x_{new}) = [\hat{y}_{new} - \hat{q}, \hat{y}_{new} + \hat{q}]$.

- The `y_pis` (prediction intervals) array contains the lower and upper bounds.

Let's visualize the intervals on a subset of the test data.

```
def plot_intervals(y_test, y_pred, y_lower, y_upper, n_sample=100):
    """Helper function to plot test data and intervals."""
    # Sort for cleaner plotting
    idx = np.argsort(y_test.iloc[:n_sample].values)
    y_test_sample = y_test.iloc[:n_sample].values[idx]
    y_pred_sample = y_pred[:n_sample][idx]
    y_lower_sample = y_lower[:n_sample][idx]
    y_upper_sample = y_upper[:n_sample][idx]
    plt.figure(figsize=(12, 6))
    plt.plot(y_test_sample, "ko", label="Actual Sales")
    plt.plot(y_pred_sample, "b-", label="Point Forecast")
    plt.fill_between(
        np.arange(n_sample),
        y_lower_sample,
        y_upper_sample,
        alpha=0.2,
        color="blue",
        label="90% Prediction Interval"
    )
    plt.xlabel("Test Sample Index (Sorted by Sales)")
    plt.ylabel("Sales")
    plt.legend()
    plt.title("Split Conformal Prediction Intervals")
    plt.show()
# Plot the first 100 test points
plot_intervals(y_test, y_pred, y_lower, y_upper, n_sample=100)
```

The output is:

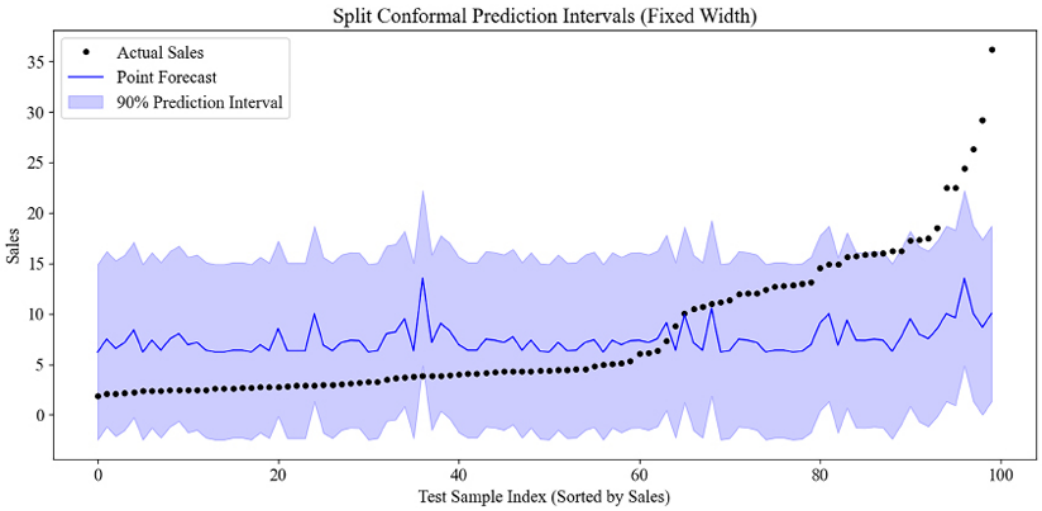


Figure 8.3: Split conformal 90% prediction intervals on test points sorted by actual sales

In this figure, the shaded band has constant width across the range: too wide for low-sales items (**HOBBIES** with one or two units per day) where the lower bound runs negative, and visibly straining to cover the right-hand high-sales tail (**FOODS** on **Promo** days).

The resulting plot shows a shaded blue band around the point forecast. The most important thing to notice is that the width of this band, $y_{upper} - y_{lower}$, is *constant*. This is because p is a single, fixed number calculated from the calibration set, and it is applied uniformly to all new predictions.

We now have valid 90% intervals. But are they *good* intervals? As we will see, this fixed-width assumption is a significant problem.

Adaptive intervals with conformalized quantile regression (CQR)

The fixed-width intervals from *Fixed-width intervals with mapie* are a substantial improvement over no intervals at all. They come with a robust statistical guarantee, and we can validate their 90% coverage on the test set.

However, they have a hidden flaw. The guarantee $P(Y \in C(X)) \geq 1 - \alpha$ is **marginal**, meaning it holds *on average* across the entire dataset. It does not guarantee 90% coverage for specific subgroups.

This becomes a problem when our model's uncertainty is not constant. It means our model is predictably less accurate under certain conditions.

For the M5 data, we can spot this easily. First, we can plot the absolute error (the residual) against the model's prediction.

```
# Assumes y_test and y_pred are from the previous section
import matplotlib.pyplot as plt
import seaborn as sns
residuals_abs = np.abs(y_test - y_pred)
plt.figure(figsize=(10, 6))
sns.scatterplot(
    x=y_pred,
    y=residuals_abs,
    alpha=0.3,
    s=10,
    label='Absolute Residual'
)
sns.regplot(
    x=y_pred,
    y=residuals_abs,
    scatter=False,
    lowess=True,
    line_kws={'color': 'red', 'linestyle': '--', 'linewidth': 2},
    label='Trend (LOWESS)'
)
plt.xlabel("Predicted Sales")
plt.ylabel("Absolute Error (Residual)")
plt.title("Heteroskedasticity: Error is Not Constant")
plt.legend()
plt.show()
```

On running this code, our output is:

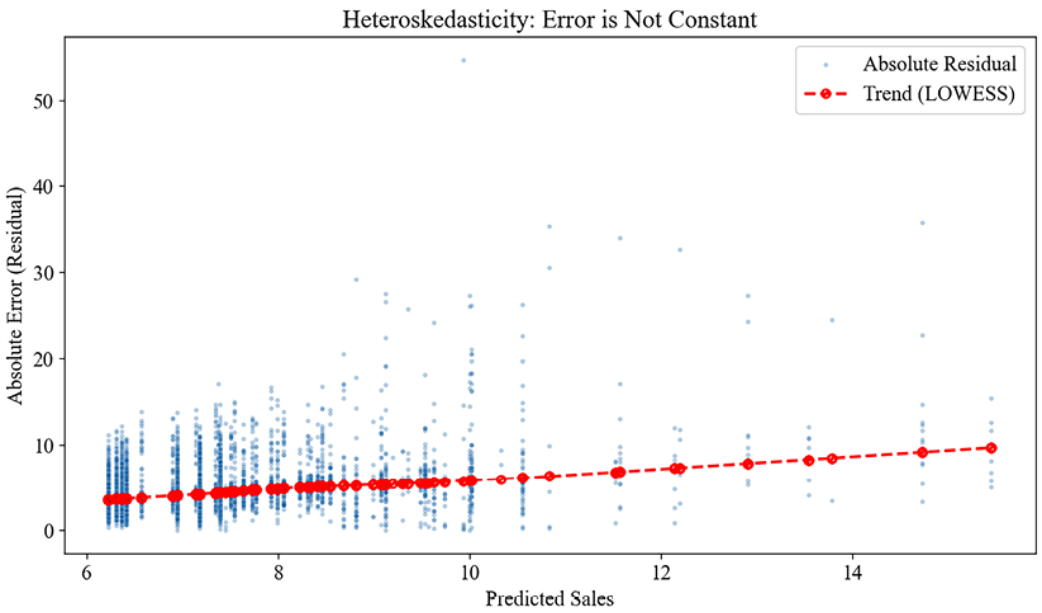


Figure 8.4: Absolute residuals against predicted sales

In this figure, the point cloud fans out as the prediction grows, and the LOWESS trend slopes up: a classic visual signature of heteroskedasticity.

The resulting *fanning out* shape clearly shows that as the predicted sales get higher, the model's error also tends to get larger.

The cause for the M5 data is even more specific. The model is less certain on promotion days. We can see this by plotting the residuals grouped by the `Promo` feature.

```
# Assumes X_test is available
promo_results = pd.DataFrame({
    'Absolute Error': residuals_abs,
    'Promotion': X_test['Promo'].map({0: 'No Promo', 1: 'Promo'})
})
plt.figure(figsize=(8, 6))
sns.boxplot(
    data=promo_results,
    x='Promotion',
    y='Absolute Error'
)
plt.title("Error is Higher During Promotions")
plt.ylabel("Absolute Error (Residual)")
plt.xlabel("")
plt.show()
```

The output:

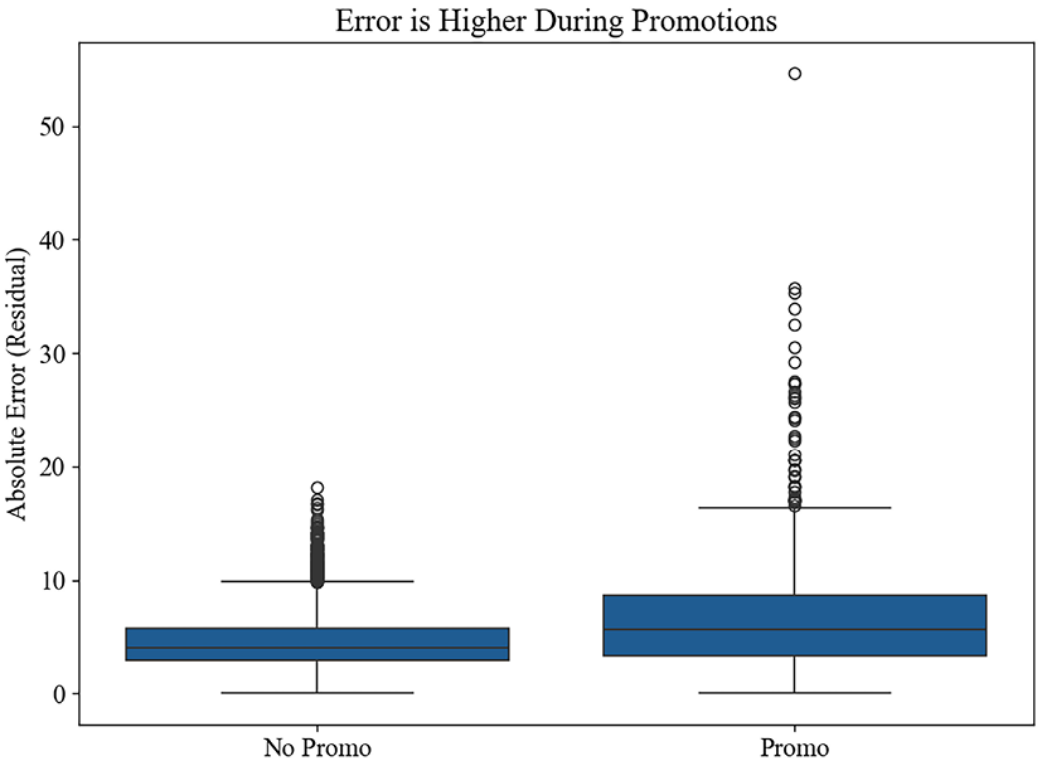


Figure 8.5: Absolute residuals on the test set, split by Promo status

In the figure, **Promo** days have a wider distribution and many more outliers, confirming that the model's uncertainty is feature-driven.

This plot confirms it: the distribution of errors for *Promo* days is larger and wi*der than* for "`No Promo`" days.

The failure of fixed-width intervals

This heteroskedasticity breaks our fixed-width interval. The single value p we calculated is a compromise: it's too small for promotion days and too large for normal days.

We can prove this by checking the **conditional coverage** (the coverage for each subgroup).

```
# Assumes y_lower and y_upper are from the fixed-width model
results_df = X_test.copy()
results_df['y_test'] = y_test
results_df['y_lower'] = y_lower
results_df['y_upper'] = y_upper
results_df['covered'] = (results_df['y_test'] >= results_df['y_lower']) & \
                        (results_df['y_test'] <= results_df['y_upper'])
conditional_coverage = results_df.groupby('Promo')['covered'].mean()
print(conditional_coverage)
# Expected Output:
Promo
0    0.941  # Over-covered (intervals are too wide)
1    0.832  # Under-covered (intervals are too narrow)
Name: covered, dtype: float64
```

This is the critical failure. Our 90% interval is only covering 83% of promotion days, exposing us to risk. At the same time, it's 94% covered on normal days, meaning our intervals are inefficiently wide.

The problem is clear: we are using a *fixed* interval for a *variable* error. The solution is to build an *adaptive* interval, one that gets wider when `Promo=1` and stays narrower when `Promo=0`. This is the goal of CQR.

The solution to feature-dependent uncertainty is an interval that is also feature-dependent. If the model is less certain when `Promo=1`, the interval should be wider on those days.

We achieve this by combining Quantile Regression with Conformal Prediction, resulting in **CQR**. To understand CQR a little further, let's discuss some of its guiding principles:

- **Quantile regression:** A quantile regression predicts a chosen percentile of the conditional distribution rather than its mean. Train a model with the pinball loss at $\tau = 0.05$ and its output is the value below which 5% of outcomes fall for the given features; train another $at\ \tau = 0.95$ and the gap between them is the model's belief about how spread out the outcomes are. Wider gap means more uncertainty at that feature combination. So we train two models. One predicts a low quantile (e.g., the 5th percentile, $\alpha/2 = 0.05$) and the other predicts a high quantile

(e.g., the 95th percentile, $1 - \alpha/2 = 0.95$). Tree-based models like LightGBM and XGBoost can do this natively by setting their objective to quantile and specifying the alpha parameter (which represents the quantile). Since these models use features like Promo, their quantile predictions will already be adaptive: the predicted interval $[\hat{y}_{low}, \hat{y}_{high}]$ will be wider for promotions.

- **Conformalization:** The intervals from quantile regression alone are just heuristic. They have no *guarantee* of 90% coverage. We must *conformalize* them.
- **New Nonconformity Score:** We use the calibration set to compute a new score, Y_t. This score measures how far the true value y_t' is *outside* its predicted interval $[\hat{y}_{low_i}, \hat{y}_{high_i}]$.

$$s_i max(\hat{y}_{low_i}, y_i - \hat{y}_{high_i})$$

  If y_t' is inside the interval, the score is negative or zero. We only care about positive scores, which represent miscoverage.

- **Quantile of calibration scores:** We again find the $n \times n$ quantile of these calibration scores, $|R_t|$. Let's call this p. This p represents the *correction factor* we need.
- **Final prediction interval:** The final CQR interval is the original quantile interval, expanded on both sides by p.

$$C_{CQR}(x_{new}) = [\hat{y}_{low} - \hat{q}, \hat{y}_{high} + \hat{q}]$$

This CQR method gives us the best of both worlds: the **adaptivity** of quantile regression and the **validity guarantee** of conformal prediction.

In the final part of this section we look at an implementation.

Implementation with mapie

`mapie` provides a dedicated class, `ConformalizedQuantileRegressor`, that handles this entire process. It takes three pre-fit estimators in the order [low quantile, high quantile, median]; the median is the point forecast that lives inside the interval.

```
from lightgbm import LGBMRegressor
from mapie.regression import ConformalizedQuantileRegressor
# Set the target error rate
alpha = 0.1  # 90% coverage
# Train low (α/2), high (1-α/2), and median quantile models.
estimator_low = LGBMRegressor(objective='quantile', alpha=alpha/2, random_stat
    objective='quantile', alpha=1.0 - alpha/2, random_state=42
)
estimator_median = LGBMRegressor(objective='quantile',
                                 alpha=0.5, random_state=42)
estimator_low.fit(X_train, y_train)
estimator_high.fit(X_train, y_train)
estimator_median.fit(X_train, y_train)
# 3. Wrap with ConformalizedQuantileRegressor
# Pass both estimators as a tuple
cqr = MapieQuantileRegressor(
    estimator=(estimator_low, estimator_high),
    cv="prefit"  # Use our dedicated calibration set
)
# Wrap with ConformalizedQuantileRegressor; order must be [low, high, median].
cqr = ConformalizedQuantileRegressor(
    estimator=[estimator_low,
    estimator_high, estimator_median],
    confidence_level=1 - alpha, prefit=True,
)
# 4 Calibrate on the held-out set.
# This computes the CQR nonconformity scores and q_hat.
print("Calibrating CQR wrapper...")
cqr.conformalize(X_calib, y_calib)
# 5. Generate predictions and *adaptive* intervals
print("Generating CQR intervals...")
y_pred_cqr, y_pis_cqr = cqr.predict_interval(X_test)
y_lower_cqr = y_pis_cqr[:, 0, 0]
y_upper_cqr = y_pis_cqr[:, 1, 0]
print("Done.")
```

Now, we can recreate the plot in *Figure 8.3*, but overlay our new adaptive intervals. This plot will be striking. It will show the fixed-width (light blue)

band failing to cover points during high-sales periods, while the CQR (dark blue) band dynamically *widens* in those same periods, successfully capturing the true sales.

Recipes for the modern time series ecosystem

We have worked with the `mapie` library in this chapter and we now know that it is an excellent, general-purpose tool. However, the modern Python time series libraries we introduced in *Chapter 4* often have their own streamlined APIs for conformal prediction. This section provides practical, copy-paste recipes for `nixtla`/`statsforecast`, `sktime`, and `darts`.

Library	Key Class / Function	Primary Use Case
`mapie`	`SplitConformalRegressor`, `ConformalizedQuantileRegressor`, `TimeSeriesRegressor`	An existing `sklearn`-style model that needs CP. `MapieTimeSeriesRegressor` for block bootstrap on non-exchangeable series; MapieQuantileRegressor for CQR.
`statsforecast`	`ConformalIntervals`	ARIMA / ETS at scale.
`mlforecast`	`ConformalIntervals` (passed to `MLForecast`)	Gradient-boosted forecasters in the `nixtla` ecosystem.
`neuralfo`	`ConformalInterva`	Deep models (NHITS, NBEATS, TFT) from

`recast`	`ls` (passed to `NeuralForecast`)	*Chapter 7*.
`sktime`	`ConformalIntervals` (wrapper), `EnbPIForecaster`	An `sktime` forecaster you do not want to rewrite.
`darts` (native)	`ConformalNaiveModel`, `ConformalQRModel`	Wrapping any Darts forecaster (regression or neural) without leaving the Darts API.
`conformal-tights`	`ConformalCoherentQuantileRegressor`, `DartsForecaster`	When you need multiple quantile levels (e.g., 5/25/50/75/95) with a guarantee they do not cross, and two-level (absolute + relative residual) calibration. Wraps `sklearn` regressors and darts forecasters.

Table 8.3: Python ecosystem for conformal prediction

Recipe 1: nixtla/statsforecast

The `nixtla` ecosystem (`statsforecast`, `mlforecast`, `neuralforecast`) has first-class support for conformal prediction. It is not a wrapper, but an argument you pass to the model itself. It uses residuals from internal cross-validation to build its calibration set. `ConformalIntervals(h=24, n_windows=5)` controls the forecast horizon and the number of CV splits used for residual collection. `train_df` follows the `nixtla` panel schema (`unique_id, ds, y`), and the resulting `forecasts_df` carries lower/upper columns named after the level you requested (here, AutoARIMA-lo-90 and AutoARIMA-hi-90).

```
from statsforecast import StatsForecast
from statsforecast.models import AutoARIMA
from statsforecast.utils import ConformalIntervals
```

```
intervals = ConformalIntervals(h=24, n_windows=5)
models = [AutoARIMA(prediction_intervals=intervals)]
sf = StatsForecast(models=models, freq='H', n_jobs=-1)
forecasts_df = sf.forecast(df=train_df, h=24, level=[90])
```

Recipe 2: sktime

`sktime` uses its compositor API, treating conformal prediction as a wrapper that adds new capabilities to a base forecaster.

```
from sktime.forecasting.naive import NaiveForecaster
from sktime.forecasting.conformal import ConformalIntervals
from sktime.datasets import load_airline
y = load_airline()  # Use airline data as an example
y_train = y.iloc[:-12]
y_test = y.iloc[-12:]
forecaster = NaiveForecaster(strategy="drift")
# The wrapper handles the train/calibration split internally
# using the 'initial_window' parameter.
conformal_forecaster = ConformalIntervals(
    forecaster,
    initial_window=int(len(y_train) * 0.75),
    method="conformal"  # Use standard absolute residuals
)
# 3. Fit and predict intervals
fh = np.arange(1, 13)
conformal_forecaster.fit(y_train, fh=fh)
#.predict_interval() is now available
pred_intervals = conformal_forecaster.predict_interval(coverage=0.9)
```

Recipe 3: DARTS (native conformal models)

darts ships its own conformal wrappers in `darts.models`. `ConformalNaiveModel` applies split conformal to any darts forecaster (regression-based or neural). `ConformalQRModel` applies conformalized quantile regression on top of a base model that already produces a probabilistic forecast (i.e., was trained with a likelihood). Both stay inside the `darts` API, so the same TimeSeries / fit / predict flow you used in *Chapter 7* carries over.

```python
from darts.models import ConformalNaiveModel, LinearRegressionModel
# Assume 'train_series' is a Darts TimeSeries object.
# 1. Fit a base point forecaster on the training history.
base = LinearRegressionModel(lags=24)
base.fit(train_series)
# 2. Wrap it with the split-conformal wrapper. The quantiles list
# determines what the predict() call will return.
conformal_forecaster = ConformalNaiveModel(
    model=base,
    quantiles=[0.05, 0.5, 0.95],
    cal_length=None,  # use the full training history for calibration
    random_state=42,
)
conformal_forecaster.fit(train_series)
# 3. Predict. num_samples controls the size of the predictive sample
# that darts builds from the calibrated quantiles.forecast = conformal_forecas
# 4. Access the interval bounds.
p05 = forecast.quantile(0.05)
p95 = forecast.quantile(0.95)
```

For a CQR-style adaptive interval, swap `ConformalNaiveModel` for `ConformalQRModel` and give it a base model that supports probabilistic forecasting (e.g., a neural model created with a `QuantileRegression` likelihood, or `LinearRegressionModel(likelihood=quantile, quantiles=[0.05, 0.5, 0.95]))`. The wrapper then conformalizes the base's quantile estimates instead of its point forecast.

The third-party `conformal-tights` library (*Table 8.3*) covers an adjacent need: when you want multiple quantile levels at once with a guarantee that they do not cross. It exposes a `sklearn` meta-estimator and a `DartsForecaster` adapter; the darts adapter targets the post-0.40 darts API, so check its current version against your darts pin before using it.

Conformal prediction for non-exchangeable time series

So far, we have leaned on a convenient assumption: that once a good model strips out trend, seasonality, and exogenous effects, the residuals are approximately exchangeable. That is enough for split conformal and CQR to give valid intervals on well-behaved series like M5 sales.

Time series rarely sit still, though. Consumer behavior shifts, a supplier breaks, a competitor opens across the street. The calibration set you fixed last year is no longer a draw from the same distribution as next quarter. The conformal guarantee $P\big(Y \epsilon C(X)\big) \geq 1 - \alpha$ depends on exchangeability between calibration and test points. When that breaks, coverage drifts down silently, and the rolling dashboard from Production-ready uncertainty: evaluation and monitoring is the signal that catches it.

There are three practical responses:

- **Assume approximate exchangeability after the model removes structure.** This is what the chapter has done so far. It works when drift is mild and the model captures the main patterns; it fails when residuals themselves are autocorrelated or the data-generating process shifts.
- **Use methods designed for dependent data. Ensemble batch Prediction Intervals** (**EnbPI**) and weighted conformal prediction drop the exchangeability requirement and rely on stationarity or weaker conditions.
- **Adapt online. Adaptive Conformal Inference** (**ACI**) treats T as a state variable and adjusts it after every observation, guaranteeing long-run empirical coverage even under arbitrary distribution shift.

EnbPI in practice

EnbPI builds prediction intervals from leave-one-out residuals of a bootstrap ensemble. There is no separate calibration split; every training point contributes to calibration via its out-of-bag predictions. As new ground truth arrives in production, the residual buffer slides forward and intervals update.

The conceptual shift from split conformal: split conformal fixes $\hat{q}$ once at calibration time. EnbPI recomputes the nonconformity score distribution from a moving window of recent OOB residuals, so the interval width changes with the data.

MAPIE provides `TimeSeriesRegressor` for this. The snippet below is an API sketch; the runnable version with the M5 data lives in the companion repo. We use `BlockBootstrap` rather than standard i.i.d. bootstrap so each resample keeps contiguous chunks of the series; sampling individual rows would mix yesterday's residual into tomorrow's calibration set and destroy the temporal dependence we are trying to model around.

```
from mapie.regression import TimeSeriesRegressor
from mapie.subsample import BlockBootstrap
from xgboost import XGBRegressor
cv = BlockBootstrap(n_resamplings=30, n_blocks=10, random_state=42)
mapie = TimeSeriesRegressor(
    estimator=XGBRegressor(random_state=42),
    method="enbpi",
    cv=cv,
    agg_function="mean",
    n_jobs=-1,
)
# Single fit on the full training history; no separate calibration split.
mapie.fit(X_train, y_train)
# Initial intervals on the test set.
y_pred, y_pis = mapie.predict(X_test, alpha=0.1, ensemble=True)
# In production: as true values arrive, slide the residual buffer forward.
for t in range(len(X_test)):
    y_pred_t, y_pis_t = mapie.predict(
        X_test[t:t+1], alpha=0.1, ensemble=True)
    # ... record interval, observe y_true[t] downstream ...
    mapie.partial_fit(X_test[t:t+1], y_true[t:t+1])
```

Two pieces are doing real work. `BlockBootstrap` samples contiguous chunks, preserving temporal structure inside each resample. `partial_fit` is what makes this online rather than batch: the residual buffer slides as you observe new ground truth.

`sktime` has a parallel API via `sktime.forecasting.enbpi.EnbPIForecaster` if you are already in that ecosystem.

ACI in practice

ACI takes a different angle. Instead of changing the calibration mechanism, it changes the target. At each step t define

$$\alpha_{t+1} = \alpha_t + \gamma \cdot (\alpha_0 - err_t)$$

where $err_t = 1$ if y'_t fell outside the interval and 0 if it was covered, and p is a small learning rate (typically 0.005 to 0.05). After a miss, T drops and the next interval widens; after a hit, α inches up and the next interval narrows. The long-run average error rate converges to ρ_k for any sequence of observations, exchangeable or not.

The implementation is a few lines on top of any conformal procedure:

```
alpha = 0.1
alpha_target = 0.1
gamma = 0.05
errors = []
for t in range(len(X_test)):
    y_pred_t, y_pis_t = mapie.predict(
        X_test[t:t+1], alpha=alpha, ensemble=True)
    lower, upper = y_pis_t[0, 0, 0], y_pis_t[0, 1, 0]
    # Observe ground truth (in production, this arrives downstream).
    miss = int(not (lower <= y_true\[t] <= upper))
    errors.append(miss)
    # ACI update
    alpha = alpha + gamma * (alpha_target - miss)
    alpha = max(0.001, min(0.999, alpha))  # clip to a valid range
```

ACI guarantees long-run average coverage; it does not guarantee coverage on any single prediction. That tradeoff is often what production wants, since the failure mode under drift is precisely a slow mis-coverage accumulating over weeks of missed alerts.

The following table summarizes when each method earns its keep.

Scenario	**Method**	**Computational Cost**	**Multi-Horizon Support**
Stationary or mildly drifting data; IID-after-detrending residuals	Split conformal or CQR	Low to moderate	Marginal only; lacks formal calibration guarantee
Stationary block dependencies; batch or online inference	EnbPI	Low to moderate	Requires Bonferroni correction for joint coverage
Arbitrary distribution shift in a long-running production system	ACI on top of EnbPI or split	High (bootstrap ensemble)	Native path-wise coverage via bootstrap residuals
Heavy-tailed financial returns with volatility clustering	GARCH-Conformal hybrid (see *Modeling financial volatility*)	Low overhead	Inherits directly from base method
Feature-dependent	Standard	Moderate	Marginal

uncertainty where formal coverage guarantees are not required	Quantile Regression (no CP)		per step; requires simulation for joint coverage

Table 8.4: A selection guide

Production teams operating under heavy drift, complex dependence structures, or thin calibration data have a larger toolbox than this chapter's framework alone.

Multi-horizon coverage

The library recipes earlier in this chapter set `h=24` for `statsforecast`, `fh=1..12` for `sktime`, and `n=12` for `darts`. We never stopped to ask what coverage means across multiple steps simultaneously.

Consider a 7-day demand forecast. Each daily interval is calibrated to cover its day's true demand with 90% probability, which is marginal coverage at each horizon. The probability that all seven intervals contain the truth simultaneously is much lower. Under independent errors the joint coverage is $0.9^7 \approx 47\%$. Time-series errors are usually correlated, so the true number lands somewhere between 47% and 90%, but never the 90% a naïve reader expects.

This matters whenever the downstream decision depends on the full path rather than any single step. Inventory planning needs cumulative demand over the next week to stay inside a band, not day 3 to have 90% coverage in isolation. Capacity planning needs peak load contained across every hour of the provisioning window, not the peak at any single hour.

Three concepts to keep separate:

- **Marginal coverage** (what every recipe so far returns): for each horizon h in 1..H, P$(Y_{t+h} \in C_{t+h}) \geq 1 - \alpha$. Each interval is valid on its own.
- **Joint coverage**: $P(Y_{t+h} \in C_{t+h}$for all $h = 1..H) \geq 1 - \alpha$. Every step in the forecast covered simultaneously. Stricter, and usually what production decisions want.
- **Path-wise coverage**: define a single score over the whole forecast path (for example, max-residual across horizons) and calibrate against that. Yields joint coverage as a side effect.

The Bonferroni correction is the simplest fix. To get joint coverage $1 - \alpha$ across H horizons, calibrate each step at level α / H:

```
H = 7
alpha_marginal = 0.10
alpha_per_step = alpha_marginal / H   # 0.0143, i.e. 98.57% per-step
y_pred, y_pis = mapie.predict(X_test, alpha=alpha_per_step)
```

The cost is conservatism: Bonferroni intervals are wider than they strictly need to be, since the union bound is loose under correlated errors. For moderate H (say $H \leq 20$) the inflation is acceptable, and the guarantee holds regardless of the error correlation structure.

EnbPI handles the multi-horizon case more naturally. Its calibration set is the full distribution of residuals across all bootstrap models and all horizons, so the empirical quantile already accounts for the typical path-wise error pattern. You do not get a strict joint-coverage guarantee for free, but the intervals are tighter than Bonferroni and marginal coverage at each horizon stays near target.

For any multi-horizon system, log both metrics in production:

```
# Marginal coverage at each horizon h:
marginal_h = ((y_true_h >= y_lower_h) & (y_true_h <= y_upper_h)).mean()
# Joint coverage across the full forecast path:
joint = (
```

```
      (y_true_path >= y_lower_path) & (y_true_path <= y_upper_path)
  ).all(axis=1).mean()
```

If marginal coverage is near 90% but joint sits at 40%, the system is behaving as designed for a marginal-coverage setup; downstream decisions need to account for that. If you specifically need joint coverage, switch to Bonferroni or EnbPI.

Production-ready uncertainty

We can now *generate* adaptive, valid intervals. But in a production system, our job is not done. We must *evaluate* the quality of these intervals and *monitor* them over time for *calibration drift*. Let's begin.

Evaluating interval quality

A valid 99.9% interval that predicts sales between very large bounds is useless. It is *valid* but not *efficient*. We must evaluate our intervals on two criteria: **validity (coverage)** and **efficiency (width)**. Here are some metrics we can use to evaluate our intervals:

Metric	**Description**	**Objective**	**Example Implementation**
Empirical coverage	The actual fraction of test points inside the intervals.	$Target \geq 1 - \alpha$	`mapie.metrics.regression_coverage_score`
Conditional coverage	Evaluates whether coverage remains valid within specific data subgroups (e.g.,	$Target \geq 1 - \alpha$ per group	`df.groupby('feature')['covered'].mean()`

	periods of high volatility).		
Mean interval width	Measures the average tightness of the prediction bands.	Minimize (while maintaining coverage)	`mapie.metrics.regression_mean_width_score`
Winkler score	A proper scoring rule for a single interval. It returns the interval width if covered, and applies a heavy penalty if missed.	Minimize	`sktime.performance_metrics.forecasting.mean_winkler_score`
Interval score	The generalized form of the Winkler score, penalizing width and miscoverage simultaneously (Gneiting & Raftery, 2007).	Minimize	Standard custom function
Pinball loss	Evaluates individual quantile predictions, applying asymmetric penalties for over- or under-estimation relative to the target quantile.	Minimize	`sklearn.metrics.mean_pinball_loss`
CRPS	Evaluates the full	Minimize	`properscoring.crp`

	predictive distribution against the true value; generalizes **Mean Absolute Error** (**MAE**).		`s_ensemble`

Table 8.5: Standard evaluation metrics for prediction intervals

In *The problem with uncertainty* section, we noted that an observation is only anomalous if it falls outside the range of expected values. Our prediction intervals give us that range. An anomaly is any observation where the true value $\hat{y}_{new}$ falls outside its predicted [$O(n^3)$, y_{upper}] bound.

This is why the adaptive CQR intervals are so useful. A simple fixed-width interval would flag many high-sales promotion days as anomalies. The CQR interval, however, widens for promotion days, correctly modeling that high sales are *expected* in that context. An anomaly detected by CQR is more meaningful—it might be a day that had extremely high sales without a promotion.

We can find these anomalies by filtering the `results_df` we created earlier. The covered column identifies any point that fell inside the interval, so an anomaly is any point where `covered == False.`

```
# Assumes results_df is from the CQR model
# and calculate_metrics() has been run
results_df['anomaly'] = ~results_df['covered']
```

We then select the anomalies:

```
anomalies = results_df[results_df['anomaly'] == True]
print(f"Found {len(anomalies)} anomalies out of {len(results_df)} test points.
```

And finally, display a few anomalies to inspect:

```
print(anomalies[['ds', 'Promo', 'y_test', 'y_pred', 'y_lower', 'y_upper']].hea
```

Inspecting these *anomalies* gives us a list of the most surprising events in our test set, which can then be investigated.

Monitoring for calibration drift

This is the most critical MLOps task for uncertainty. The conformal guarantee $S_{cal} = \{s_1, s_2, \ldots s_{n_{cal}}\}$ depends on the exchangeability assumption. For time series, we assume our *model residuals* are exchangeable.

But what happens when the underlying data-generating process changes? This is **distribution drift**. Our model, trained on past data, will start to make systematically different errors. The distribution of its residuals will change, and our calibration set (and its p) will become **stale**. Our intervals will no longer be valid.

The solution is to monitor coverage as a production KPI and recalibrate when it drifts.

Building a rolling coverage dashboard

We can build a simple monitoring dashboard by plotting the empirical coverage over a rolling window.

```
import pandas as pd
# Assume 'results' is a DataFrame with test results and features
results = X_test.copy()
results['y_test'] = y_test
results['y_lower'] = y_lower_cqr
results['y_upper'] = y_upper_cqr
results['ds'] = pd.to_datetime(results['ds']) # Assuming a date column 'ds'
results = results.sort_values('ds')
# 1. Calculate a boolean 'covered' column
results['covered'] = (results['y_test'] >= results['lower']) & \
                     (results['y_test'] <= results['upper'])
# 2. Calculate rolling empirical coverage (e.g., over 100 data points)
# This is our monitoring dashboard metric
rolling_coverage = results['covered'].rolling(window=100).mean()
```

```
# 3. Plot the dashboard
plt.figure(figsize=(12, 6))
rolling_coverage.plot(label='Rolling 100-step Coverage')
plt.axhline(
    y=1.0 - alpha,
    color='r',
    linestyle='--',
    label=f'Target Coverage ({(1.0-alpha)*100:.0f}%)'
)
plt.title('Coverage Monitoring Dashboard')
plt.xlabel('Time')
plt.ylabel('Empirical Coverage')
plt.legend()
plt.show()
```

This plot is our early-warning system.

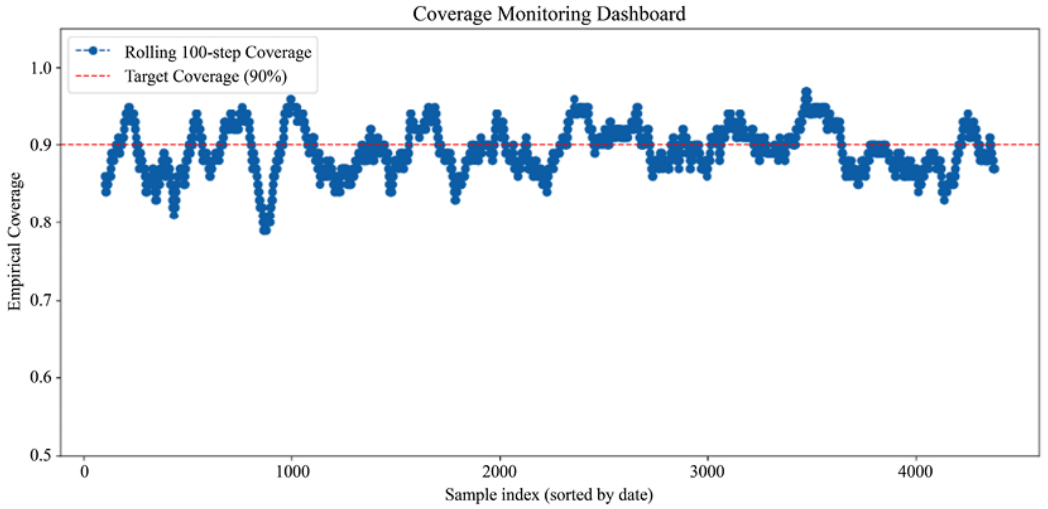


Figure 8.6: Rolling 100-step empirical coverage of the CQR intervals on the test set, plotted against the 90% target line

In the figure, coverage oscillates around the target without sustained dips, which is what you want to see in production. A run of points below the dashed line is the signal to recalibrate.

Recalibration strategy

When the `rolling_coverage` plot dips below the target (e.g., below 85% for a 90% target), our MLOps system must trigger an alert. The fix is **recalibration**.

We must use the most recent data (e.g., the last 1000 data points) as a *new* calibration set, re-run `cqr.fit()` (or `mapie.fit()`), and deploy the new, recalibrated p value. This rolling or online approach is needed for maintaining validity in non-stationary environments.

Diagnosing conditional coverage

We now look at a final, deeper diagnostic because our *marginal* coverage (the rolling plot) might be 90%, but we could have 95% coverage for `cat_id='FOODS'` and 85% for `cat_id='HOBBIES'`. This is a **conditional coverage failure**. Our model is reliable for some subgroups but not others.

We can diagnose this by stratifying our coverage metric.

```
conditional_coverage = results.groupby('Promo')['covered'].mean()
print(conditional_coverage)
```

A representative print on the M5 test set looks like:

Promo	covered
0	0.912
1	0.875

Table 8.6: The M5 test set representative

This result (87.5% for promotion days) tells us our CQR model is *still* not adaptive enough. The solution is to go back to *Recipes for the modern time series ecosystem*, improve our LGBMRegressor models (e.g., add more features, tune hyperparameters), and then recalibrate.

When monitoring tells us that our calibration is stale, we must recalibrate. We can automate this using two primary strategies.

- **Rolling window conformal**: This is the simplest approach. Instead of using a fixed, historical calibration set, you define a sliding window of the most recent data (e.g., the last 1,000 observations). You then re-run the calibration step (e.g., `mapie.fit(X_recent, y_recent)`) on a regular schedule (like daily or weekly) to compute a new p using only this recent data. This ensures your interval width reflects recent model performance, not performance from months ago.
- **Online conformal**: For high-frequency streaming data, you can update your calibration with every new data point. In this method, you maintain a buffer of the most recent T residuals. As each new true value y'_t arrives, you calculate its residual, add it to the buffer, and remove the oldest one. You can then re-compute p on this constantly-updated buffer. This is responsive to sudden changes but can be less stable than a rolling window.

A few rules of thumb for sizing both the window and the cadence. The calibration window should be large enough that q̂ is stable across resamples: at least a few hundred points, and 1,000+ if you can afford it. The recalibration cadence should be much faster than the window: a useful default is to refit every ($window\ size/10$) new points, so a 1,000-point window refits every 100 points. If your monitoring dashboard shows coverage drifting within a single window, the window is too long for the rate of change in the data; shorten it. If p jumps noticeably between refits, the window is too short for the variance in the residuals; lengthen it.

Modeling financial volatility

The heteroskedasticity in the Rossmann data was *feature-driven* (promotions, day of week). In specialized domains like finance or energy, we encounter a different kind: **time-driven** heteroskedasticity, or **volatility clustering**.

This means that a large shock (a big error) *today* makes a large shock *tomorrow* more likely. Volatility is autocorrelated.

This pattern has roots in how information and markets actually behave. Macro news (earnings surprises, central bank decisions, geopolitical events) arrives in bursts, and markets take days or weeks to fully digest it. During stress, liquidity dries up, and forced position unwinds amplify the next day's move, feeding another wave. Energy markets show similar persistence for different reasons: weather patterns, supply disruptions, and demand surprises all play out over multiple sessions. Tomorrow's volatility carries today's regime, which is exactly what an i.i.d. residual assumption misses.

Identifying volatility clustering

We can see this in S&P 500 daily returns (see *Figure 8.7*). The returns themselves are unpredictable, but the *variance* is not. We see clusters of high volatility (e.g., 2008, 2020) and clusters of low volatility. We can prove this by plotting the autocorrelation (ACF) of the *squared returns*. A significant ACF for squared returns is the statistical signature of volatility clustering.

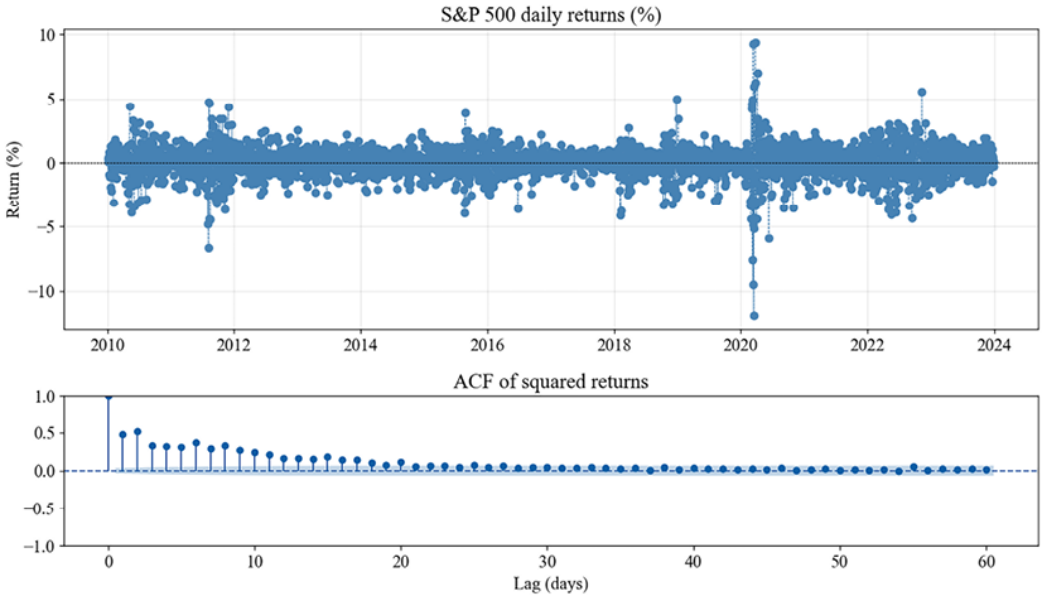


Figure 8.7: S&P 500 daily returns 2010–2024 (top) and the ACF of their squared values (bottom)

The 2020 COVID crash and the 2011 / 2015 / 2018 stress episodes show up as clusters of large-magnitude moves rather than isolated outliers. The squared-

return ACF stays positive and significant for dozens of lags: the statistical signature that volatility is autocorrelated even when the returns themselves are not.

The classical tool for modeling this is the **Generalized Autoregressive Conditional Heteroskedasticity** (**GARCH**) model. A `GARCH(1,1)` model, implemented in the arch library. A GARCH(1,1) writes today's conditional variance as

$$\sigma_t^2 = \omega + \alpha \cdot \varepsilon_{t-1}^2 + \beta \cdot \sigma_{t-1}^2$$

where $\varepsilon_{t-1} = y_{t-1} - \mu_{t-1}$ is yesterday's shock (the residual from the mean model) and σ_{t-1}^2 is yesterday's variance. The three parameters do specific jobs: H is a baseline variance floor, α is how much yesterday's shock kicks up today's variance (the ARCH effect, the reactive part), and $\hat{q}$ is how much yesterday's variance carries over (the GARCH effect, the persistent part). Stationarity requires $\alpha + \beta < 1$; the closer their sum is to 1, the longer a shock takes to fade. That is what produces the visible clusters of high and low volatility: a big shock today raises σ_t^2, which raises σ_{t-1}^2, and the elevated regime decays only as fast as (p, d, q) lets it.

Another advanced approach to modeling this behavior is a **Bayesian Stochastic Volatility** (**SV**) model. Unlike GARCH, which models today's variance as a function of yesterday's variance, an SV model treats the volatility itself as a hidden (latent) autoregressive process.

This can be implemented in libraries like PyMC. This method shares the same critiques as other Bayesian models. The resulting intervals depend on the chosen priors and model structure, and they do not come with the same formal validity guarantee as conformal prediction.

A hybrid model: GARCH combined with conformal prediction

We should not *replace* conformal prediction with GARCH. We should *combine* them. GARCH is an excellent *model* for volatility, but its intervals rely on parametric assumptions (e.g., that residuals follow a Student's t-distribution). CP provides the *guarantee*.

This hybrid *ML-ARCH* approach gives us the best of both:

- **Fit Model:** Fit a `GARCH(1,1)` model to the financial returns (our training set).
- **Get Volatility:** Use the fitted model to predict the conditional volatility γ_0 for each point in our *calibration set*.
- **Standardize Residuals:** Calculate the *standardized residuals* for the calibration set: $z_i = (y_i - \widehat{\mu_i})/\widehat{\sigma_i}$. If the GARCH model is good, these Y_t *should* be homoskedastic and approximately exchangeable. That is a strong claim; see Validating exchangeability of the standardized residuals below before trusting it.
- **Conformalize:** Apply *simple split-conformal prediction* (from *Fixed-width intervals with mapie*) to these *standardized residuals* Y_t. This gives us a fixed-width interval ρ_k that is valid for the *standardized* space.
- **Forecast:** To predict for a new day, use GARCH to forecast the mean $\hat{\mu}_{new}$ and, critically, the volatility x_{new}. The final, hybrid interval is: $C(Hybrid(x_{new}) = [\hat{\mu}_{new} - q_z \cdot \hat{\sigma}_{new}, \hat{\mu}_{new} + q_z \cdot \hat{\sigma}_{new}]$.

As pseudo-code, the whole pipeline is four lines of training and three of inference:

```
# --- Training ---
# 1. fit GARCH on the training set:
garch = GARCH(1, 1).fit(train_returns)
# 2a. get μ̂, σ̂ on the calibration set:
mu_cal, sigma_cal = garch.predict(calib_returns)
# 2b. standardize exchangeable space:
z = (calib_returns - mu_cal) / sigma_cal
# 3. split-conformal quantile on z:
q_z = quantile(abs(z), 1 - alpha)
# --- Inference for a new day ---
```

```
# 4a. GARCH forecast:
mu_new, sigma_new = garch.forecast(horizon=1)
# 4b. scale q_z back by σ^new:
lower = mu_new - q_z * sigma_new
upper = mu_new + q_z * sigma_new
```

This interval is adaptive (it widens and narrows daily with x_{new}) and conformalized (its standardized width ρ_k is guaranteed to be valid).

Validating exchangeability of standardized residuals

The hybrid's conformal guarantee rests on z_i being approximately exchangeable. GARCH removes first-order volatility clustering by construction, but standardized residuals can still carry autocorrelation in higher moments (kurtosis clustering) or non-zero skew that varies through time. Skip this check and you risk inheriting a stale guarantee. Two quick diagnostics, both built on the ACF of the calibration-set Y_t:

- ACF of z_i: should look like white noise. Significant spikes mean GARCH did not fully capture the mean dynamics; refit the mean model before trusting the intervals.
- ACF of z_i^2: should also look like white noise. Significant spikes mean GARCH did not fully absorb the volatility clustering; consider `GARCH(p,q)` with higher orders, EGARCH for asymmetry, or a heavier-tailed conditional distribution.

```
from statsmodels.graphics.tsaplots import plot_acf
import matplotlib.pyplot as plt
# z_calib is the array of standardized residuals on the calibration set
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
plot_acf(z_calib, lags=40, ax=axes[0], title="ACF of z (mean dynamics)")
plot_acf(z_calib ** 2, lags=40, ax=axes[1],
         title="ACF of z² (volatility dynamics)")
plt.tight_layout()
plt.show()
```

The resulting plot is:

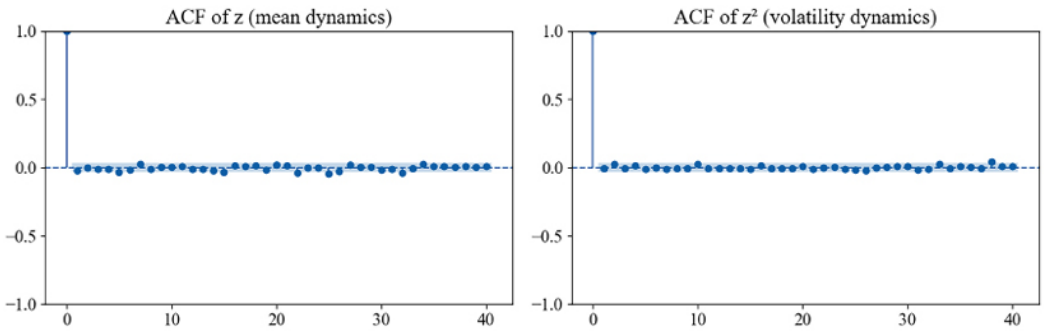


Figure 8.8: ACF of standardized residuals (left) and squared standardized residuals (right) from a GARCH(1,1) fit on 14 years of S&P 500 daily returns

Both panels sit inside the confidence band: `GARCH(1,1)` absorbed the mean and volatility dynamics, and the z_i pass the visual exchangeability check.

If either plot shows persistent spikes outside the confidence band, the exchangeability assumption is shaky and the hybrid interval's coverage will degrade in the same direction as the leftover autocorrelation. The Ljung-Box test on z^2 is a numeric companion to the visual check: a low p-value (< 0.05) confirms remaining volatility clustering.



When implementing volatility models like the GARCH-Conformal hybrid, success depends on the quality of the underlying model. Here are a few practical *gotchas* to ensure your model is reliable:

- Decompose first: Volatility modeling is far more stable after you have removed strong, predictable patterns like trend and seasonality from the data.
- The mean model matters: A poorly specified mean model (e.g., the x_{new} in our GARCH model) will lead to spurious volatility. If the model is bad, its errors (residuals) will be large and patterned, which the volatility model will then incorrectly try to fit.

- Mind the heavy tails: Financial returns often have heavy tails (more extreme events than a normal distribution would suggest). Using a model that assumes a t-distribution is often more robust than assuming a simple normal distribution.
- Backtest your intervals: As we saw in Production-ready uncertainty: evaluation and monitoring, you must backtest your predictive intervals. Compare your nominal coverage (e.g., 90%) to the realized empirical coverage. Use proper scoring rules like the Winkler Score or CRPS to compare different models .

Production and deployment patterns

A model's uncertainty is only useful if it is deployed correctly. A production system involves three distinct pipelines.

- **Training pipeline**: Fits the base model (e.g., `XGBRegressor`) on $p, d, q$and the conformal wrapper (e.g., `ConformalizedQuantileRegressor`) on D_{calib}. The outputs are the saved models and any calibration data (like $\hat{q}$).
- **Inference pipeline**: Loads the saved model(s). When a new `x` arrives, it generates the point forecast and constructs the final interval .
- **Monitoring pipeline**: Collects the true values `y` after they occur. It plots rolling coverage and triggers alerts if coverage drops below the target.

Your model artifacts must include the uncertainty data.

- **Storage**: When you version and save your base model, you must also save its calibration artifacts. For split conformal, this might be a simple JSON file containing the $\hat{q}$ value. For CQR, this would be the two saved quantile models (`estimator_low` and `estimator_high`) and the CQR $\hat{q}$ correction factor.

- **API design**: Your prediction endpoint should return the full interval. Instead of just returning a single number, the API response should be a JSON object that includes the bounds:

```
{
  "point_forecast": 5200.50,
  "lower_bound": 4800.00,
  "upper_bound": 5600.50,
  "confidence_level": 0.90
}
```

Choosing your uncertainty method

Selecting the right method to quantify uncertainty depends on diagnosing the *type* of errors your model produces. You should analyze your model's residuals to see if they are stable, if they change with features, if they cluster over time, or if they change unpredictably.

- **Stable (homoskedastic) errors**: If your model's errors are stable, you can use **Standard Split-Conformal Prediction**. This is the simplest baseline method.
    - **Diagnosis:** You should use this method when your residual diagnostics show no clear patterns. The errors appear random, and their variance is constant. Statistically, a Breusch-Pagan test for heteroskedasticity would have a high p-value (e.g., > 0.05), indicating a lack of evidence for changing variance.
    - **Pros and Cons:** This method is simple, fast, and works with any model. It provides a valid statistical guarantee of coverage. However, its main drawback is that it produces a **fixed-width** interval. If your data *does* have hidden volatility, this single-width interval will be inefficient. It will be too wide for stable periods and dangerously too narrow for volatile periods, leading to under-coverage for specific subgroups.

- **Feature-driven (heteroskedastic) errors**: If your model's errors change based on input features, you should use CQR. This is common in retail, where uncertainty is higher during promotions or holidays.
    - **Diagnosis:** This type of error is often easy to spot visually. You can create a scatter plot of your residuals against your predicted values; a *fanning out* shape indicates that errors get larger as the predicted value increases. You can also plot boxplots of the residuals grouped by a specific feature, like Promo. If the boxes have different sizes, your uncertainty is feature-driven.
    - **Pros and Cons:** CQR solves the main problem of fixed-width intervals by creating adaptive intervals. The interval width will automatically get wider when a promotion is active and narrower on normal days. This provides more uniform coverage across all subgroups and is more efficient. The main trade-off is a small increase in complexity, as it requires training and maintaining two separate quantile models instead of one mean-prediction model.
- **Time-driven (volatile) errors**: If your data shows volatility clustering, you should use a **Hybrid GARCH-Conformal Model**. This is highly specific to domains like finance or energy, where a large shock today makes another large shock tomorrow more likely.
    - **Diagnosis:** This pattern is identified by time, not features. You will see clusters of high volatility followed by clusters of low volatility in your data . The formal statistical test is to plot the **Autocorrelation Function** (**ACF**) of the squared residuals. If this plot shows significant spikes, it means your volatility is autocorrelated .
    - **Pros and Cons:** This hybrid model uses the GARCH model to explicitly forecast the next day's volatility, and then uses that forecast to scale the conformal interval. This creates an interval that is adaptive to time. The primary con is its high complexity; it is a

specialized tool and should not be used unless you have clear evidence of volatility clustering.

- **Unstable errors** (**distribution drift**): If your data distribution changes over time, you should use one of the methods from *Conformal prediction for non-exchangeable time series*. This is common in high-frequency streaming data where patterns are not stationary.
    - **Diagnosis**: Monitor your intervals in production. If empirical coverage over a rolling window dips frequently and unpredictably below your target (e.g., 90%), your original calibration set is stale and no longer represents the new data.
    - **Pros and Cons**: Three options, each with a different trade-off. Rolling-window recalibration refits $\hat{q}$ on a sliding window of recent residuals on a schedule; it bolts onto any CP method but has no formal guarantee under arbitrary shift, and requires tuning the window size and cadence. EnbPI replaces the train/calibration split with bootstrap OOB residuals; it handles dependent residuals natively but costs B bootstrap models. ACI leaves the calibration mechanism fixed and adapts α after every observation; it is provably valid in the long run for any sequence of data, but says nothing about coverage on any single prediction.

The four regimes above are the same four leaves of Figure 8.1. If you skipped the roadmap at the top of the chapter, this is the moment to flip back to it: the decision tree turns the diagnostic questions into a single visual.

By following this framework, you can move beyond simple point forecasts and build forecasting systems that quantify their own uncertainty.

Summary

In this chapter, we moved beyond the limits of point forecasts. You learned that for real-world decisions, a *best guess* is not enough; we need a reliable measure of its uncertainty.

We built a robust framework using Conformal Prediction, a model-agnostic wrapper that provides a statistical guarantee of coverage. You started with simple, fixed-width intervals (Split Conformal) and then progressed to adaptive intervals (CQR) to handle feature-driven uncertainty. You also learned the MLOps essentials for evaluating, monitoring, and recalibrating these intervals in production to handle distribution drift. Finally, we looked at specialized hybrid models like GARCH-Conformal for financial data.

You now have a complete system for generating a forecast and quantifying its uncertainty. The chapter's worked examples use the M5 Forecasting Accuracy dataset (`id`, `Date`, `sales`, `cat_id`, `store_id`, `snap_active`, `DayOfWeek`, plus the engineered Promo flag derived from `sell_price` markdowns) and 14 years of S&P 500 daily returns. M5 is hosted on Zenodo (https://zenodo.org/records/12636070) under CC-BY 4.0; the companion repo's chapter5/download_m5.py fetches it and the *Chapter 5* prep notebook lands it in the schema *Chapter 8* assumes. The S&P 500 returns are pulled at runtime via `yfinance` in `gen_chapter8_figures.py`.

For practical deployment, one should establish a baseline with split conformal prediction, which requires minimal infrastructure modification. If diagnostics reveal heteroskedasticity, conformalized quantile regression offers a natural progression. Highly specialized tools, such as GARCH-conformal hybrids, are best reserved for financial domains where phenomena like volatility clustering justify their computational expense.

In the next chapter, we will explore the new frontier of time series: foundation models. You will learn a pragmatic guide to using pre-trained models like Chronos and `TimeGPT` for zero-shot forecasting, as well as how to use Large Language Models (LLMs) to extract qualitative insights.

Further Reading

- Angelopoulos, Anastasios N., Stephen Bates, Michael I. Jordan, and Jitendra Malik. 2024. "*Conformal PID Control for Time Series Prediction*."

arXiv. https://arxiv.org/abs/2307.16895.

- Barber, Rina Foygel, Emmanuel J. Candès, Aaditya Ramdas, and Ryan J. Tibshirani. 2021. "*Predictive Inference with the Jackknife+.*" Annals of Statistics 49 (1): 486–507. https://arxiv.org/abs/1905.02928.
- Barber, Rina Foygel, Emmanuel J. Candès, Aaditya Ramdas, and Ryan J. Tibshirani. 2023. "*Conformal Prediction Beyond Exchangeability*." arXiv. https://arxiv.org/abs/2202.13415.
- Chernozhukov, Victor, Kaspar Wüthrich, and Yinchu Zhu. 2018. "*Exact and Robust Conformal Inference Methods for Predictive Machine Learning with Dependent Data*." arXiv. https://arxiv.org/abs/1802.06300.
- Gibbs, Isaac, and Emmanuel J. Candès. 2021. "*Adaptive Conformal Inference Under Distribution Shift.*" arXiv. https://arxiv.org/abs/2106.00170.
- Tibshirani, Ryan J., Rina Foygel Barber, Emmanuel J. Candès, and Aaditya Ramdas. 2019. "*Conformal Prediction Under Covariate Shift.*" arXiv. https://arxiv.org/abs/1904.06019.
- Xu, Chen, and Yao Xie. 2021. "*Conformal Prediction Interval for Dynamic Time-Series.*" arXiv. https://arxiv.org/abs/2010.09107.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers.

https://discord.gg/VXpz9YPUTW



9

Foundation Models: Quantitative and Qualitative Forecasting

For the past eight chapters, we have operated under a consistent constraint: to predict the future, we must meticulously train a model on the past. Whether building statistical baselines using ARIMA or training *global* deep learning architectures like DeepAR, the workflow has remained the same: gather data, engineer features, tune hyperparameters, and train a specific model for a specific task.

This *train-from-scratch* paradigm is rigorous, but it is a bottleneck. In the field of **Natural Language Processing** (**NLP**), this workflow was upended years ago. Instead of training a new model to classify sentiment or summarize text, practitioners now rely on **Large Language Models** (**LLMs**) that have learned the universal structure of language from internet-scale datasets.

A similar shift has finally arrived for time series. We have entered the era of the TSFMs. By pretraining massive transformer architectures on billions of real-world and synthetic data points, researchers have created models that understand the general *grammar* of time series—trends, seasonality, and noise—before they ever see your specific data.

The overall goal of this chapter is to equip you with both the conceptual understanding and practical skills needed to use foundation models effectively. You will learn how to apply tools such as TimesFM and Chronos for instant forecasting, understand their architectural differences, and explore multimodal workflows that combine time series data with unstructured text for richer insights.

This knowledge is important because it reflects a major industry shift toward scalable, inference-driven forecasting systems. For data professionals, this means faster experimentation, improved performance in cold-start scenarios, and the ability to build more intelligent, context-aware forecasting solutions that align with modern real-world applications.

In this chapter:

- A taxonomy of foundation-model architectures
- The need for time series foundation models
- Quantitative foundation models
- Qualitative and multimodal forecasting
- Critical challenges

A taxonomy of foundation model architectures

It helps to organize the design space along two axes rather than treating every model as a unique black box: the backbone architecture (Transformer vs. non-Transformer) and the input representation

(tokenized vs. continuous patches).

Most TSFMs in production today share the same backbone, the Transformer, and differ mainly in how they convert raw values into something the model can ingest.

- **Tokenized inputs (Chronos, Time-LLM)**: Continuous values are quantized into a fixed vocabulary, and an off-the-shelf language model (T5, Llama) treats forecasting as next-token prediction. This gives natural probabilistic outputs but pays a quantization cost on precision.
- **Continuous patches (TimesFM, Moirai)**: Time steps are grouped into short patches of real numbers and fed straight into the Transformer, avoiding the quantization step. Patching is the standard trick for handling long histories without exploding the context length. TimesFM and Moirai 2.0 are decoder-only patch models. Moirai 2.0 replaces its prior masked encoder design to enhance inference efficiency

A second, smaller group steps away from Transformers entirely to cut latency.

- **MLP-based mixers (e.g., Tiny Time Mixers)**: Replace self-attention with channel- and time-mixing MLP blocks. Cheap and fast, with surprisingly competitive accuracy on shorter horizons.
- **State-space models (e.g., TSMamba, S-Mamba)**: Use selective linear-recurrence operators instead of attention. They scale linearly in sequence length and are well-suited to very long contexts.

These two non-Transformer families share a motivation (efficiency) but rest on different mathematical foundations; *MLP* and *SSM* are not interchangeable.

Most models focus strictly on numerical data. This section introduces systems that integrate textual context to explain the reasoning behind a forecast.

Foundation models bring one capability that no statistical model in the previous chapters possessed: the ability to read. Standard forecasting models are blind to the context that drives them—a CEO's resignation letter, a hurricane warning, or a competitor's press release.

We are now seeing the rise of **Time-Series Language Models** (**TSLMs**) and multimodal workflows. By combining quantitative forecasters (like TimesFM) with reasoning engines (like GPT-4), we can build systems that don't just predict that sales will drop, but explain why, citing the breaking news report that just hit the wire.

With the architectural map in hand, we now turn to why a practitioner would reach for one of these models in the first place and when they should not.

The need for time series foundation models

Traditional forecasting requires building and maintaining a separate model per dataset: feature engineering, tuning, retraining. That cost is manageable for a handful of series and unmanageable for thousands. TSFMs trade per-dataset training for inference against a pre-trained model that has already absorbed trend, seasonality, and shock patterns from billions of points. The promise is short setup time, instant baselines, and useful forecasts in cold-start situations where there is barely any history to fit on.

The claim that *TSFMs replace traditional forecasting* is the wrong question. The right question is when each approach wins. Let's measure it.

A hands-on example

We will use the AirPassengers dataset (monthly international airline passengers, 1949–1960) used later in the Chronos section, hold out the last 24 months, and forecast them with three approaches:

- **Seasonal Naive**: repeat the last full year of training data.
- **LightGBM** with five lag features (t-1, t-2, t-3, t-12, t-24) plus a one-hot month indicator. Trained on a sliding window of the first 120 months.
- **Chronos-Tiny** zero-shot: an 8M-parameter foundation model loaded from Hugging Face, no fine-tuning. CPU inference; no GPU required.

Each is scored with three error metrics: **Mean Absolute Percentage Error** (**MAPE**), **Symmetric MAPE** (**sMAPE**; bounded 0–200%), and **Mean Absolute Scaled Error** (**MASE**; which divides MAE by the in-sample seasonal-naive MAE; values < 1 mean you beat seasonal-naive on the training fit). The *Evaluating foundation-model forecasts* subsection later in this chapter gives the formulas and a Python helper; this section uses them as black boxes.

```
import lightgbm as lgb
import numpy as np
import pandas as pd
import torch
from chronos import ChronosPipeline
URL = ("https://raw.githubusercontent.com/AileenNielsen/"
       "TimeSeriesAnalysisWithPython/master/data/AirPassengers.csv")
y = pd.read_csv(URL)["#Passengers"].values.astype(float)
H, M = 24, 12
y_train, y_test = y[:-H], y[-H:]
# 1. Seasonal Naive
seasonal_naive = np.tile(y_train[-M:], H // M + 1)[:H]
# 2. LightGBM with lag + month features
def make_features(values, lags=(1, 2, 3, 12, 24), m=12):
    X, y_out, months = [], [], []
    for i in range(max(lags), len(values)):
        X.append([values[i - L] for L in lags])
        y_out.append(values[i])
        months.append(i % m)
    return np.hstack([np.array(X), np.eye(m)[months]]), np.array(y_out)
X_train, yt = make_features(y_train)
booster = lgb.train(
    {"objective": "regression", "learning_rate": 0.05, "num_leaves": 15,
     "min_data_in_leaf": 4, "feature_fraction": 0.9,
     "verbosity": -1, "seed": 0},
    lgb.Dataset(X_train, label=yt),
    num_boost_round=400,
)
lags = (1, 2, 3, 12, 24)
history = list(y_train)
gbm_forecast = []
for _ in range(H):
    idx = len(history)
    row = np.concatenate([[history[idx - L] for L in lags],
                          np.eye(12)[idx % 12]])
    pred = float(booster.predict(row.reshape(1, -1))[0])
    gbm_forecast.append(pred)
    history.append(pred)
gbm_forecast = np.array(gbm_forecast)
# 3. Chronos-Tiny zero-shot
pipeline = ChronosPipeline.from_pretrained(
```

```
    "amazon/chronos-t5-tiny", device_map="cpu", torch_dtype=torch.float32,
)
samples = pipeline.predict(
    inputs=torch.tensor(y_train, dtype=torch.float32),
    prediction_length=H, num_samples=100,
)
chronos_forecast = samples.numpy().squeeze(0).mean(axis=0)
```



Please note that we import `lightgbm` before `torch` to avoid a `libomp double-load segfault` on macOS.

The full script (with the three metric helpers) lives in the `baseline_benchmark.py` file in the companion repo. The `Quantitative Foundation Models.ipynb` notebook reproduces the same comparison using the same native libraries (`chronos.ChronosPipeline`, `timesfm.TimesFM_2p5_200M_torch`) and adds Google's `TimesFM-2.5` as a fourth row. Running the script produces:

Model	**MAPE**	**sMAPE**	**MASE**
Seasonal Naive	15.52	17.01	2.494
LightGBM (lag + month)	10.96	11.89	1.815
Chronos-Tiny zero-shot	14.45	15.85	2.357

Table 9.1: 24-month hold-out scores on AirPassengers. Lower is better

Three observations the table makes vivid:

1. Zero-shot beats the naive baseline. Chronos-Tiny, with no training on this series and no awareness of *monthly airline data,* produces a forecast meaningfully better than seasonal-naive (MAPE 14.45 vs. 15.52). This is the cold-start promise: useful output from minute one.
2. The tuned baseline beats zero-shot. LightGBM with five lag features and a one-hot month vector wins on every metric, and by a wide margin (MAPE 10.96, MASE 1.815). On a single series with twelve years of clean monthly history, the foundation model is not the right tool.
3. All three have MASE > 1. The denominator is the in-sample naive error, which is small because AirPassengers is highly regular in training. The test years have a sharper trend than the model's training window, so every approach pays a tax. The metric is doing its job by flagging that.

The reader's mental model should be: this is the worst case for a TSFM. One series, long stable history, strong seasonality the baseline can lock onto, and a small zero-shot model. The TSFM advantage grows as you add series (cold-start, thousands of SKUs), shrink history (a new product launched last month), or need probabilistic outputs without engineering a quantile-aware baseline. Andy Naqvi's warning, often called the **Baseline Illusion**, is the corollary: a TSFM that wins on a leaderboard against an untuned baseline is winning against a strawman.

This single benchmark is not a verdict on Chronos or on foundation models broadly. The point is the discipline: before you ship a TSFM, run this comparison on your data. The decision flowchart in the

next section formalizes the rule.

Quantitative foundation models

Before we dive into the code for specific architectures, it is vital to map the landscape. Not all foundation models are built equally; they make distinct trade-offs between computational efficiency, input handling (continuous vs. tokenized), and probabilistic capabilities.

The *Best For* column in *Table 9.2* maps the common use cases to specific models. A caveat on the SSM row: as of writing, TSMamba exists primarily as a research paper rather than as a packaged library. Related implementations (TimesMamba, MambaTS, S-Mamba, D-Mamba) are scattered across research repositories with varying code quality; treat this family as exploratory rather than production-ready until that situation matures.

Use *Table 9.2* to select the starting point for your use case.

Model Name	**Primary Architecture**	**Key Mechanism / Input Handling**	**Best For**	**Typical Parameter Counts**	**Status**	**License**
TimesFM (Google)	Transformer (Decoder-only)	Continuous Patching: Breaks time series into patches of real values; predicts future patches directly.	High-precision continuous values (stock prices, sensor readings); point + quantile forecasts.	~200M (e.g., timesfm-2.5-200M)	Production (BigQuery ML integration)	Apache
Chronos/Chronos-Bolt (Amazon)	Transformer (Encoder-only)	Tokenization: Quantizes values into a fixed vocabulary, treating forecasting as next-token language prediction.	Probabilistic forecasts with sampled paths (safety stock, risk); multi-modal demand. Bolt for low latency.	~8M (Tiny) to ~710M (Large)	Production (Bolt = production-optimized)	Apache

Moirai (Salesforce)	Transformer (Decoder-only)	Universal Patching: Uses *any-variate* attention to handle varying frequencies and variable counts natively.	Multivariate problems with known future covariates (price, weather, promos).	~14M (Small) to ~300M (Large)	Production (Small/Base); Emerging (R-Large)	Apache
Tiny Time Mixers (IBM)	MLP-Mixer (no attention)	Channel- and time-mixing MLP blocks; very fast inference.	Edge deployment, IoT, short horizons where latency matters more than accuracy.	~1M to ~5M	Emerging	Apache
TSMamba / S-Mamba	State Space Model (SSM / Mamba)	Linear Recurrence: A non-Transformer architecture offering O(N) linear computational complexity for long sequences.	Very long contexts (years of high-frequency data) where Transformer attention is too expensive.	~5M to ~50M (research weights)	Research-stage (code availability varies)	Various (MIT/A

Table 9.2: Strategic comparison of key open-weight time-series foundation models. Status reflects whether stable open weights, a packaged library, and at least one production-grade integration exist as of early 2026

Before we pick a model we need a yardstick. Three error metrics show up in almost every TSFM paper and benchmark, and they each tell you something different:

- **MAPE**: Easy to explain to stakeholders (we were off by 7% on average), but blows up when actuals are near zero and is asymmetric: under-forecasts and over-forecasts of the same magnitude get different penalties

$$\frac{1}{n}\sum\frac{(|y-\hat{y}|)}{|y|} * 100$$

- **sMAPE**: Bounded between 0 and 200%, handles zero actuals more gracefully, and is the metric the M4 / M5 competitions used. Still has a quirk: it penalizes over- and under-forecasts asymmetrically when one of y or $\hat{y}$ is zero.

$$mean\ (2 * \frac{|y - \hat{y}|}{(|y| + |\hat{y}|)}) * 100$$

- **MASE**: Divides MAE by the in-sample MAE of a naive forecast ($y_t = y_{t-m}$ where m is the seasonal period). A MASE of 1.0 means *no better than seasonal-naive*; below 1.0 means you beat the naive baseline. MASE is scale-free, comparable across series, and the metric that surfaces the *did this expensive model actually help?* question.

$$MASE = \frac{MAE\ of\ model}{MAE\ of\ seasonal\ naive}$$

The code for these three metrics is:

```
import numpy as np
def mape(y, yhat):
    return np.mean(np.abs((y - yhat) / y)) * 100
def smape(y, yhat):
    return np.mean(2 * np.abs(y - yhat) / (np.abs(y) + np.abs(yhat))) * 100
def mase(y, yhat, y_train, m=12):
    naive_err = np.mean(np.abs(y_train[m:] - y_train[:-m]))
    return np.mean(np.abs(y - yhat)) / naive_err
```

Report all three when comparing TSFMs to baselines. MAPE/sMAPE tell stakeholders the user-facing error; MASE tells you whether you've earned the right to deploy the larger model. A TSFM that *wins* on MAPE but has MASE > 1 is losing to seasonal-naive in disguise.

A note on when not to reach for a foundation model: the decision flowchart's left branch (Have you tuned a baseline? Does the baseline meet your target?) and the Performance risks section later in this chapter cover the negative case in detail. Short version: if you have years of stable history on a single series with strong seasonality, a tuned Seasonal-Naive + LightGBM stack is usually cheaper, faster, easier to debug, and roughly as accurate.

Table 9.2 lists what each model is; the harder question is when to reach for one. The decision flowchart in *Figure 9.1* makes the trade-offs explicit. Read it top to bottom: the first two checks send you back to a baseline rather than to a foundation model, which is the right answer more often than people expect.

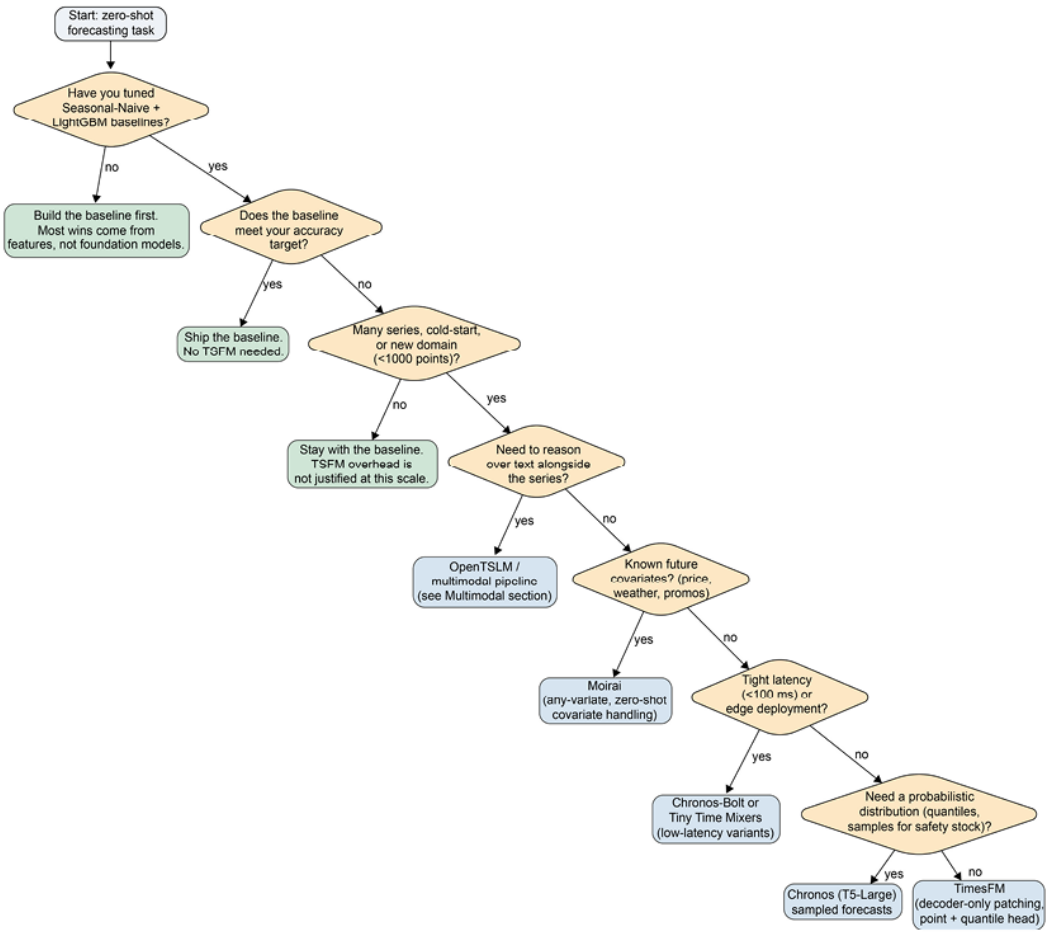


Figure 9.1: A decision flowchart for choosing between a tuned baseline and the major foundation models.

The flowchart enforces three habits worth calling out:

- **Baseline first.** The first two diamonds (**Have you tuned ...?** and **Does the baseline meet your target?**) exist because zero-shot scores look impressive only against untuned baselines. A tuned Seasonal-Naive plus a LightGBM with lag and calendar features beats most TSFMs on stable, data-rich problems, and is far cheaper to ship and monitor.
- **Cold-start is the real TSFM win.** The *many series / <1000 points* branch is where foundation models genuinely shine: new SKUs, new sensors, brand-new geographies. If you have years of history and a single series, the cost-benefit shifts back toward the baseline.
- **Pick on capabilities, not parameter count.** The bottom half of the tree routes by what you need from the output: text reasoning (OpenTSLM), known covariates (Moirai), low latency (Chronos-Bolt /TTM), probabilistic samples (Chronos), or point + quantile forecasts (TimesFM). A 700M-parameter Chronos-Large is not *better* than a 200M TimesFM; they answer different questions.

In this section, we will examine the *Big Three* architectures that currently define the modernity of quantitative foundation modeling. Each represents a distinct philosophical approach to the problem of processing continuous temporal data with neural networks originally designed for discrete text.

We begin our technical review with a model designed for continuous data. This subsection explores how Google adapts the transformer architecture for time series.

Google TimesFM

TimesFM (**Time Series Foundation Model**) represents Google Research's effort to translate the decoder-only transformer architecture, the engine behind GPT models, directly to time series forecasting. *Decoder-only* simply means the network has one stack of layers that reads its own prior outputs as additional context; there is no separate encoder reading a different modality. It is pretrained on a massive corpus of over **100 billion real-world and synthetic time points**, including data from Google Trends, Wikipedia page views, and the Monash Time Series Repository.

The core challenge in applying transformers to time series is the *context length explosion.* If every single time point (e.g., a minute of data) is treated as a token, a standard transformer context window of 4096 tokens would only cover less than 3 days of history. TimesFM solves this via **patching.** . In patching, the input sequence is split into non-overlapping chunks of length T (for example, 32 time steps), reducing the effective sequence length and enabling the model to process much longer histories.

TimesFM 2.5 supports context lengths up to 2048 patches, which, with a patch size of 32, allows the model to process over 65,000 time points of history.

It's also important to mention that TimesFM is strictly autoregressive. It predicts the *next patch* given the sequence of previous patches. This aligns perfectly with the forecasting task, which is fundamentally about extrapolating the future from the past.

Foundation models must handle data of vastly different scales, from stock prices ($150) to GDP ($20 Trillion). TimesFM employs standard normalization (often **Reversible Instance Normalization** (**RevIN**)), where the context is scaled by the mean and standard deviation of the input window. This allows the model to learn scale-invariant patterns.

This diagram illustrates the TimesFM architecture as a continuous autoregressive engine. The process begins on the left where the raw time series is sliced into non-overlapping patches to reduce the effective sequence length. These patches pass through a **RevIN** layer before entering the transformer decoder stack, which uses a causal mask to preserve temporal order. The visualization highlights the autoregressive loop on the right where the predicted output patch is fed back into the model to generate the subsequent step in the sequence.

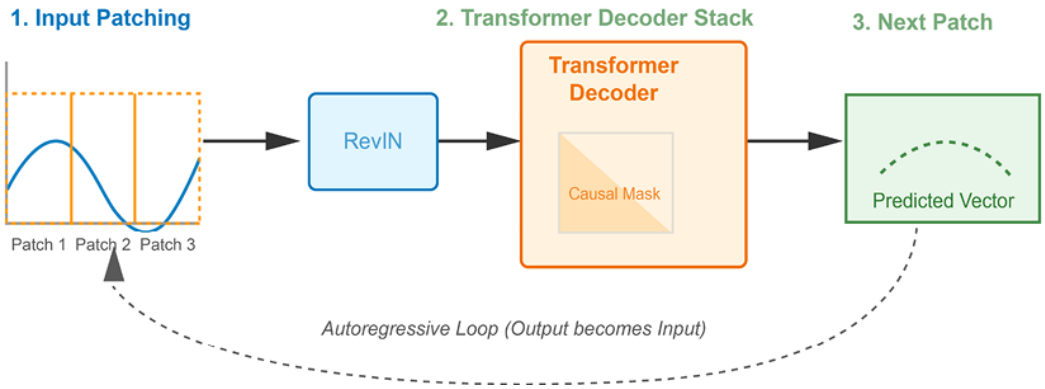


Figure 9.2: The TimesFM architecture

To demonstrate TimesFM, we will simulate a retail scenario using the `timesfm` Python library. We will forecast the demand for two distinct SKUs: a seasonal product (e.g., swimwear) and a trending product (e.g., a new electronic gadget).

1. The first step is setting up the environment. TimesFM requires specific dependency management. It is best installed in a `Python 3.10` environment to avoid package conflicts.

```
# Create and activate a dedicated environment
conda create -n timesfm_env python=3.10
conda activate timesfm_env
# Install TimesFM with Torch backend (JAX is also supported)
pip install timesfm[torch]
```

2. The next step is initializing the pre-trained model and compiling it with a forecasting configuration `ForecastConfig`. Compilation is important because it sets the model's operating constraints, such as how much history it can use and how far into the future it can predict, while also optimizing execution for performance.

```
import timesfm
import torch
import numpy as np
import matplotlib.pyplot as plt
# Initialize the 200 Million parameter model (v2.5)
# This model is small enough to run on a standard GPU or even high-end CPU
model = timesfm.TimesFM_2p5_200M_torch.from_pretrained(
    "google/timesfm-2.5-200m-pytorch",
    torch_compile=True  # Optimizes the backend (Inductor/XLA)
)
# Configure the Forecasting Constraints
# max_context: Maximum history to look back (in patches).
# max_horizon: The forecasting horizon needed.
model.compile(timesfm.ForecastConfig(
    max_context=1024,        # 1024 patches allows for extensive history
    max_horizon=96,          # Common horizon for M4/M5 benchmarks
    normalize_inputs=True,   # Critical for zero-shot generalization
    use_continuous_quantile_head=True, # Enable probabilistic
                                       # forecasting
    force_flip_invariance=True, # Augmentation: model sees flipped
                                # series during inference
    infer_is_positive=True,  # Constraint: Demand cannot be negative
    fix_quantile_crossing=True, # Enforce statistical validity
                                # (P10 <= P50 <= P90)
))
print("TimesFM 2.5 Compiled and Ready.")
```

3. We then generate synthetic data to mimic a *cold start* problem where we have history but no trained model for these specific series. Basically, this is where the model must generate forecasts for unseen time series without any additional training. This highlights one of the key advantages of foundation models.

```
# Generate Synthetic Retail Data
# Series 1: Seasonal (e.g., Summer Clothing)
t = np.linspace(0, 50, 500)
```

```
series_seasonal = 10 * np.sin(t) + np.random.normal(0, 1, 500) + 20
# Series 2: Strong Linear Trend (e.g., New Viral Product)
series_trend = (
    np.linspace(10, 100, 500) + np.random.normal(0, 2, 500))
# Input format: List of numpy arrays
input_data = [series_seasonal, series_trend]
# Perform Zero-Shot Inference
# The model creates a forecast for the next 96 steps automatically
point_forecast, quantile_forecast = model.forecast(
    horizon=96,
    inputs=input_data
)
# Shapes:
# point_forecast: (Batch_Size, Horizon) -> (2, 96)
# quantile_forecast: (Batch_Size, Horizon, Quantiles) -> (2, 96, 10)
# Quantiles usually map to [Mean, 0.1, 0.2,..., 0.9]
```

4. Finally, we visualize both the point forecasts and the uncertainty intervals to understand how the model behaves across different patterns. The seasonal subplot should show the blue mean tracking the sine wave with a narrow uncertainty band; the trend subplot should show the green mean continuing the linear growth with a wider band that grows over the horizon, reflecting the model's higher uncertainty when extrapolating a trend it has not seen plateau.

```
# Visualization Logic
plt.figure(figsize=(15, 6))
# Plot Seasonal Series
plt.subplot(1, 2, 1)
plt.plot(range(500), series_seasonal,
         label="History", color='black', alpha=0.7)
plt.plot(range(500, 596), point_forecast[0],
         label="Forecast (Mean)", color='blue')
# Shade the 10th-90th percentile confidence interval
plt.fill_between(
    range(500, 596),
    quantile_forecast[0, :, 1], # 10th percentile
    quantile_forecast[0, :, 9], # 90th percentile
    color='blue', alpha=0.2, label="80% Confidence Interval"
)
plt.title("TimesFM: Seasonal Item (Zero-Shot)")
plt.legend()
# Plot Trend Series
plt.subplot(1, 2, 2)
plt.plot(range(500), series_trend,
         label="History", color='black', alpha=0.7)
plt.plot(range(500, 596), point_forecast[1],
         label="Forecast (Mean)", color='green')
plt.fill_between(
    range(500, 596),
    quantile_forecast[1, :, 1],
    quantile_forecast[1, :, 9],
    color='green', alpha=0.2, label="80% Confidence Interval"
)
plt.title("TimesFM: Trending Item (Zero-Shot)")
plt.legend()
plt.tight_layout()
plt.show()
```

The previous example demonstrates how TimesFM can generate forecasts in a zero-shot setting with minimal setup. In practice, however, these models are often deployed at scale, where thousands or even

millions of time series must be processed efficiently. This is where enterprise integrations become important.

For enterprise deployments, Google has integrated TimesFM directly into **BigQuery ML**. This allows data engineers to generate forecasts using SQL without ever moving data out of the data warehouse.

In this setup, forecasting becomes part of the existing data pipeline rather than a separate modeling workflow. Users can call `ML.FORECAST` on a model type `TIMESFM` directly on BigQuery tables, making it accessible for teams that are already comfortable working in SQL. At the same time, BigQuery manages the batching and parallelization automatically, scaling to millions of time series. For example, SKU-level forecasting for a global retailer.

As with every other approach, this too has its limitations. TimesFM is univariate and models each series in isolation. If the *trend* series we just looked at was driven by an external marketing budget (a covariate), standard TimesFM cannot explicitly ingest that covariate channel.



Running a 700M parameter model for 10,000 SKUs is expensive. Use FMs for *'Head'* items (high value) and statistical baselines (*Chapter 3*) for *'Tail'* items.

While TimesFM handles continuous values, other models take a different approach. We now examine an architecture that treats forecasting as a language translation task.

Amazon Chronos

Amazon Chronos approaches the problem from a fundamentally different angle. Rather than modifying the transformer architecture to handle continuous numbers (like TimesFM), Chronos modifies the data to fit standard language models. It treats time series forecasting as a translation task: translating a sequence of historical *tokens* into a sequence of future *tokens*.

Chronos-2 operates on an encoder-only architecture. It shifts away from its earlier T5 framework to improve performance.

- **Scaling**: The time series context is scaled (e.g., mean-scaled) to a normalized range.
- **Quantization**: The continuous values are binned into a fixed vocabulary of tokens (e.g., 4096 bins).
- **Language Modeling**: The T5 model processes these tokens using standard categorical cross-entropy loss. It learns the probability distribution of the *next token* given the history.

This approach allows Chronos to model arbitrary distributions. Unlike a Gaussian output head (common in DeepAR), Chronos can predict multi-modal distributions, essential for erratic data like slow-moving inventory where demand is either 0 or 10, but never 5.

The Amazon Chronos architecture functions as a translation pipeline that converts time into language. Unlike standard forecasting models that process continuous numbers, the input stage on the left shows the data being quantized into discrete bins and mapped to specific text tokens. These tokens are processed by the T5 encoder-decoder block in the center. The output on the right is depicted not as a single numerical value but as a softmax probability distribution over the vocabulary. This requires a sampling step to produce the final probabilistic forecast. This can be seen in *Figure 9.3*.

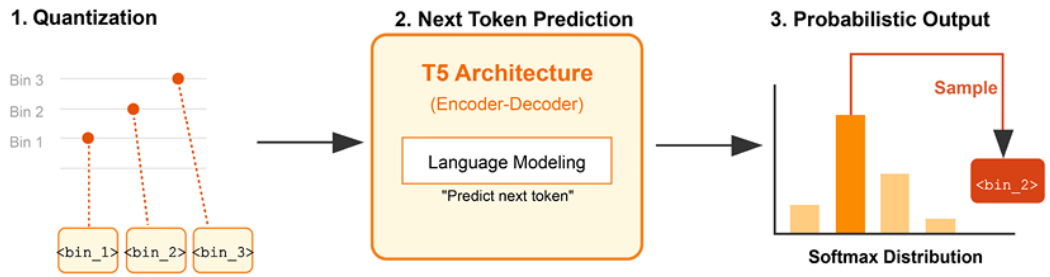


Figure 9.3: Amazon Chronos architecture

Amazon offers two distinct flavors of Chronos, catering to different production needs:

Feature	**Chronos (Classic)**	**Chronos-Bolt**
Architecture	Autoregressive (Token-by-Token)	Direct Multi-Step (Encoder-Decoder)
Speed	Slow (High Latency)	Fast (~250x faster)
Memory	High Footprint	Efficient
Best Use Case	High-precision, low-volume tasks	High-volume, real-time tasks (e.g., IoT)
Multivariate	No (univariate only)	No (univariate only)

Table 9.3: Chronos Classic vs. Chronos-Bolt

Chronos-Bolt is particularly notable for high-frequency finance or operational monitoring applications where latency is critical. Instead of generating the forecast one step at a time, it outputs the entire distribution for the forecast horizon in a single forward pass.

With some theory understood, let's move onto a practical implementation. We will use the `chronos-forecasting` library to predict monthly passenger numbers (a proxy for travel demand) using the high-accuracy T5-Large model.

1. The first step as always is installing this new library

```
pip install chronos-forecasting
```

2. Next, we load a pretrained Chronos model and prepare the input data. Chronos models operate on univariate sequences and expect the input as a PyTorch tensor.

```
import torch
import pandas as pd
import numpy as np
from chronos import ChronosPipeline
# Load the "Large" model (710M parameters)
# This model provides the highest zero-shot accuracy in the family
pipeline = ChronosPipeline.from_pretrained(
```

```
    "amazon/chronos-t5-large",
    device_map="cuda",      # Strong GPU recommendation for T5-Large
    torch_dtype=torch.bfloat16,
)
# Data Preparation: AirPassengers Dataset
url = "https://raw.githubusercontent.com/AileenNielsen/TimeSeriesAnalysisWithPython/master/data/
df = pd.read_csv(url)
# Convert specific column to PyTorch Tensor
context_tensor = torch.tensor
    df["#Passengers"].values, dtype=torch.float32)
```



Please note that Chronos expects a 1D float32 tensor for univariate context. Without the explicit `dtype`, NumPy's `float64` default propagates and Chronos either errors on CPU or silently produces poor forecasts under bfloat16.

3. Chronos generates forecasts by sampling from the learned token distribution, similar to how an LLM generates diverse text responses via temperature sampling

```
# Forecasting with Sampling
# prediction_length: Number of steps (tokens) to predict
# num_samples: Number of potential futures to generate (Monte Carlo)
forecast = pipeline.predict(
    inputs=context_tensor,  # older releases used "context"
    prediction_length=24,  # 2 Year Forecast
    num_samples=100,  # Generate 100 paths to estimate distribution
    limit_prediction_length=False
)
# Output Shape: ->
forecast_np = forecast.numpy()
# Post-Processing for Inventory Planning
# Calculate P10, P50 (Median), and P90
p10 = np.quantile(forecast_np, 0.1, axis=1)
p50 = np.quantile(forecast_np, 0.5, axis=1)
p90 = np.quantile(forecast_np, 0.9, axis=1)
# Visualization
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))
# Plot History
plt.plot(df.index, df["#Passengers"], label="History")
# Plot Forecast Median
forecast_index = range(len(df), len(df) + 24)
plt.plot(forecast_index, p50, label="Chronos Median", color='purple')
# Plot Uncertainty
plt.fill_between(
    forecast_index,
    p10,
    p90,
    color='purple', alpha=0.3, label="10-90% Quantile"
)
plt.title("Amazon Chronos: Long-Horizon Travel Demand Forecast")
plt.legend()
plt.show()
```

Unlike TimesFM, which produces structured quantile outputs directly, Chronos generates forecasts through sampling, allowing us to approximate the full predictive distribution. This makes it particularly

useful in decision-making scenarios where understanding uncertainty is as important as the forecast itself.



Advanced: Fine-Tuning with LoRA

For specialized domains (e.g., predicting server metrics with unique micro-second patterns), Zero-Shot performance might be insufficient. Chronos supports **LoRA (Low-Rank Adaptation)** fine-tuning, which allows users to adapt the model to new data distributions without retraining all 700M+ parameters. This is accessible via the training extras in the package installation (`pip install chronos-forecasting[training]`).

So the issue is our previous models primarily forecast single variables in isolation. The next section introduces a universal architecture designed to handle complex, multivariate relationships.

Salesforce Moirai

Both TimesFM and the original Chronos are primarily **univariate** models. They forecast a single variable based on its own history. However, in complex systems like energy grids or biological systems, variables are deeply interconnected. Temperature affects Load; Blood Pressure affects Heart Rate.

Salesforce Moirai 2.0 is a decoder-only universal forecaster designed to manage datasets with varying dimensions natively.

Moirai draws inspiration from **Vision Transformers** (**ViT**) and **Masked Language Modeling** (**BERT**). Its architecture incorporates several key innovations that enable this flexibility and generalization:

- **Multi-scale patching**: Moirai processes data using multiple patch sizes simultaneously. This allows it to capture high-frequency jitter (small patches) and low-frequency seasonality (large patches) in a single pass.
- **Flattened multivariate attention**: To handle multivariate data, Moirai flattens all variates into a single sequence but appends *Variate ID* embeddings. The attention mechanism is then masked so the model understands which tokens belong to *Temperature* and which belong to *Load,* allowing it to learn cross-variate correlations zero-shot.
- **LOTSA dataset**: Moirai is pretrained on the **LOTSA** archive, a collection of 27 billion observations from disparate domains, ensuring broad generalization capabilities.

This figure visualizes the Universal Masked Encoder design of Moirai which handles multivariate data. The input stage on the left demonstrates how multiple variables such as price and sales are flattened into a single one-dimensional sequence. The diagram explicitly shows the data splitting into parallel branches representing different patch sizes to capture both high-frequency jitter and low-frequency trends simultaneously. The central masked encoder uses any-variate attention to learn correlations across both time and variables before reconstructing the masked portions on the right.

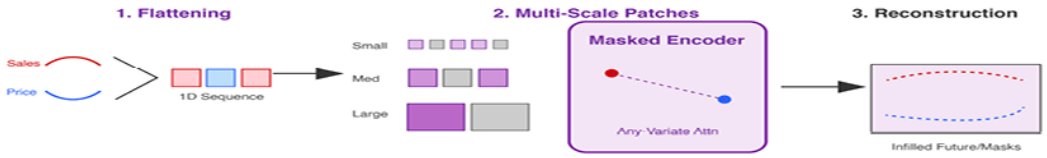


Figure 9.4: Moirai's architecture

Let's work on a practical example.

We will demonstrate a scenario where we forecast a target series given a known dynamic covariate (e.g., forecasting *Sales* given known *Price* changes).

1. We begin by installing the `uni2ts` library, which provides access to the Moirai model and its integration with GluonTS. Salesforce's recommended path is an editable install from the source repository:

   ```
   git clone https://github.com/SalesforceAIResearch/uni2ts.git
   cd uni2ts
   pip install -e '.[notebook]'
   ```

   This pulls in heavy dependencies (PyTorch, GluonTS, Lightning) and works on Linux and macOS with `Python 3.10–3.11`; Windows users typically run it inside `WSL2`. A `pip install uni2ts from PyPI` also exists, but the editable install is what the upstream notebooks expect.

2. Next, we load a pretrained Moirai model and wrap it in a GluonTS predictor. This step defines the forecasting horizon, context length, and how many variables (target + covariates) the model will use.

   ```
   import torch
   import pandas as pd
   from gluonts.dataset.pandas import PandasDataset
   from uni2ts.model.moirai import MoiraiForecast
   # 1. Initialize Moirai
   # We load the '1.0-R-Large' checkpoint (Robust version)
   model = MoiraiForecast.load_from_checkpoint(
       checkpoint_path="Salesforce/moirai-1.0-R-large",
       prediction_length=20,
       context_length=200,
       patch_size='auto', # Auto-select patch size based on data frequency
       num_samples=100,
       target_dim=1,           # We are predicting 1 target
       feat_dynamic_real_dim=1,  # We have 1 dynamic covariate
                                 # (e.g., Price)
       past_feat_dynamic_real_dim=0
   )
   # Create the Predictor (GluonTS wrapper)
   predictor = model.create_predictor(batch_size=32, device='cuda')
   ```

3. Here, we simulate a scenario where sales depend on price, and future price values are

   ```
   # 2. Data Preparation: Target + Covariate
   # Create a dataframe with a Target (Sales) and a Covariate (Price)
   # Assume 'Price' is known in the future (e.g., planned promotions)
   dates = pd.date_range("2023-01-01", periods=300, freq="D")
   ```

```
price = np.random.uniform(10, 20, 300)  # Random price fluctuations
# Sales negatively correlated with Price + Seasonality
sales = 100 - (price * 2) + 10 * np.sin(np.arange(300)/10) + np.random.normal(0, 2, 300)
df_wide = pd.DataFrame({
    "sales": sales,   # Target
    "price": price    # Dynamic Feature
}, index=dates)
# Convert to GluonTS Dataset
# Moirai expects specific field mappings for covariates
ds = PandasDataset(
    df_wide,
    target="sales",
    feat_dynamic_real=["price"] # Explicitly link the covariate
)
```

4. Finally, we generate forecasts and inspect how the model incorporates the covariate signal into its predictions

```
# 3. Forecast
# The model uses the future 'price' values to inform the 'sales' forecast
forecasts = list(predictor.predict(ds))
# 4. Analysis
# Comparing the forecast accuracy
for cast in forecasts:
    print(f"Forecast Mean: {cast.mean[:5]}")  # First 5 days
    # Moirai effectively uses the 'price' drop/hike in the horizon
    # to adjust the sales prediction down/up.
```

In this example, if we used a univariate model like Chronos, it would have to guess the future sales based *only* on past sales. It would miss the impact of a planned 50% price discount next week. Moirai, by ingesting the *Price* covariate, can predict the demand spike associated with that discount zero-shot.

We have implemented quantitative models. Next, we explore how to combine these numerical predictions with unstructured text for richer insights.

Qualitative and multimodal forecasting

We now cross the frontier from purely quantitative *number-crunching* to forecasting that also reads text. A clarification on terminology: the classical forecasting literature uses *qualitative methods* to mean expert judgment, Delphi panels, and consensus techniques where no historical data is required. That is not what we mean here. In this chapter, *qualitative* refers to the multimodal workflows that pair numerical series with textual context, financial news reports, doctor's clinical notes, server error logs, economic bulletins, and use a language model to reason over both. You may also see this called **context-aware forecasting** in recent papers.

Time-Series Language Models (TSLMs) are a new breed of foundation models designed to process time series as a native modality alongside text. This allows for complex reasoning tasks that numbers alone cannot support.

Theoretical multimodal workflows require specific tools to function. We now examine a framework that bridges the gap between vision-language models and temporal data.

Introducing OpenTSLM

Wu and others (2025) in their paper *OpenTSLM: Time-Series Language Models for Reasoning over Multivariate Medical Text* (https://arxiv.org/html/2510.02410v1) developed **OpenTSLM**, a framework from researchers at Stanford and ETH Zurich. It adapts Vision-Language Models (like DeepMind's Flamingo) for the temporal domain. Instead of an image encoder, it uses a **Time Series Patch Encoder** and feeds the resulting embeddings into a frozen LLM (like Llama 3) via gated cross-attention layers. Two terms borrowed from the vision-language world that we will use repeatedly: a **perceiver resampler** is a small attention module that compresses a variable-length input into a fixed number of tokens (think of it as a learnable downsampler); **gated cross-attention** is a cross-attention layer that starts with its output multiplied by zero and gradually opens up during training, so a frozen LLM's behavior is recovered exactly when the gate is closed.

The OpenTSLM framework implements two distinct strategies for integrating time series into LLMs:

- **OpenTSLM-SP (Soft Prompting):** This architecture treats time series patches as *virtual tokens.* It uses a TransformerCNNEncoder to project time series segments directly into the LLM's embedding space. These tokens are concatenated to the text prompt. This is parameter-efficient but memory-intensive for long sequences.
- **OpenTSLM-Flamingo (Cross-Attention):** This is the more scalable *inference-ready* architecture. It uses a Perceiver Resampler to compress variable-length time series into a fixed number of latent tokens. These latents are then injected into the frozen LLM via Gated Cross-Attention layers. This allows the model to handle long histories without exploding the context window. The OpenTSLM-Flamingo implementation relies on several key components:
    - **Time Series Encoder**: The model uses a patch-based encoder (similar to TimesFM) to convert the raw time series into dense embedding vectors.
    - **Perceiver Resampler**: This component compresses the variable-length time series embeddings into a fixed number of tokens. That fixed-size handoff is what keeps the time-series side from overwhelming the LLM's context window.
    - **Frozen LLM Backbone**: The core reasoning engine is a standard LLM (e.g., Llama-3.2-1B). It is *frozen* during training to preserve its linguistic capabilities, while the cross-attention layers are trained to align the time series representations with the text.

To understand how **OpenTSLM** works, take a healthcare scenario. A raw ECG signal (time series) is often ambiguous without the patient's history (text). OpenTSLM can ingest both.

- **Input**: A 12-lead ECG signal + Text: *Patient is a 65-year-old male complaining of chest pain.*
- **Task**: *Diagnose the rhythm and explain your reasoning.*
- **Output**: *The ECG shows ST-segment elevation in leads V2-V4, consistent with an Anterior Myocardial Infarction. The patient's age and symptoms support this diagnosis.*

The actual OpenTSLM implementation relies on wrapping a frozen Large Language Model (like Llama 3.2) with a trainable adapter. We use the OpenTSLMFlamingo class, which handles the *Perceiver Resampler* (to compress long time series) and injects *Gated Cross-Attention* layers into the LLM.

OpenTSLM is research code rather than a packaged library. The from model.llm.OpenTSLMFlamingo import OpenTSLMFlamingo line below refers to the directory layout of the project repository at

https://github.com/StanfordBDHG/OpenTSLM, not a pip install-able module. To run this example, clone that repository and execute from its root (or add it to `PYTHONPATH`). The companion repo for this book contains a wrapper that mirrors this layout.

1. We initialize the model specifying the base LLM and how frequently to inject time-series awareness (`cross_attn_every_n_layers`)

```
import torch
from model.llm.OpenTSLMFlamingo import OpenTSLMFlamingo
# Initialize the model on GPU
# We use the 1B parameter Llama model as the frozen backbone.
# cross_attn_every_n_layers=1 means we inject time-series info into *every* transformer block.
device = "cuda" if torch.cuda.is_available() else "cpu"
model = OpenTSLMFlamingo(
    device=device,
    llm_id="meta-llama/Llama-3.2-1B",
    cross_attn_every_n_layers=1,
    freeze_lm_embeddings=False
)
print("Model loaded. Backbone is frozen; Cross-Attention layers are trainable.")
```

2. Inference OpenTSLM requires a specific input structure. Instead of raw text, we feed it a list of dictionaries containing the text prompts and the aligned time-series tensors. The `<image>` token (borrowed from the Flamingo vision framework) serves as the placeholder where the time-series embedding is injected.

```
# Create a sample batch (Batch Size = 1)
# 'time_series' must be a tensor of shape (Channels, Length)
sample_input = [{
    "pre_prompt": "Analyze the following 12-lead ECG signal:",
    "time_series_text": ["<image>"],  # Placeholder for the
                                      # signal embedding
    "time_series": torch.randn(12, 5000),  # Simulated ECG
                                           # (12 leads, 5000 time steps)
    "post_prompt": "Question: What is the diagnosis? Answer:",
    "answer": "Normal Sinus Rhythm"  # Optional: Used for
                                     # training/eval
}]
# Generate Reasoning
# The model automatically compresses the 5000 time steps
# into fixed tokens and generates a text response.
output_text = model.generate(
    batch=sample_input,
    max_new_tokens=100
)
print(f"Model Output: {output_text[0]}")
```

This example demonstrates how OpenTSLM moves beyond numerical forecasting to multimodal reasoning. By combining time series signals with textual context, the model can generate explanations alongside predictions, an essential capability in domains such as healthcare, where interpretability is as important as accuracy.

OpenTSLM is one route to multimodal forecasting: train a specialized architecture that ingests time series as a native modality. There is a second route that requires no new model at all. Off-the-shelf multimodal LLMs (GPT-4o, Claude, Gemini) already read plots as images, and a well-structured

prompt is often enough to pull useful analysis out of them. The trade-off is precision against accessibility: OpenTSLM has tighter grounding on the raw signal and lower latency; a multimodal LLM works on day one against whatever model your team already pays for. The pattern that makes the second route reliable is the subject of the next subsection.

Prompt engineering for multimodal LLMs: Chain-of-Charts

When the model is a general-purpose multimodal LLM rather than a TSLM, the time series usually arrives as a plot image, not as a tensor. Naive prompting (*What is the minimum value in this chart?*) fails for the same reason it fails on math problems: the model takes a shortcut and hallucinates. Just as Chain-of-Thought prompting fixed math reasoning by forcing intermediate steps, **Chain-of-Charts** fixes chart QA by forcing the model to extract data features before it reasons over them. Zhang and others (2024) in their paper ChartInsights: Evaluating Multimodal Large Language Models for Low-Level Chart Question Answering (https://aclanthology.org/2024.findings-emnlp.710.pdf) showed this empirically on the ChartInsights dataset: direct extraction prompts hallucinate; step-wise prompts ground the model in actual chart content.

When asking a TSLM or Multimodal LLM (like GPT-4o) to analyze a plot or time series, use this structure:

Step	Prompt Instruction	Rationale
1. Structure	*What type of chart/data is this?*	Orients the model to the data format (Line chart vs. Scatter).
2. Axes	*What are the labels on the x-axis? What is the range of the y-axis?*	Forces the model to tokenize the scale and temporal boundaries.
3. Extraction	*What are the data labels/values for the key elements?*	Populates the context window with the actual numbers.
4. Reasoning	*Based on the extracted values, calculate the trend, minimum,* and *maximum.*	Performs the logic on the extracted text rather than the raw image or signal.
5. Conclusion	*My answer is [Answer].*	Finalizes the output.

Table 9.4: Chain-of-Charts prompt template. A five-step decomposition that grounds a general-purpose multimodal LLM in the data before asking it to reason

Using this template has been shown to boost accuracy on low-level analysis tasks by over 14%.

Effective models require aligned training data. We will now look at a large-scale resource that connects financial news with stock prices.

Multimodal datasets

This dataset is typically used to train models like OpenTSLM. To train or fine-tune these qualitative models, one requires aligned Text-Time data. Dong and others (2024), in their paper *FNSPID: A Comprehensive Financial News Dataset in Time Series* (https://arxiv.org/html/2402.06698v1), presented the **Financial News and Stock Price Integration Dataset** (**FNSPID**) as a premier resource. Its key characteristics:

- **Scale**: 15.7 million news items aligned with 29.7 million stock price records.
- **Features**: Includes *Sentiment* scores, *Impact Level* (High/Medium/Low), and raw article text.
- **Application**: Training models to predict *Volatility* based on *News Sentiment*. A generic *Positive* sentiment might not predict price direction (the market may have already priced it in), but *Breaking News* almost always predicts *High Volatility*.

Due to its large size (terabytes), the dataset is typically accessed using streaming rather than a full download. The Hugging Face `datasets` library provides a convenient interface:

```
from datasets import load_dataset
from transformers import pipeline
def analyze_financial_stream(limit: int = 5) -> None:
    """Stream financial news and predict sentiment using FinBERT."""
    analyzer = pipeline(
        "sentiment-analysis",
        model="ProsusAI/finbert"
    )
    dataset = load_dataset(
        "Zihan1004/FNSPID",
        streaming=True
    )
    print("Streaming and analyzing financial news...")
    for i, sample in enumerate(dataset["train"]):
        ticker = sample["Stock_symbol"]
        news = sample["Article_title"]
        date = sample["Date"]
        result = analyzer(news)[0]
        label = result["label"]
        confidence = result["score"]
        print(f"[{date}] {ticker}: {news}")
        print(f" -> Sentiment: {label} ({confidence:.2f})")
        if i >= limit - 1:
            break
if __name__ == "__main__":
    analyze_financial_stream()
```

This code provides a functional baseline. We route the streamed headlines through a pre-trained financial language model to generate an immediate sentiment signal. You can aggregate these discrete signals over a rolling window to create a dynamic covariate for the quantitative models discussed earlier.

We have explored what these tools can do. Now we must turn to the risks and failure modes that come with deploying them.

Critical challenges

While Foundation Models offer Zero-Shot capabilities that seem like magic, they introduce specific *Zero-Day* risks unique to this paradigm. Unlike a local ARIMA model, where failure is usually obvious (flat lines or exploding variance), TSFM failures can be subtle, confident, and catastrophic.

Recent benchmarking by Puvvada and Chaudhuri (2024) and Zhang and others (2025) in their paper *The Memorization Problem: Can We Trust LLMs' Economic Forecasts?* (https://arxiv.org/pdf/2504.14765) suggest that for production systems, we must treat TSFMs not as silver bullets, but as *high-risk, high-reward* components requiring rigorous containment.

Deployment requires an honest assessment of performance. This section warns against assuming complex models automatically outperform simple baselines.

Performance risks

Why this happens: a TSFM minimizes average loss across millions of unrelated series during pretraining. The model that wins on average across all data is rarely the model that wins on your specific series. Once a baseline has been allowed to exploit series-specific structure (lag features, calendar effects, vacation flags), the foundation model's general-purpose priors lose their edge.

The most common trap for practitioners is assuming a Foundation Model is superior simply because it is large. In rigorous benchmarks, TSFMs like TimesFM or TimeGPT are often outperformed by simple, well-tuned gradient boosting models (like LightGBM) or even statistical baselines. Foundation models often win on *Zero-Shot* leaderboards against untuned baselines. Once a baseline like XGBoost is hyperparameter-tuned with proper feature engineering, the advantage of the foundation model frequently evaporates. The benchmark in The need for time series foundation models earlier in this chapter shows the pattern: zero-shot Chronos beats seasonal-naive, but loses to a LightGBM with five lag features.

Recommendation: never deploy a TSFM without first benchmarking it against a *Simple Seasonal Naive* and a tuned *LightGBM* baseline. If the TSFM cannot beat the naive forecast by a significant margin, it is adding technical debt without value.

Performance issues often stem from fundamental data differences. We next examine how discrepancies in frequency and domain can degrade model accuracy.

Generalization risks

Why this happens: a TSFM's *priors* are statistical fingerprints absorbed from its training corpus, the typical frequency bands, value ranges, autocorrelation shapes, and shock patterns of the series it saw. When your data lives outside those fingerprints, the priors are not merely imprecise; they are wrong, and the model confidently extrapolates the wrong structure.

Foundation models rely on the patterns absorbed during pretraining: trend, seasonality, noise, and shocks repeated across millions of unrelated series. Time series data is not as universal as language; two specific mismatches account for most generalization failures:

- **Spectral Shift:** TSFMs degrade when the frequency domain (spectral properties) of your target data differs from the pre-training corpus. A model trained primarily on human-centric data (weekly

sales, daily traffic) may hallucinate periodicity when applied to high-frequency IoT sensor data or chaotic biological signals. The mechanism is direct: the model has learned which frequencies are *signal* vs. *noise* from its training distribution, and that judgment doesn't transfer.

- **Domain Mismatch**: *Universal* models often struggle across drastically different domains (e.g., finance to healthcare). If the physical dynamics of your system (e.g., conservation of energy, monotonicity, hard bounds) are not represented in the training set, the model may predict trajectories that violate physical laws. The forecast looks plausible to a generic loss function but absurd to a domain expert.
- **Mitigation**: run an FFT comparison between your data and the model's training corpus (the dominant frequencies should overlap) and sanity-check forecasts against any hard physical constraints your domain enforces.
- Generalization failures are not the only risk. We must also ensure the model is predicting the future rather than recalling it from memory.

Memorization risks

Why this happens: a sufficiently large model trained on internet-scale data approximates a lookup table on points it has seen during training. The *forecast* for a date the model saw at training time is effectively a retrieval, not a prediction, even though the model has no explicit awareness of the difference.

Large Language Models have *read* the internet, which includes historical stock prices, GDP reports, and weather archives. If you backtest a model on the S&P 500 from 2022, it may achieve 99% accuracy not because it learned market dynamics, but because it memorized the values during training. Even when you anonymize the data (e.g., renaming *Apple* to *Asset A*), models can sometimes recognize the unique *shape* or volatility signature of major assets and hallucinate the correct future values from memory. Pattern fingerprinting beats name anonymization.

Mitigation: strict decontamination. Ensure your test set is temporally after the model's training cutoff date, and prefer hold-out periods that post-date the model's release by at least a quarter.

Adversarial fragility

Why this happens: the patching or tokenization layer at the input edge of a TSFM is differentiable, and the model has many capacity dimensions over a continuous input space. Gradient-based attacks find the smallest perturbation that flips an output decision boundary; because the budget is measured in floating-point shifts rather than pixels, the perturbation can be far below the human perception threshold yet still cross a forecast threshold.

Zhang and others (2025) in their paper *The Memorization Problem: Can We Trust LLMs' Economic Forecasts?* (https://arxiv.org/pdf/2504.14765) demonstrated this empirically. Small, imperceptible noise injected into the input window can reverse the predicted trend from "*Growth*" to "*Collapse*." The same brittleness affects safety-critical applications: a forecast used for autonomous scaling, automated trading, or grid-frequency control can be redirected by an adversary who controls a small fraction of the input stream.

Mitigation: input sanitization (smoothing, outlier removal, bounded clipping) before the data reaches the model; ensembling forecasts from architecturally different models; for the highest-stakes deployments, randomized smoothing or certified-robust variants of the architecture.

The following script is a teaching sketch of the **Fast Gradient Sign Method** (**FGSM**) that can be used to stress-test a model's fragility before deployment. It assumes the model is a callable PyTorch module that takes the time-series tensor as input and returns the forecast. Wrapping a real foundation model for gradient-based attacks is more involved: TimesFM and Chronos package their tokenization/patching inside `pipeline.predict()` calls that detach gradients, so you generally need to expose the underlying `forward()` of the encoder or call the model with `torch.enable_grad()` explicitly. Treat the snippet below as the diagnostic logic, not a drop-in attack on the libraries we used earlier.

```
import torch.nn as nn
def fgsm_attack(model, input_series, target, epsilon=0.1):
    """Generates an adversarial time series to fool the model."""
    # 1. Enable gradient tracking on the input data
    input_series.requires_grad = True
    # 2. Forward Pass
    forecast = model(input_series)
    # 3. Calculate Loss (Maximize error from target)
    loss = nn.MSELoss()(forecast, target)
    # 4. Zero Gradients
    model.zero_grad()
    # 5. Backward Pass to get Gradient w.r.t INPUT
    loss.backward()
    # 6. Create Perturbation
    # Move input in the direction that increases loss
    data_grad = input_series.grad.data
    perturbed_series = input_series + epsilon * data_grad.sign()
    return perturbed_series
# If the error between model(perturbed_series) and model(input_series)
# is high, the model is too fragile for safety-critical deployment.
```

Probabilistic calibration

Why this happens: TSFMs that emit quantile forecasts or sampled paths calibrate their interval widths on the training distribution. The variance the model expects to see, the tail thickness, the regime-shift frequency, are all baked in during pretraining. When your data has a different volatility regime, fatter tails, or a structural break the model never saw, the published *90% interval* no longer covers 90% of actuals.

A calibration failure looks innocuous on a point-forecast metric like MAPE but is catastrophic for downstream decisions. A 90% interval that empirically covers 60% of actuals under-reports tail risk to a risk manager; one that covers 99% is so wide it carries no information for inventory planning. The model's intervals are not *wrong* in the sense of being noisy; they are systematically biased against the deployment distribution.

Mitigation: wrap the foundation model in conformal prediction, the framework covered in chapter 8. Conformal calibration uses a held-out portion of your data to adjust the model's intervals so empirical coverage matches the target coverage. In production, track the **Prediction-Interval Coverage**

Probability (**PICP**) on a rolling window; drift below the target is a leading indicator that calibration is breaking down.

Long-horizon decay

Why this happens: autoregressive TSFMs (TimesFM, Chronos in classic mode) feed each predicted patch back as context for the next step. Any error at step t corrupts the input for step t+1, which compounds at step t+2. The error growth is geometric, not additive. Patching softens the effect by spreading the prediction across multiple time steps per autoregressive call, but it does not eliminate the underlying compounding. Models trained on short forecast horizons (e.g., 96 steps) have especially poor long-horizon behavior because they never saw the error-compounding regime during training.

The symptom is a forecast whose error curve looks flat for the first few patches and then climbs sharply, a *hockey stick* past some horizon threshold. The threshold tends to fall near the maximum horizon the model was trained on.

Mitigation: bound the deployment horizon (a conservative rule of thumb: do not forecast beyond the model's training horizon). For applications that need long horizons, prefer direct multi-step variants like Chronos-Bolt, which emit the entire horizon in a single forward pass without autoregressive compounding. For truly long horizons (years on monthly data), hand off to statistical methods (exponential smoothing, ARIMA, structural time series) that capture trend without compounding step errors.

Identifying risks is the first step toward mitigation. We conclude the risk discussion with a framework to audit and secure your deployment.

The risks discussed can be systematically evaluated using the following framework:

Risk Category	**Symptom / Failure Mode**	**Mitigation Strategy**
Performance risks	TSFM underperforms simple models on specific granularities (e.g., daily vs. weekly).	**Benchmark:** Must beat a tuned LightGBM and Seasonal Naive model on the specific test set.
generalization risks	Model fails on high-frequency or irregular data (e.g., IoT sensors) not seen in training.	**Frequency Check:** Analyze the dominant frequencies (FFT) of your data vs. the model's training corpus.
Memorization risks	Suspiciously high accuracy (>99%) on historical public data.	**Decontamination:** Only test on data generated after the model's release date.
Adversarial Fragility	Small input noise causes trend reversals or massive variance.	**Sanitization:** Apply smoothing or outlier detection to inputs before feeding them to the FM.

Probabilistic Calibration	Prediction intervals are too narrow or too wide (poor CRPS scores).	**Recalibration:** Use Conformal Prediction (post-processing) to adjust the output intervals.
Long-Horizon Decay	Error increases monotonically as the forecast horizon increases.	**Horizon Limits:** Restrict FM usage to short/medium horizons; use statistical methods for long-term trends.

Table 9.5: Risk matrix

Deploying foundation models in production

The generic deployment story, freshness, latency SLAs, feature stores, drift monitoring, and distributed inference apply to TSFMs as much as to any other model class, and chapter 10 is where we cover it end-to-end. Five concerns are specific to foundation models and worth pinning down here, while the architectures are still fresh in mind.

- **Head/tail routing.** A 200M-parameter TimesFM call per SKU is fine for the top 1% of revenue (*Head* items) and ruinous for the long tail. The standard pattern is a router that sends head items to the foundation model and tail items to a statistical baseline (seasonal-naive, ARIMA, or a global LightGBM trained on the cohort). The decision flowchart in *Figure 9.1* already implies this: the same chart that picks a model for a single series picks a cohort-level policy when you run it per segment.
- **Caching identical contexts.** Many production callers ask the same question many times: a rolling 30-day forecast for a stable SKU rarely changes when called twice in the same hour. Cache the (input window, model version, horizon) tuple with a short TTL (minutes for fast-moving signals, hours for stable ones). On a typical retail deployment, caching cuts FM call volume by 60–80% with no accuracy cost.
- **Zero-shot to fine-tune as an escalation path.** Zero-shot is the default not the destination. When the held-out benchmark shows the FM losing to a tuned LightGBM (the Performance risks failure mode), the next step is parameter-efficient fine-tuning, LoRA for Chronos, adapters for OpenTSLM, on a slice of your own data. This recovers most of the foundation-model upside (cross-series transfer, cold-start coverage) without the cost of training from scratch. Budget a few GPU-hours, not a few GPU-weeks.
- **Versioning the upstream model.** Foundation models are external dependencies that get re-released (`TimesFM 1.0 → 2.0 → 2.5, Chronos T5 → Chronos-Bolt`). Treat each upstream version as a new deployment: pin the model hash, run shadow-mode comparison against the incumbent on a calibration set, and promote only after coverage and PICP hold. Surprise drift on a model upgrade is one of the easier failure modes to avoid and one of the more embarrassing ones to debug after the fact.
- **Cold-start as the production case for FMs.** Statistical and global ML models need a training history. TSFMs do not. The high-value production deployment is the cohort with thin data: new SKUs, new sensors, new geographies, anomaly detection on series that have only existed for weeks. Building the head/tail router above is also the mechanism for routing every new series

automatically to the FM until enough history accumulates to graduate it to a cheaper specialized model.

The rest of the production picture—ingest pipelines, point-in-time feature stores, shadow deployments, drift alerts, the Bullwhip Effect—is generic to all model classes and is the subject of the next chapter.

Summary

In this chapter, we shifted from training models on every new problem to using pre-trained Foundation Models for inference. The trade-off is the focus of the chapter: zero-shot speed and cold-start coverage on one side, fragility and memorization risk on the other.

We explored two distinct categories of these foundation models. First, we looked at quantitative models like TimesFM and Chronos. These architectures treat time series as patches or tokens to generate zero-shot forecasts without any gradient updates. Second, we examined qualitative workflows. We saw how Time-Series Language Models (TSLMs) and frameworks like OpenTSLM allow us to reason about the textual context surrounding the data.

However, we also confronted the safety risks unique to this approach. We learned that these models are brittle. They can fail due to imperceptible noise or provide misleadingly high accuracy by *memorizing* historical data instead of predicting it.

The next generation of models will likely be Action-Oriented Agents. Instead of just forecasting *Inventory will drop to 0,* an Agentic Forecaster will read the forecast, read the supplier's email (Text), check the shipping rates (Covariate), and automatically place a reorder, closing the loop between prediction and decision.

As you deploy these models, remember: Zero-Shot is a starting point, not an endpoint. Rigorous validation, robustness testing, and continuous monitoring remain the pillars of reliable machine learning systems.

You now possess a complete modeling toolkit ranging from classical statistical methods to modern deep learning and foundation models. The final step is deployment. In the next chapter, we will focus on a production workflow. You will learn to use MLOps practices and tools like MLflow to package, monitor, and scale your forecasting systems.

Get this book's PDF version and more

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

10

Production Workflows: Deployment, Monitoring, and Scaling

We have spent the previous chapters exploring the wide range of modeling options, from the statistical rigor of ARIMA in *Chapter 3* to the zero-shot capabilities of Foundation Models in *Chapter 9*. In those contexts, success was defined by a single metric: accuracy on a hold-out test set.

However, a model residing in a Jupyter notebook is not a product. The transition from a research environment as defined by static CSV files and clean validation sets to a production environment defined by unbounded data streams, strict latency SLAs, and shifting distributions is often where time-series projects fail.

In the industry, the practice of developing and deploying models is often separated into distinct job roles, where job titles like machine learning engineer are concerned with deployment. However, even if you are more concerned with developing the models rather than with deploying them to production, you still need to understand how different model types and training decisions affect deployment, production performance, and monitoring.

Furthermore, time-series production systems operate in feedback-driven environments where today's forecast influences tomorrow's data, creating complex loops like the Bullwhip Effect.

In this chapter, we will explore:

- The production gap
- Architectural patterns for time-series deployment
- Data engineering for time series
- Scaling strategies
- Deployment strategies
- Monitoring and observability
- Advanced failure modes and mitigations



To keep these topics concrete, we will follow one system through the whole chapter. Glide is a fictional ride-hailing platform operating in 40 cities. Its core forecasting problem is trip demand: how many riders will request a car in each city zone over the next 15 minutes, the next hour, and the next week. That one forecast drives decisions at three speeds. Long-range demand sets weekly driver-incentive budgets, day-ahead demand positions drivers across each city, and next-15-minutes demand feeds surge pricing. A data scientist at Glide has just built a demand model that beats the current baseline in a notebook. The rest of this chapter is the story of getting it into production, and every section returns to Glide to make the pattern concrete.

Operationalizing time series systems requires an architecture capable of bridging the Production Gap. Let's start with this!

The production gap

We've been discussing a lot of different model types in the chapters so far such as statistical models in *Chapter 3*, forecasting reduced to regression with ML models in *Chapter 4*, feature engineering in *Chapter 5*, more complex models in *Chapter 6*, deep learning in *Chapter 7*, uncertainty and prediction intervals in *Chapter 8*, and finally foundation models in *Chapter 9*. All these models and variants have their implications for production. For example, the pursuit of algorithmic novelty with architectures like Temporal Fusion Transformers, N-BEATS, or PatchTST introduces a host of complexities.

The transition from a research environment to a production environment creates what is often termed the **Production Gap**.

The following table illustrates these differences.

Feature	**Training Environment (Research)**	**Production Environment (Operations)**
Primary Artifact	Jupyter Notebooks and static files	Real-time services and unbounded data streams
Data Source	Static CSVs, fixed	Continuous, unbounded

	holdout sets	streams with shifting distributions
Data Assumptions	Independent and Identically Distributed (IID)	Non-stationary, susceptible to concept drift and regime shifts
Constraint	None (code runs until completion)	Strict Latency SLAs and *Freshness* Requirements
Feedback Loop	Passive: Model observes historical data without changing it	Active: Predictions influence future data (for example, Bullwhip Effect)
System State	Stateless/Isolated execution	Entangled: Complex dependency between code, data, and system state
Success Metric	Accuracy (for example, RMSE) on a static test set	Reliability, scalability, and resistance to feedback loops

Table 10.1: Differences between training and production environments

As noted in a foundational paper, *Machine Learning: The High-Interest Credit Card of Technical Debt* from Google Research (D. Sculley, et al., 2014; https://research.google/pubs/machine-learning-the-

high-interest-credit-card-of-technical-debt/), ML systems have a special capacity for incurring maintenance costs because they create entanglement between data, code, and system state. While in standard software, you can usually change the code without breaking the data, in ML, the code, the data, and the system settings are tied together. This means that if you change the input data even slightly, the model behavior might change completely. This makes updates risky and expensive.

For time series specifically, this debt is compounded by the **Freshness** requirement; the operational necessity that input data must be up-to-date to be valuable. A forecast is often useless if the data is old. Success in a production setting is defined by the system's ability to handle data delays, maintain state consistency, recover from feedback loops, and scale to millions of unique series without linear cost growth, not by the lowest **Root Mean Squared Error** (**RMSE**) on a static test set.

The collapse of algorithmic home-buying businesses, most notably *Zillow Offers* in 2021, serves as a cautionary example of deploying predictive models without sufficient safeguards against market volatility and endogenous risk. Zillow relied heavily on algorithmic price forecasts to guide large-scale home purchases, assuming short-term appreciation and stable resale conditions. When housing market dynamics shifted rapidly, these forecasts proved unreliable, leaving the company exposed to significant inventory risk.

While Zillow did not publicly attribute its failure to algorithmic feedback loops, the episode illustrates how model-driven market participation can amplify error. Aggressive buying based on

optimistic price projections increased exposure precisely as the market cooled. Because the models were trained primarily on historical and lagging indicators, they struggled to detect the regime change in time. Zillow ultimately exited the business after writing down hundreds of millions of dollars in inventory losses and laying off roughly 25 percent of its workforce.



Lessons learned:

- Zillow's forecast moved the market it was trying to predict. Buying at scale on optimistic projections inflated exposure just as the market cooled, an active feedback loop of the same kind as the Bullwhip Effect covered later in this chapter. The mitigation is to damp how aggressively the system acts on its own forecasts, rather than optimize for forecast error alone.
- A model trained on lagging indicators cannot catch a regime change in time. Drift detection on deseasonalized residuals, covered under monitoring and observability, is built to flag that structural break before the losses confirm it.

Similarly, the 2021 failure of the *Texas power grid* during Winter Storm Uri in the United States highlights the risks of relying on forecasting models that do not generalize to extreme, out-of-distribution events. The grid is operated by the **Electric Reliability Council of Texas** (**ERCOT**), an independent organization responsible for balancing electricity supply and demand for most of

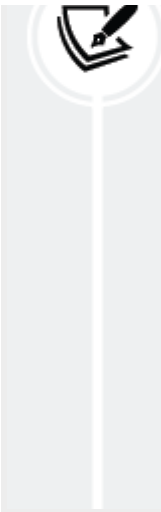
the state of Texas. Unlike many other power systems, ERCOT operates largely in isolation from neighboring grids, which limits its ability to import electricity during emergencies.

Power grids require advance planning because electricity must be produced at the same moment it is consumed. Grid operators rely on load forecasting models to estimate how much electricity will be needed under different weather conditions so they can ensure sufficient generation capacity and fuel availability. During Winter Storm Uri, unusually severe cold caused electricity demand to surge while large portions of the generation fleet and fuel supply failed due to inadequate winterization. Forecasting models underestimated the severity of the event, in part because such temperatures fell outside the range of historical conditions typically used for planning.

The result was a combination of unexpected demand, insufficient available supply, and limited ability to draw power from outside the system. This mismatch forced ERCOT to order widespread blackouts to prevent a total grid collapse. While the crisis was driven primarily by physical and institutional failures, it also illustrates how planning models that rely on historical patterns can break down when confronted with rare but high-impact events.

Lessons learned:

- Models extrapolate poorly beyond the range they trained on. Winter Storm Uri ran colder than the historical record, so the load forecast was confidently wrong. Feature drift monitoring catches this: an input outside the training



distribution should raise an alarm instead of producing a forecast the model has no basis for.

- When ground truth is delayed, performance estimation fills the gap. Methods like CBPE detect that production inputs have moved into a region of the feature space where the model was never validated, raising the estimated error before the outcome confirms it.

Understanding the technical debt and constraints is only the first step. Bridging the production gap is the work of the rest of this chapter, and it follows a fixed sequence, shown in *Figure 10.1*.



Figure 10.1: The production forecasting pipeline

Data moves left to right through five stages, each the subject of one section of this chapter. *Stage 1*, **Architecture**, fixes how data flows. *Stage 2*, Data engineering, keeps that data correct. Stage 3, **Training and scaling**, fits models across many series. *Stage 4*, **Deployment**, releases a model to live traffic. *Stage 5*, **Monitoring**, watches it for degradation. The dashed arrow is the operational loop: monitoring detects drift and triggers retraining. A second loop, where the model's own forecasts change the future data it sees, is the subject of the chapter's final section.

Each section that follows takes one stage of this pipeline, and we return to the figure at the start of each to mark where we are. Each stage also has its tools; introduced one section at a time, they can feel

like a scattered toolbox. *Table 10.2* heads that off, placing every tool the chapter uses on the pipeline at once, as a single workflow.

Pipeline stage (Figure 10.1)	**Tools**	**Role in Glide's system**
Architecture	Airflow, Kafka, Flink	Schedule batch runs; carry and process the live event stream
Data engineering	Feast, Pandera	Serve point-in-time-correct features; validate the incoming feed
Training and scaling	Ray	Fit and serve a demand model for every zone across a cluster
Deployment	MLflow, FastAPI	Package the model; shadow-test it on live traffic
Monitoring	NannyML	Estimate error and detect drift before ground truth arrives

Table 10.2: Glide's production stack

Read top to bottom, the table is the workflow: a trip event enters at the architecture stage and a monitored forecast leaves at the bottom. Every later section introduces one row in depth.

Architectural patterns for time-series deployment

Stage 1 of the *Figure 10.1* pipeline is the architecture, the choice the other four stages build on. For *Glide*, it decides how the demand pipeline is wired. A nightly batch run sizes next week's driver-incentive budget; surge pricing needs a forecast refreshed every few minutes. One system, two clocks.

The way a time-based forecasting system is built determines how fast it can respond, how reliable it is when things go wrong, and how expensive it is to run (latency profile, fault tolerance, and operational cost). Forecasting systems are tightly tied to the rhythm (or cadence) of the real-world processes they support. They must keep up with how often new data arrives and how quickly decisions need to be made.

As a result, the most important design choice in forecasting is how the data is processed. Systems that process data in large batches are usually cheaper and more efficient, but slower to react. Systems that process data continuously can respond much faster, but at higher operational cost. In the upcoming sections we will look at two architectures in depth.

Batch processing architectures

Batch processing is still the most common approach for forecasting tasks that support planning and decision-making, such as managing retail inventory, projecting revenue, or planning staffing levels. In batch systems, data is collected over a period of time and predictions

are generated on a fixed schedule, such as hourly, daily, or weekly, rather than continuously.

For these kinds of decisions, getting a forecast a few minutes late rarely matters. What matters far more is that the prediction is accurate and stable. In many cases, the added cost and operational complexity of building and maintaining real-time forecasting systems outweigh any benefit from faster updates.

The schedule-compute-cache pattern

Batch architectures typically use a **Schedule-Compute-Cache** pattern. An orchestration engine triggers a workflow at a specific time. The system computes forecasts in bulk using distributed frameworks. It saves the results to a low-latency database for later retrieval.

An orchestration engine, such as Apache Airflow, Prefect, or Dagster, serves as the heartbeat of the system, triggering workflows at a fixed cadence (for example daily at 00:00 UTC or weekly on Sundays). This end-to-end workflow can be broken down into a series of well-defined stages, each responsible for a specific part of the *Schedule-Compute-Cache* lifecycle:

1. **Ingestion:** High-volume historical data is extracted from a Data Warehouse (for example Snowflake, BigQuery) or a Data Lake (S3/Parquet). This step often involves heavy SQL queries to join disparate tables (for example sales, pricing, weather) into a unified dataset.
2. **Transformation and inference:** Distributed compute frameworks like Apache Spark, Ray, or Dask perform feature

engineering and inference in bulk. This allows for vectorized operations across millions of time series (for example Stock Keeping Unit, SKUs) simultaneously, maximizing hardware utilization and minimizing the overhead of network round-trips.

3. **Serving layer:** The generated forecasts are not returned to a user immediately but are materialized into a low-latency read store. Key-value stores like Redis, Cassandra, or DynamoDB are preferred here.
4. **Retrieval:** When a downstream application (for example an ERP system or a dashboard) requires a forecast, it performs a simple database lookup ($O(1)$ complexity) rather than triggering model inference

Global context and robustness

Batch architectures offer practical advantages for many types of time series tasks because they can collect and process the full dataset at once, which makes it straightforward to use global information across all time series and hierarchy levels. This **global context** is especially important in **hierarchical forecasting**, where forecasts at different aggregation levels must be coherent, for example ensuring that individual store forecasts sum up to regional totals. Methods such as **Minimum Trace** (**MinT**) reconciliation adjust forecasts using an estimated covariance matrix of forecast errors so that they respect these aggregation constraints. Estimating and manipulating these covariance matrices for a large hierarchy can be computationally intensive and naturally fits a batch processing workflow, where the

full history of data and errors is available, rather than a system that only processes data one event at a time.

Batch jobs are also more resilient when something goes wrong. Because they process data in clearly defined runs, the same job can be rerun safely if it fails due to bad data or a temporary system problem. Running the job again produces the same result instead of duplicating or corrupting the data. This property, known as **idempotency**, makes batch systems easier to recover and reason about.

This reliability is especially important for high-stakes outputs such as financial reports or supply chain plans, where numbers must be consistent and repeatable. When decisions and audits depend on the results, the ability to rerun a process with confidence becomes a core requirement rather than a convenience.

The critical weakness of batch architecture is **data staleness**. A forecast generated at midnight is based on data available up to midnight. If a significant demand spike occurs at 10:00 AM the next day (for example due to a viral social media trend), the batch forecast remains oblivious until the next scheduled run. In environments like intraday energy trading, high-frequency algorithmic trading, or ride-hailing supply positioning, this latency is unacceptable and can lead to significant lost revenue or system instability.

Stream processing (real-time) architecture

For operational decisions requiring immediate responsiveness, such as credit card fraud detection, network anomaly detection, or dynamic pricing, streaming architectures are the right choice. These systems process data as continuous unbounded streams rather than finite batches, minimizing the time-to-insight.

Event-driven inference and state management

Streaming architectures split the work in two. A high-throughput message bus (like Apache Kafka, Amazon Kinesis, or Apache Pulsar) handles **ingestion**: it accepts events as they arrive (a rider opening the Glide app, a driver's GPS ping, a completed trip) and holds them in an ordered, durable log. A stream processing engine (Apache Flink, Spark Structured Streaming) handles consumption: it reads events back off that log and runs the feature computation and inference. The separation matters: the bus can absorb a traffic spike that would overwhelm the processor, and several consumers can read the same stream independently.

Unlike image classification, where each input is independent, time series features are inherently stateful. Calculating a simple feature like a *7-day moving average* or a *count of login attempts in the last 5 minutes* requires maintaining a rolling window of history across distributed nodes. Engines like Flink use embedded state stores (for example RocksDB) to persist this state locally on workers, ensuring low-latency access.

Inference can occur in two modes:

- **Event-driven:** Each incoming event triggers a model prediction immediately. This offers the lowest latency but can overwhelm the system during traffic spikes.
- **Micro-batching:** Events are grouped into small windows (for example 1 second) to balance latency with throughput, allowing for vectorized operations similar to batch processing but on a much smaller scale.

The training-serving skew challenge

Streaming makes **Training-Serving Skew** especially hard to manage. The logic used to compute features in the batch training pipeline (often written in Python/Pandas) must be mathematically identical to the logic used in the streaming inference pipeline (often written in Java/Scala/Flink SQL). Any divergence, for example, a slight difference in how time windows are rounded, can lead to model degradation that is notoriously difficult to debug. Platforms like Uber's Michelangelo address this by using a shared **Domain Specific Language** (**DSL**) that compiles to both Spark (for batch) and Flink (for streaming), guaranteeing consistency.

Hybrid approaches: lambda and kappa architectures

To balance the stability of batch processing with the responsiveness of real-time systems, sophisticated organizations have developed hybrid patterns:

- **Lambda architecture:** This system maintains two parallel pipelines. It has a batch layer for accuracy and a speed layer for low latency.
- **Kappa architecture:** This system treats all data as a single stream. It simplifies the stack by using one pipeline for both real-time processing and historical replay.

Let's go into more detail about these two patterns:

Lambda processing is an overloaded term in Computer Science. While AWS Lambda refers to serverless functions and Lambdas in programming languages such as Java and Python refer to anonymous functions, in the context of deployment, Lambda Architecture is a hybrid design, used to balance accuracy and speed, which maintains two parallel pipelines:

- A **batch layer** (**cold path**) that processes the full history into an accurate Master Dataset
- And a **speed layer** (**hot path**) for low-latency estimates that compensate for the batch layer's staleness

For example, Instacart uses a variation of this for availability forecasting: a batch model predicts long-term trends, while a real-time layer adjusts scores based on immediate item not found signals. The operational overhead of maintaining two distinct codebases often leads to logic divergence.

Modern systems increasingly favor the Kappa Architecture, which simplifies the stack by treating all data as a stream. History is simply a stream stored on disk (for example in a Kafka topic with infinite retention). When retraining or backfilling is required, the system

replays the historical stream through the same pipeline used for live data. This ensures perfect consistency between offline training and online serving logic but requires a mature streaming infrastructure capable of high-throughput replay.

Figure 10.2 contrasts the two data flows.

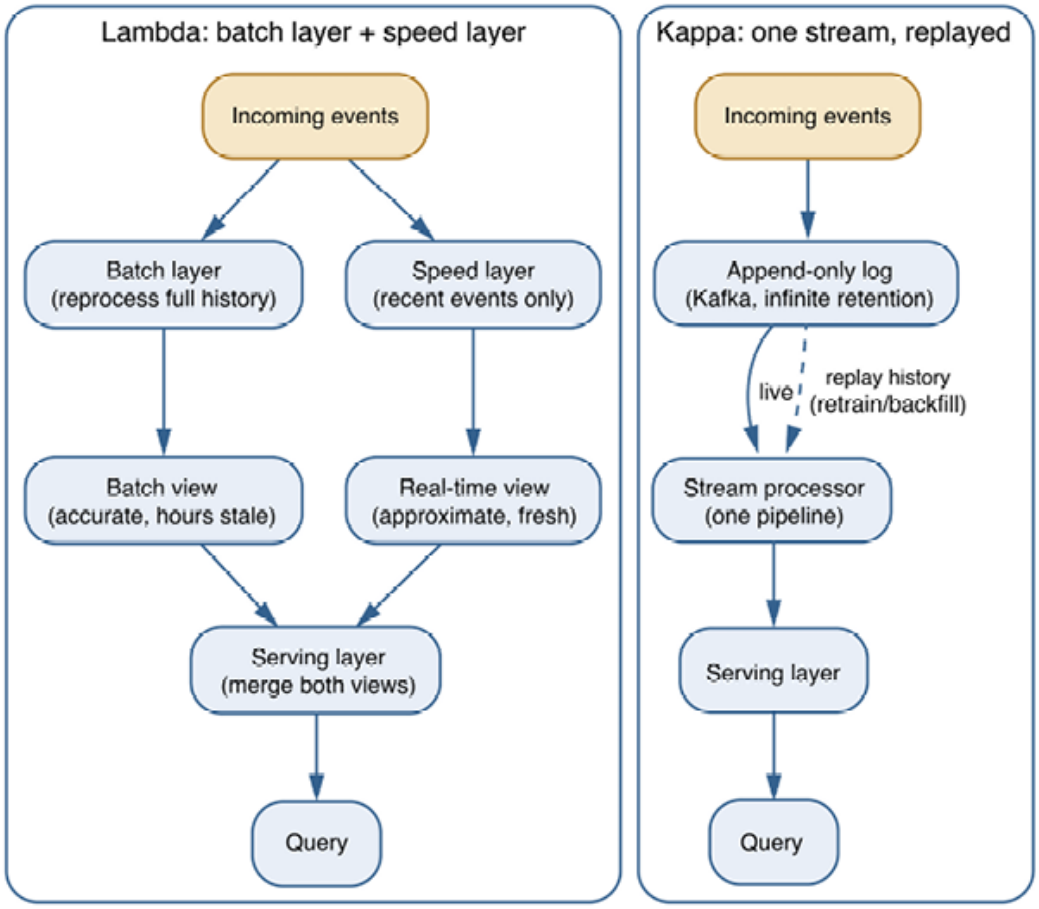


Figure 10.2: How data moves through each hybrid

Lambda splits incoming events into two paths: a batch layer reprocesses the full history into an accurate but hours-stale batch view, while a speed layer processes only recent events into a fresh but approximate real-time view; the serving layer merges both at query time. Kappa keeps one path: events land in an append-only

log, a single stream processor reads them live, and reprocessing for retraining or backfill replays the same log through the same pipeline (the dashed arrow). Lambda's two paths are the source of both its accuracy and its maintenance cost; Kappa's single path is the source of both its consistency and its dependence on mature streaming infrastructure.

Jay Kreps, who co-created Apache Kafka at LinkedIn, proposed it in 2014 to escape exactly the dual-codebase problem of Lambda. His argument: if the stream processor can replay history as easily as it reads live data, the batch layer is redundant. A Kappa surge-pricing system for Glide works this way. Every ride request, driver position update, and completed trip is appended to one Kafka log. The surge model consumes that log live to set prices. When the team retrains, they point the same consumer at the start of the log and replay months of history through the identical feature and inference code. There is no second pipeline to drift out of sync.

Live serving and historical replay use the same code; they differ only in where the consumer starts reading.

```
from kafka import KafkaConsumer
import json
def process_event(event):
    # The single pipeline: feature computation and inference.
    # Identical for live traffic and for historical replay.
...
def run(mode):
    # Kappa's trick: 'live' and 'replay' run the same code,
    # differing only in where the consumer starts reading the log.
    start = "latest" if mode == "live" else "earliest"
    consumer = KafkaConsumer(
        "glide_trip_events",
        bootstrap_servers="kafka:9092",
```

```
        group_id=f"surge_model_{mode}",
        auto_offset_reset=start,
        value_deserializer=lambda b: json.loads(b),
    )
    for message in consumer:
        process_event(message.value)
# Live serving reads only new events:       run("live")
# Retraining replays the full retained log:  run("replay")
```

This is a stripped-down sketch: it needs a running Kafka cluster, and a production Kappa system runs this logic inside a stream processor such as Flink or Spark Structured Streaming, which add checkpointing, exactly-once delivery, and parallel replay across partitions. The point here is only the shared code path, which is what gives Kappa its consistency guarantee.

Feature	**Batch Processing**	**Stream Processing**	**Hybrid (Lambda)**
Latency	Hours to Days	Milliseconds to seconds	Variable (Base + Adjustment)
Throughput	High (high efficiency)	Variable (Event-dependent)	High
Complexity	Low (stateless logic possible)	High (state management required)	Very High (Dual pipelines)

Use Case	Inventory Planning, Financials	Fraud Detection, Dynamic Pricing	Logistics, Ride-Sharing Supply
Correction	Easy (Re-run job)	Hard (Replay stream/Kappa arch)	Desynchronization risk

Table 10.3: Comparison of Deployment Architectures

Figure 10.3 turns that comparison into a decision procedure.

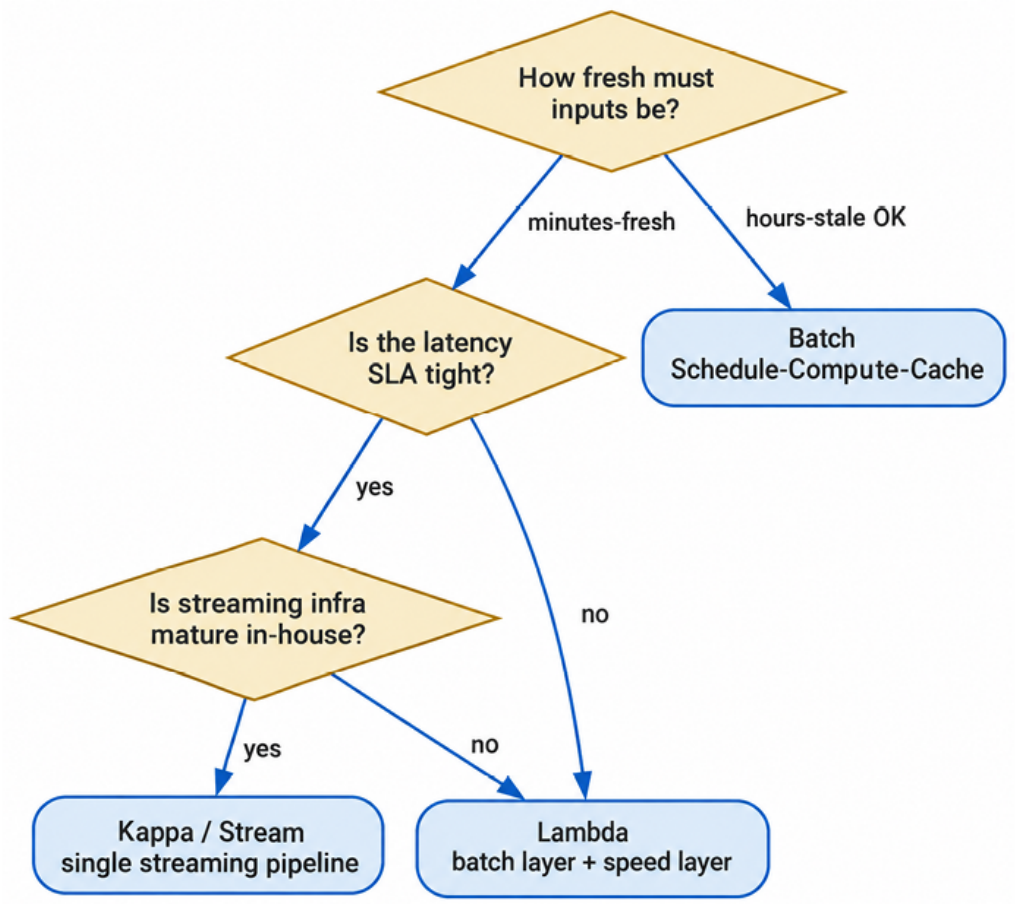


Figure 10.3: A decision tree for choosing a deployment architecture

Each diamond is a question about the use case; follow the answer edges down to a recommended architecture.

Start with freshness. If hours-old data is acceptable, batch processing is almost always right: it is the cheapest and simplest option, and it reruns cleanly. If inputs must be fresher, the latency SLA decides the next step. A loose SLA can be met by adding a thin speed layer on top of a batch job, the Lambda setup. A tight SLA points to streaming, and the final question is whether your team already operates mature streaming infrastructure. If it does, Kappa keeps the stack simple with a single pipeline. If it does not, Lambda is the pragmatic fallback, because the batch layer keeps working while the streaming layer matures.

Two business patterns sharpen the choice between the hybrids. Finance and regulated reporting favor Lambda: the batch layer is an auditable, reproducible record while the speed layer absorbs intraday updates without exposing the official numbers to streaming bugs. Event-driven and IoT systems, such as fraud scoring or sensor anomaly detection, favor Kappa: the data is born as a stream, latency budgets are tight, and one pipeline avoids the training-serving skew that two separate codebases invite.

This decision tree operationalizes the Freshness-Latency Matrix named at the start of the section: every question trades freshness or latency against cost and complexity.

Architecture defines the pipes of our system, but the quality of what flows through them determines the value of the forecast. While distributed frameworks handle the volume, we need specialized data engineering patterns to handle the complexity of time.

Data engineering

Stage 2 of the pipeline is data engineering. For Glide, it is about trust in the inputs. The demand model leans on features like recent trip counts per zone, local weather, and event calendars, and each must reflect what was actually known at forecast time, not what the database holds now. It is row 2 of the stack in Table 10.2 the two tools this section covers, a Feature Store and a validation layer, exist to keep that promise.

In time series forecasting, the quality of the data infrastructure is often more determinant of success than the choice of model architecture. The temporal nature of the data introduces unique challenges in feature engineering, specifically regarding leakage, point-in-time correctness, and the handling of irregularities.

Point-in-time correctness and time travel

A pervasive and dangerous failure mode in production forecasting is **temporal data leakage**. This occurs when a model is trained using data that would not have been available at the moment of inference, leading to Look-Ahead Bias (*Chapter 9*) and unrealistically high offline performance that collapses in production.

- **The average trap:** If you teach a model to predict sales for **Wednesday**, but give it the average sales for the **whole week** (Monday to Sunday), you are letting it see Sunday's sales to predict Wednesday's. In the real world, when Wednesday comes, you don't know Sunday's sales yet, so the model fails.

- **The time travel trap:** Databases often overwrite old info with new info. If a customer moves from New York to London today, the database might say they always lived in London. If you train a model on last year's data, it might think a customer living in London bought items that were actually bought in New York. This confuses the model because the *cause* (location) doesn't match the *effect* (purchase).
- **Point-in-Time Correctness** (**PIT**) means fixing this: ensuring the model only sees exactly what was known at that specific moment in the past, with no peeking at the future or current updates.

A more subtle form of leakage arises from **mutable data**. Data Warehouses are often mutable; if a customer changes their *Home City* from New York to London today, a standard SQL update might overwrite the record. When a Data Scientist trains a model on historical data from last month, a naive join will associate the customer's *current* city (London) with their *past* behavior (in New York). This destroys the causal link required for accurate prediction.

The Feature Store solution

To solve the time travel problem, production-grade systems use a tool called a **Feature Store** (like Feast, Tecton, or Databricks). Instead of just overwriting data (e.g., updating a user's location), it keeps a permanent log of every change, marking exactly when a value started (`valid_from`) and ended (`valid_to`).

When you need to train a model, the Feature Store performs what is called an *AS OF* join. It looks at the timestamp of a specific past

event and asks: *What was the value of this feature at that exact moment?*

It retrieves the data as it existed then, ignoring any changes that happened afterward. This allows your model to effectively time travel during training, ensuring it learns from the past exactly as it occurred (immutable log).

When generating a training dataset for a prediction at time t, the Feature Store retrieves the specific feature value where $t_{start} \leq t < t_{end}$. This accurately reconstructs the state of the world as it existed at the millisecond of the prediction, effectively allowing the model to time travel during training.

To make this concrete, follow one zone through a change. SF-07 is a quiet residential zone until a concert venue opens nearby in May 2023, after which its evening demand looks completely different. A mutable `zone_profile` table keeps one row per zone, so the update overwrites history. Queried while assembling training data in January, the row reads:

`zone_id`	category	`updated_at`
SF-07	residential	2021-03-01

Queried again in August, after the venue opened, the same row reads:

`zone_id`	category	`updated_at`
SF-07	nightlife	2023-05-12

A model assembled in August and trained on January's trips will, with a naive join, label every January ride in SF-07 as nightlife demand, for a venue that did not yet exist. The Feature Store never overwrites; it keeps both facts as a log:

`zone_id`	category	`valid_from`	`valid_to`
SF-07	residential	2021-03-01	2023-05-12
SF-07	nightlife	2023-05-12	(open)

An AS-OF join for a January event matches the first row, an AS-OF join for a July event matches the second, and each training row sees the zone as it actually was.

We will now look at implementing point-in-time correctness with feast. Feast abstracts the complexity of point-in-time joins. The user provides an `entity_df` containing the entities and timestamps for which they want historical features. Feast's `get_historical_features` method then scans the offline store to find the feature values that were valid *immediately prior* to those timestamps.

```
from feast import FeatureStore
import pandas as pd
from datetime import datetime
# Initialize the feature store connection
store = FeatureStore(repo_path=".")
# Define the entity dataframe:
# one row per demand forecast we want to train on.
# 'event_timestamp' is the time the forecast WOULD have been made.
entity_df = pd.DataFrame.from_dict({
```

```
    "zone_id": ["SF-07", "SF-12", "SF-07"],
    "event_timestamp": [
        datetime(2023, 4, 12, 10, 59, 42),
        datetime(2023, 4, 12, 8, 12, 10),
        datetime(2023, 4, 12, 16, 40, 26)
    ]
})
# Retrieve historical features. Feast performs a point-in-time (AS OF
# It finds the value of 'trips_prev_hour' for zone SF-07 that was val
# immediately prior to 10:59:42 on April 12, preventing future leakag
training_df = store.get_historical_features(
    entity_df=entity_df,
    features=[
        "zone_hourly_stats:trips_prev_hour",
        "zone_hourly_stats:avg_surge",
        "zone_daily_features:active_drivers"
    ],
).to_df()
print(training_df.head())
```

This snippet is not runnable on its own: it assumes a Feast repository is already defined and applied, with the `zone_hourly_stats` and `zone_daily_features` feature views, a registry, and an offline store (the Feast quickstart walks through that setup). Given that, it ensures the training data matches the information available at inference time.

For environments without a dedicated Feature Store, the `pandas.merge_asof` function provides a local implementation of this logic. It matches the nearest key from the left DataFrame to the right DataFrame without going over, ensuring that we only join data that was available *prior* to the timestamp.

Point-in-time correctness resolves only one part of the data engineering challenge. The second critical problem a feature store solves is training-serving skew, which was introduced in the

streaming section. Skew occurs when feature calculation logic diverges between the batch training environment and the live inference environment. A feature store eliminates this divergence by acting as a central repository. Every feature is defined once. That single definition serves both the offline training query and the online inference lookup. Consequently, the batch and streaming code paths remain perfectly aligned. Glide's `trips_prev_hour` is computed once in the store and read universally, rather than being written in a Pandas training pipeline and then duplicated in a Flink streaming application.

Handling data gaps, irregularities, and censored data

Real-world data is rarely perfect. Systems crash, sensors break, and networks fail, leaving holes in your timeline. It is tempting to use simple fixes to fill these gaps (imputation), such as just copying the last number you saw (forward filling) or plugging in the average value. However, these simple fixes can be misleading.

In production pipelines, it is critical to distinguish between *True Zeros* (a zone with genuinely no ride requests at 4 a.m.) and *Missing Data* (the events pipeline dropped that interval). Pipelines should generate Indicator Features, boolean flags indicating if a value was imputed, so the model can learn to lower its confidence during data outages.

A critical failure mode for a platform like Glide is the **supply gap**. When no driver is near a zone, completed trips drop to zero. Rider

demand does not: people still open the app, see no cars nearby, and give up.

This creates Censored Data. In statistics, data is censored when we only know part of the value. Here the true demand is hidden: we know it was at least the number of trips we completed, but not how much higher it would have been with enough drivers.

Training a model on raw trip counts introduces a negative bias. The model learns that *no drivers means no demand.* This causes a Death Spiral: the model predicts low demand for the zone, so Glide positions fewer drivers there, so even more riders give up, so recorded trips fall further. The model eventually abandons a zone that riders actually wanted.

Calculating rolling statistics in the presence of zeros or missing data requires care. If a zero marks a supply gap rather than genuine quiet, including it in a demand calculation biases the forecast downward. We must often calculate rolling means on non-zero values or handle the supply-gap condition explicitly.

```
# Efficiently calculating a rolling mean excluding zeros
# This is useful for estimating demand when zeros represent  supply g
import pandas as pd
import numpy as np
# Hourly completed trips for one Glide zone
df = pd.DataFrame({'trips': [100, 0, 150, 0, 200, 300, 0]})
# Convert 0s (supply gaps) to NaNs so they are ignored by the rolling
df['demand_adjusted'] = df['trips'].replace(0, np.nan)
# Calculate rolling mean. min_periods=1 ensures we get a value even i
# The 'rolling' function naturally skips NaNs in the mean calculation
df['rolling_demand'] = df['demand_adjusted'].rolling(window=3, min_pe
# Impute the zeros: If trips were 0, assume demand was equal to the r
df['imputed_demand'] = df[trips].mask(
```

```
    df[trips] == 0, df['rolling_demand'])
print(df)
```

One caution on that `min_periods=1` guarantees a value, not a reliable one. A stray zero or two inside the window is harmless, and the rolling mean of the survivors stands in well. But when a zone goes dark for many consecutive hours, the window can come to rest on a single distant observation, and the imputed demand becomes a guess dressed as an estimate. For longer gaps, reach for time-aware interpolation (pandas interpolate with `method='time'`) to bridge short runs, and always pair imputation with the missing-value indicator described earlier, so the model can discount the rows it cannot trust.

Data validation with Pandera

The Rules of Machine Learning by Google explicitly state: *Make sure your pipeline is solid end to end* before focusing on complex modeling. For time series, this means validating temporal continuity, schema consistency, and data invariants.

Pandera is a Python library that provides a flexible and expressive API for performing data validation on DataFrame-like objects. Unlike static type checking, Pandera validates the *content* of the data at runtime, catching drift or corruption in production pipelines.

```
import pandera as pa
from pandera.typing import Series
import pandas as pd
# A schema for Glide's hourly zone-demand feed
class ZoneDemandSchema(pa.DataFrameModel):
    unique_id: Series[str] = pa.Field(coerce=True)       # Glide zone
```

```
    ds: Series[pd.Timestamp] = pa.Field(coerce=True)
    # hourly trip count, non-negative :
    y: Series[float] = pa.Field(ge=0, nullable=False)
    # Custom check:  timestamps must be monotonically increasing
    @pa.check("ds")
    def check_time_order(cls, series: Series) -> bool:
        return series.is_monotonic_increasing
# Validate incoming data (incoming_zone_demand.parquet ships with the
try:
    df = pd.read_parquet("incoming_zone_demand.parquet")
    TimeSeriesSchema.validate(df)
    print("Data passed validation.")
except pa.errors.SchemaError as e:
    print(f"Data validation failed: {e}")
```

Ensuring data correctness handles the quality dimension of production. However, enterprise forecasting also requires handling the quantity dimension. We must now address how to scale these calculations from a single laptop to millions of unique series.

Scaling strategies

Stage 3 of the pipeline is training and scaling. For Glide, this is the jump from one city in a notebook to every zone in all 40 cities at once. The same model must fit and serve all of them without its cost growing in proportion.

Scaling time-series systems involves two distinct dimensions: request volume (throughput) and cardinality (the number of unique series). Glide forecasts demand for every zone in all 40 cities, across three vehicle classes: several hundred thousand unique series, each refreshed continuously.

Local vs. global modeling paradigms

A critical architectural decision is the choice between Local and Global modeling paradigms.

- **Local Models:** The traditional approach, dominated by statistical methods like ARIMA and ETS, involves training a separate, independent model for each time series. For Glide's hundreds of thousands of zones, the system trains one model per zone. This approach is computationally expensive to train but embarrassingly parallel. Its major downside is the inability to generalize; a newly launched zone with no history cannot be modeled (the **Cold Start problem**).
- **Global Models:** The modern paradigm involves training a single model (typically a Deep Learning model like DeepAR, TFT, or N-BEATS, or a Gradient Boosted Tree like LightGBM) on all time series simultaneously. The model learns a single set of weights that minimizes the loss across all series. This allows for **Cross-Series Learning**, where the model learns latent patterns (the morning commute peak, or the demand on New Year's Eve) from data-rich zones and applies them to data-poor zones, effectively solving the cold-start problem.

Distributed forecasting with ray

To handle the computational load of training global models or fitting millions of local models, distributed computing frameworks like **Ray** have become standard. Ray allows Python-native code to scale from a laptop to a cluster with minimal changes, supporting both stateful

actors (for global model training) and stateless tasks (for parallel local inference).

For local models, Ray can parallelize the fit function across thousands of cores. For global models, Ray Train (formerly Ray SGD) allows for distributed training of PyTorch or TensorFlow models across multiple GPUs.

```
import ray
import pandas as pd
import numpy as np
from statsforecast import StatsForecast
from statsforecast.models import AutoARIMA
from statsforecast.utils import generate_series
# Initialize Ray cluster connection
ray.init(address='auto', ignore_reinit_error=True)
# Synthetic panel: each unique_id stands for one Glide zone's demand
series = generate_series(n_series=1000, freq='h',
                         min_length=200, max_length=200)
# Define a function to fit a model to a chunk of data.
# The @ray.remote decorator turns this function into a distributed ta
@ray.remote
def fit_forecast_chunk(df_chunk):
    # StatsForecast is optimized for high performance
    model = StatsForecast(models=[AutoARIMA()], freq='h')
    model.fit(df_chunk)
    return model.predict(h=24)
# Split data into chunks by unique_id for parallel processing
unique_ids = series['unique_id'].unique()
# Create 10 chunks of IDs
chunk_ids = np.array_split(unique_ids, 10)
# Create list of DataFrames
chunks = [series[series['unique_id'].isin(ids)] for ids in chunk_ids]
# Launch distributed tasks. This happens asynchronously.
# Ray schedules these tasks across the cluster.
futures = [fit_forecast_chunk.remote(chunk) for chunk in chunks]
# Retrieve results. ray.get() blocks until computations are complete.
forecasts = ray.get(futures)
```

```
full_forecast = pd.concat(forecasts)
print(full_forecast.head())
```

Note that libraries like `statsforecast` have built-in Ray backends that handle this distribution automatically for the user.

Hierarchical reconciliation at scale

Retail and supply-chain data naturally follows an aggregation hierarchy, and independent forecasts at each level rarely sum up correctly. *Chapter 6* covers the statistical reconciliation methods that restore this coherency. The primary production concern is scaling the computation. Reconciling millions of series requires a distributed batch job where workers partition the tree, reconcile sub-trees in parallel, and assemble the totals. This architectural requirement places reconciliation firmly in the Schedule-Compute-Cache batch path rather than the streaming path.

We have trained a global model at scale and seen where hierarchical reconciliation fits in the batch path. The next challenge is introducing this model to the real world without disrupting operations or eroding user trust.

Deployment strategies

Stage 4 of the pipeline is deployment. For Glide, it is the moment the new demand model meets real traffic. Driver pay and rider wait times now depend on it, so it cannot simply be swapped in.

Deploying a new model version is a high-risk operation. Unlike code changes, model updates can degrade performance in subtle, non-deterministic ways. A model might have a lower global RMSE but perform catastrophically on a specific, critical customer segment.

Shadow deployment

For high-risk applications, **Shadow Deployment** (or Champion/Challenger) allows validation on live traffic without user impact. In this pattern, the inference router sends requests to both the Champion (current Production model) and the Challenger (Candidate model). Only the Champion's response is returned to the user. The Challenger's response is logged asynchronously for analysis.

This setup allows for Forward Testing on real-world data, which is superior to backtesting as it exposes the model to current production latency, data quality issues, and market regimes that may not be present in historical data.

Let's work through an implementation with FastAPI Middleware.

We can implement a lightweight shadow traffic mechanism using FastAPI middleware. This intercepts requests, spawns a background task to query the shadow model, and logs the result without blocking the main response.

```
import asyncio
from fastapi import FastAPI, Request
import httpx
app = FastAPI()
CHAMPION_URL = "http://glide-demand-v1:8000/predict"
```

```python
CHALLENGER_URL = "http://glide-demand-v2:8000/predict"
async def query_shadow_model(payload: dict):
    """Query the challenger model and log the result."""
    try:
        async with httpx.AsyncClient() as client:
            response = await client.post(CHALLENGER_URL, json=payload
            # Log prediction for offline analysis
            # (for example  to S3 or Kafka)
            print(f"Shadow Prediction: {response.json()}")
    except Exception as e:
        print(f"Shadow request failed: {e}")
@app.middleware("http")
async def shadow_traffic_middleware(request: Request, call_next):
    # Clone the request body because consuming it makes it unavailabl
    # This is a simplification;
    # handling streaming bodies requires more complex logic
    body_bytes = await request.body()

    # Process the main request (Champion)
    response = await call_next(request)

    # Fire and forget the shadow request using the same payload
    # In production, use BackgroundTasks
    # or a message queue (Kafka) for resilience
    asyncio.create_task(query_shadow_model(body_bytes))

    return response
```

The middleware above is a teaching sketch. Reading `request.body()` buffers the entire payload and consumes the stream: fine for the small JSON of a forecast request, but a production version must handle large or streamed bodies and restore the body for the downstream handler. It should also use FastAPI's BackgroundTasks or a message queue instead of a bare asyncio.create_task, which drops the shadow request if the process restarts. The pattern itself, mirroring traffic to the challenger and logging asynchronously, allows safe evaluation of new models under real load.

Shadow deployment validates computational stability and statistical distribution. It cannot measure operational outcomes because the predictions of the challenger model do not influence the environment. To evaluate metrics such as rider wait times, the model must serve live traffic. This transition is managed through gradual rollouts and A/B testing. A router directs a small fraction of live traffic, such as 5 percent, to the new model. The champion model serves the remainder. Teams monitor both cohorts. If the challenger model performs well, the routing fraction is incrementally increased to 25 percent, 50 percent, and finally 100 percent. This bounds the risk of unseen failures while providing the empirical data required for a complete replacement.

Model serialization and MLflow

To manage the lifecycle of models, especially complex ensembles or custom logic, **MLflow** provides a robust standard. While `MLflow` has built-in support for Scikit-Learn or PyTorch, time series models often require custom *flavors*. For instance, a `StatsForecast` object holding a fitted ARIMA model for each Glide zone is not a standard Scikit-Learn estimator and requires specific handling for serialization.

We can wrap these models using `MLflow`'s `PythonModel` interface (`PyFunc`). This allows us to define a custom predict method that handles the specific data formatting required by the forecasting library (for example, converting a JSON request into a DataFrame), while exposing a standard REST API signature to the deployment environment. The example below runs the full save-and-load cycle on a dummy forecaster, so it executes end to end; the empty

`load_context` marks where a production wrapper would read its fitted `StatsForecast` object from artifacts.

```
import os
import tempfile
import mlflow.pyfunc
import pandas as pd
import numpy as np
class DemandForecaster(mlflow.pyfunc.PythonModel):
    def load_context(self, context):
        pass  # production loads the fitted model from artifacts here
    def predict(self, context, model_input):
        horizon = int(model_input["h"].iloc[0])
        return pd.DataFrame({
            "hour": range(1, horizon + 1),
            "forecast_trips": np.random.rand(horizon) * 100,
        })
# Save the wrapped model to a fresh directory in MLflow's format
model_path = os.path.join(tempfile.mkdtemp(), "glide_demand_model")
mlflow.pyfunc.save_model(path=model_path, python_model=DemandForecast
# Load it back and predict through the standard PyFunc interface
loaded = mlflow.pyfunc.load_model(model_path)
request = pd.DataFrame({"h": [6]})
print(loaded.predict(request))
```

The `save_model` call writes the wrapper and its dependencies into one self-contained directory; `load_model` reconstructs it anywhere, and the deployment environment calls predict without importing the forecasting library itself. Swapping the dummy for a real `StatsForecast` object loaded in `load_context` changes none of this.

Safe deployment ensures the model starts healthy. To ensure it stays healthy, we need observability strategies that can handle the unique feedback lag of time series, where the truth often arrives weeks after the prediction.

Monitoring and observability

Stage 5 of the pipeline is monitoring. For Glide, it runs into the feedback lag head-on. Whether a 15-minute-ahead demand forecast was right is known within the hour; whether a week-ahead forecast was right is not known for a week.

Monitoring time-series systems is fundamentally different from monitoring standard web services due to the **Delayed Feedback Problem**. In many use cases, the ground truth is not immediately available. For example, in supply chain forecasting, the accuracy of a forecast generated today for *sales two weeks from now* cannot be calculated for two weeks. This lag renders real-time accuracy monitoring (calculating RMSE in real-time) impossible. We must rely on **Proxy Metrics**.

Proxy metrics and drift detection

To detect issues before ground truth arrives, we monitor the statistical properties of the inputs and outputs.

- **Prediction Drift:** If a demand forecasting model usually predicts values in the range [100, 200] and suddenly shifts to [5,000, 6,000] without a corresponding shift in known drivers (for example promotions), it is a strong signal of model failure or data corruption.
- **Feature Drift:** Monitor the distributions of key inputs. A sudden drop in the *Average Temperature* feature to absolute zero indicates a sensor failure, not a weather event.

- **Concept Drift vs. Seasonality:** Time series are inherently non-stationary. A standard drift detector (for example Kolmogorov-Smirnov test) comparing *Last Week's Data* to *This Week's Data* will trigger false alarms due to natural weekly cycles. The solution is to monitor **Deseasonalized Residuals**. Fit a simple baseline model (for example Seasonal Naive) and subtract it from the live data. Apply drift detection algorithms (for example Page-Hinkley, ADWIN) to the *residuals*. If the residuals drift, it indicates a change in the underlying process *unexplained* by seasonality.

Performance estimation (CBPE)

Advanced monitoring tools like **NannyML** provide algorithms like **Confidence-Based Performance Estimation (CBPE)**. This technique estimates the realized error (for example MAE) of a model in the absence of ground truth by analyzing the model's output confidence (if available) or the density of the input data in the feature space. If production data falls into regions of the feature space where the model was previously known to perform poorly (during training/testing), the estimated error metric is increased, triggering an alert.

```
import nannyml as nml
import pandas as pd
import numpy as np
# Reference data: a past period where the model's error is known and
# It carries the model's features, its predictions, the actuals, and
reference_data = pd.DataFrame({
    "trips_prev_hour": np.random.rand(1000),
    "avg_surge": np.random.rand(1000),
```

```
    "y_pred": np.random.rand(1000),
    "y_true": np.random.rand(1000),
    "timestamp": pd.date_range("2023-01-01", periods=1000, freq="h"),
})
# Analysis data: live production data, same columns but no y_true yet
# Here trips_prev_hour has drifted upward.
analysis_data = pd.DataFrame({
    "trips_prev_hour": np.random.rand(1000) + 0.5,
    "avg_surge": np.random.rand(1000),
    "y_pred": np.random.rand(1000),
    "timestamp": pd.date_range("2023-02-12", periods=1000, freq="h"),
})
# DLE estimates error from the features, so feature_column_names must
# real columns in reference_data. For regression (forecasting) use DL
estimator = nml.DLE(
    feature_column_names=["trips_prev_hour", "avg_surge"],
    y_pred="y_pred",
    y_true="y_true",
    timestamp_column_name="timestamp",
    metrics=["rmse"],
    chunk_size=500,
)
# Fit on reference data (where we have actuals)
estimator.fit(reference_data)
# Estimate performance on analysis data (live data, no actuals yet)
estimated_performance = estimator.estimate(analysis_data)
# Visualize the estimated RMSE over time
estimated_performance.plot().show()
```

This allows for proactive alerting on model degradation before actuals are observed.

When CBPE signals performance degradation, the standard response involves retraining the model on a recent batch of data. Highly non-stationary environments require a different paradigm. Online learning updates the model incrementally as each observation arrives. This process eliminates the delay of scheduled retraining runs. The river library in Python provides a dedicated

framework for this method. It processes data streams one event at a time and adjusts model parameters continuously. This approach suits high-frequency domains where the data generating process shifts rapidly.

Monitoring is not free. Logging every prediction, storing reference distributions, and running drift detectors across Glide's hundreds of thousands of zones costs real compute and storage, and every drift alarm that triggers a retraining run adds to the bill. Monitoring everything at full granularity does not scale; the usual answer is to tier it, tracking cheap aggregate metrics across all zones and reserving per-series detection and tight alerting for the high-revenue zones where a wrong forecast costs the most. Set against the alternative, the spend is modest: an undetected regime change is what turned a forecasting error into Zillow's write-down, so the budget should be sized by the business cost of finding out late, not by the compute cost of monitoring.

Observability helps us spot when a model drifts from reality. But the most dangerous failures are often not in the model itself, but in how the model interacts with the broader system to create vicious cycles.

Advanced failure modes and mitigation

This final section closes the loop in *Figure 10.1*, where the model's own forecasts feed back as a failure mode. For Glide, the loop is surge pricing. A high demand forecast raises prices, which

suppresses ride requests and repositions drivers, which changes the very demand the next forecast sees.

The most sophisticated failure modes in production are systemic issues arising from the interaction between the model and the environment.

Censored data and the demand death spiral

The supply gaps discussed in the data engineering section form a feedback loop if left unhandled. A model trained on raw completed trips interprets a supply-starved zone as low demand and allocates fewer drivers. This causes more riders to abandon the platform and degrades the subsequent training set. Breaking this cycle requires recovering the latent demand behind these censored observations. *Chapter 12* covers the survival analysis methods required for this task. The production mandate is to identify this spiral and route the demand signal through a censored-data model before it influences operational routing and pricing logic.

Feedback loops and the bullwhip effect

In supply chains, small forecasting mistakes at the retail level can grow into much larger problems further upstream. A minor change in customer demand, or even random noise such as a one-day sales spike, can cause retailers to overreact. Manufacturers then see these exaggerated orders and respond with large production changes. This

cascading amplification of small signals as they move up the supply chain is known as the bullwhip effect.

Figure 10.4 simulates it.

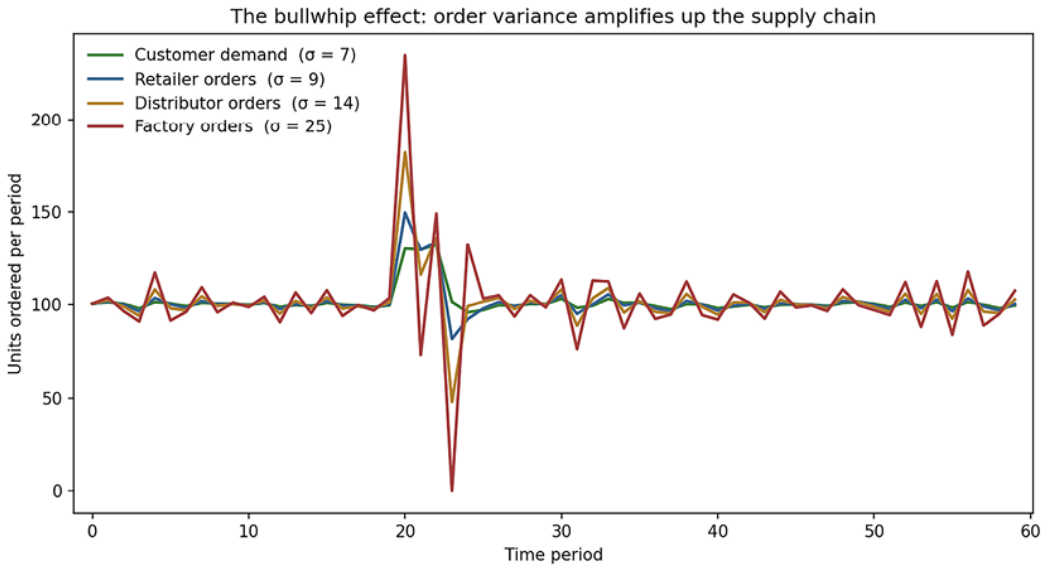


Figure 10.4: A stylized four-echelon simulation

Customer demand (green) is steady apart from one brief three-period spike. Each tier upstream reorders to cover the demand it sees plus an overreaction to how fast that demand is changing, so the spike grows: the retailer's orders swing wider than the customer's, the distributor's wider still, and the factory's widest of all. The standard deviation of orders, in the legend, roughly triples from customer to factory. The steady stretches show the same effect in miniature; ordinary noise amplified into visible swings.

When AI models react too aggressively to short-term fluctuations instead of underlying demand, they can make this problem worse. By treating noise as a real trend, the system may trigger oversized restocking orders, creating instability throughout the supply chain rather than smoothing it.

To mitigate this, production systems often employ Signal Smoothing or Control Tower logic. Rather than simply minimizing forecast error, the optimization function for inventory ordering should include a penalty term for variance in the orders themselves. This effectively acts as a low-pass filter, damping the high-frequency noise from the forecast before it translates into physical logistics actions. Simulation frameworks are often used to stress-test these policies against various demand scenarios before deployment.

Being aware of systemic risks like feedback loops allows for the design of resilient systems that stabilize rather than destabilize operations. We have now covered the full production lifecycle; let us review the journey.

Summary

In this chapter, we crossed the most significant chasm in the machine learning lifecycle: the Production Gap. For the first nine chapters, our goal was to minimize error on a static test set. In this chapter, our goal shifted to building reliable, scalable systems that function in the messy, real-time world of business operations. We followed one system throughout: the demand model of Glide, a ride-hailing platform, traced from a notebook prototype to a monitored production service.

We began by defining the unique constraints of production time-series systems. We explored the Freshness-Latency Matrix, learning that the choice between Batch (Schedule-Compute-Cache) and Stream (Kappa/Lambda) architectures is a business decision driven by the cost of stale data.

We then explored data engineering. We saw how Feature Stores allow us to time travel, ensuring PIT correctness and preventing the silent killer of model performance: data leakage. We also addressed Scaling, using distributed frameworks like Ray to train and serve a model across hundreds of thousands of series, and saw that hierarchical reconciliation, covered in *Chapter 6*, becomes a distributed batch workload at that scale.

Finally, we operationalized our models. We moved away from deploy and pray to rigorous Shadow Deployments (Champion/Challenger) and implemented observability stacks capable of detecting concept drift even when ground truth is delayed. We closed by acknowledging the systemic risks of Feedback Loops, understanding that a model is an active participant in reality, not just an observer, and can stabilize or destabilize a supply chain.

Cost runs underneath every one of these stages and is easy to overlook. Real-time inference burns compute on every event, feature stores and reference datasets accumulate storage, and each retraining run adds to the bill. A forecast that is accurate but too expensive to produce is not production-ready; sizing the spend against the business value of the forecast belongs in the design, not the postmortem.

Being aware of systemic risks of feedback loops and the bullwhip effect allows for the design of more resilient systems that stabilize the business operations they serve. The transition from research to production is the transition from predicting the future to shaping it.

We have now completed the core journey of this book. You possess the skills to explore data, build state-of-the-art forecasting models,

and deploy them into production environments. However, time series analysis is broader than just forecasting. In the final part of this book, we expand our horizon beyond predicting what happens next.

In *Chapter 11*, we will apply our toolkit to a new set of high-value problems. We will explore Time Series Classification (identifying behaviors), Anomaly Detection (spotting failures before they happen), and Causal Inference (understanding the impact of our interventions). Finally, we will conclude with intermittent forecasting and survival analysis in *Chapter 12*.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers.

https://discord.gg/VXpz9YPUTW



11

Beyond Forecasting: Specialized Applications

In the previous chapters, we built a toolkit of forecasting methods. We started with classical statistical methods like ARIMA and ETS to capture linear trends and seasonality. We advanced to powerful tree-based ensembles like XGBoost, using feature engineering to handle complex non-linearities. We scaled up to global deep learning architectures to learn across thousands of series, and finally, we entered the era of Foundation Models, using Zero-Shot inference to predict the future without training.

However, in many high-stakes environments, the most valuable insights come not from predicting the future, but from characterizing the present or explaining the past. A smart grid operator requires immediate awareness of anomalies, knowing instantly if a voltage drop is a sensor glitch or a line failure, rather than just a load forecast for the next hour. A quantitative trader often cares less about the exact price of an asset and more about classifying the current market regime as volatile or stable to trigger the correct algorithm. A Chief Marketing Officer needs to distinguish between a natural seasonal uptrend and the true causal impact of a multi-million dollar campaign.

In this chapter, we expand our toolkit beyond regression to tackle these structural and diagnostic challenges. We will move from asking *what will happen?* to asking *what is happening?* and *why did it happen?*

In this chapter, we will explore:

- Time series classification and structural pattern recognition
- Generative modeling with diffusion models
- Managing non-stationarity

- Graph and spatio-temporal modeling
- Neural ODEs for irregular time series
- Understanding causal inference

We begin by addressing the need to categorize current states rather than predicting future values. This section examines how to identify specific structural patterns within high-dimensional data to determine what is happening right now.

Time series classification and structural pattern recognition

Time Series Classification (**TSC**) asks what state a sequence represents, assigning it to one of a fixed set of labels.

Consider a high-frequency trading desk. While forecasting the exact closing price of an asset is fraught with stochastic noise, classifying the current market regime, distinguishing between a *mean-reverting stable* state and a *momentum-driven volatile* state, is a tractable problem that directly informs algorithmic execution. Similarly, in predictive maintenance, a facility manager is rarely interested in the precise vibration amplitude of a turbine bearing in the next millisecond. Their actionable insight comes from classifying the current vibration signature as *Normal*, *Inner Race Fault*, or *Cage Instability*. Here, the temporal sequence becomes a complex, high-dimensional signature of the system's physics instead of a path to a future value.

Early time series classification methods relied on distance-based approaches such as **Dynamic Time Warping** (**DTW**), which aligned sequences even when patterns occurred at different times and outperformed Euclidean distance on temporally misaligned data. Later, ensemble methods such as HIVE-COTE combined shapelets, intervals, dictionaries, and kernels into a single highly accurate framework, although their computational cost made them impractical for real-time applications. The ROCKET family transformed the field by replacing learned convolutions with thousands of randomly generated kernels and pairing them with simple linear classifiers. Successive variants such as MiniRocket and MultiRocket improved efficiency and accuracy, with MultiRocket achieving state-of-the-art performance while still training in seconds to minutes.

This progression, from distance metrics to meta-ensembles to random kernels, was mapped by the *Bake-offs*: exhaustive comparative studies from the University of East Anglia (UEA) and UC Riverside (UCR).

Self-supervised representation

Chapter 9's foundation models, Chronos, TimesFM, and Moirai, were pretrained on large corpora of unlabeled series. We used them there for zero-shot forecasting, but the encoder inside each one does something more general: it maps any input window to a fixed-length vector that captures level, trend, seasonality, and shape. That vector is not tied to forecasting. It is a feature set you can hand to any downstream model.

This is the bridge from forecasting to the classification and anomaly detection problems in this chapter. ROCKET's random kernels carry no knowledge of any particular domain: they extract generic features and lean on the linear head to learn what matters. A foundation-model embedding instead arrives pre-loaded with structure learned from millions of series, and one embedding serves several tasks. The same vector can be classified, scored for anomalies, or matched against past windows, with no task-specific representation to train.

We load the `BasicMotions` HAR panel and extract embeddings from Chronos, reusing the model and weights from *Chapter 9*:

```
import numpy as np
import torch
from sklearn.model_selection import train_test_split
from sktime.datasets import load_basic_motions
from chronos import ChronosPipeline
# The BasicMotions HAR panel, loaded as a 3D array
X, y = load_basic_motions(return_X_y=True, return_type="numpy3D")
X_train, X_test, y_train, y_test = train_test_split(
   X, y, test_size=0.3, random_state=42
)
# Same encoder weights used for zero-shot forecasting in Chapter 9
pipeline = ChronosPipeline.from_pretrained(
    "amazon/chronos-t5-tiny", device_map="cpu", torch_dtype=torch.float32
)
def embed_panel(panel):
    """Mean-pool Chronos encoder states into one vector per series."""
    vectors = []
```

```
    for sample in panel:
        channels = []
        for channel in sample:
            ctx = torch.tensor(channel, dtype=torch.float32)
            emb, _ = pipeline.embed(ctx)
            channels.append(emb[0].mean(dim=0))
        vectors.append(torch.cat(channels).numpy())
    return np.asarray(vectors)
```

The embed method returns the encoder's hidden states for the window. We mean-pool over time to get one vector per channel, then concatenate the channels into a single feature vector per series. This is a teaching sketch: it calls the encoder once per channel and pools with a flat mean. Production code would batch the encoder calls and could pool with attention instead.

With embeddings in hand, low-label classification is a standard scikit-learn fit. We keep five labeled series per class and train a logistic regression:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
E_train = embed_panel(X_train)
E_test = embed_panel(X_test)
# Low-label regime: keep only 5 labeled series per class
rng = np.random.default_rng(42)
few = np.concatenate([
    rng.choice(np.where(y_train == c)[0], size=5, replace=False)
    for c in np.unique(y_train)
])
head = LogisticRegression(max_iter=1000)
head.fit(E_train[few], y_train[few])
acc = accuracy_score(y_test, head.predict(E_test))
print(f"Accuracy with 5 labels/class: {acc:.3f}")
# Output:
Accuracy with 5 labels/class: 1.000
```

The embedding does the heavy lifting; the classifier is a thin linear layer over a representation that already separates the activities. Five labeled series per class are enough to reach full accuracy. BasicMotions is an easy dataset, so a MiniRocket pipeline reaches the same score: the embedding is not buying accuracy here. What it buys is the next task, which a classifier cannot do at all.

The same embeddings support anomaly detection without a single labeled anomaly. During commissioning you observe only normal behavior, so you fit a

novelty detector on those embeddings and flag anything that lands far from the normal cluster:

```
from sklearn.neighbors import LocalOutlierFactor
# Commissioning sees one behavior; treat it as the normal profile
normal_class = np.unique(y_train)[0]
detector = LocalOutlierFactor(n_neighbors=10, novelty=True)
detector.fit(E_train[y_train == normal_class])
# -1 marks a series that does not match the normal profile
flags = detector.predict(E_test)
print(f"Off-profile: {(flags == -1).mean():.0%} of test series")
# Output:
Off-profile: 42% of test series
```

The detector flags any test window whose embedding lands far from the commissioning cluster. Because it works in embedding space rather than on raw signal values, it responds to structural deviations, a new fault signature, or a sensor that has started clipping, even when the raw amplitude stays in its usual range. A MiniRocket classifier cannot stand in here: it only ever assigns one of the labels it was trained on, and an unseen fault has no label.

When labels are plentiful, a dedicated classifier is still the efficient choice. Building one means picking a library, and the Python ecosystem has recently split over how.

sktime vs. aeon

A critical development in the Python ecosystem is the divergence between `sktime` and `aeon`.

`sktime` remains the generalist and provides a unified, scikit-learn compatible interface for all time series tasks, including forecasting, annotation, and classification. It excels at composability, allowing you to build complex pipelines that mix forecasting transformers with classification models.

`aeon`, on the other hand, is more of a specialist toolkit. Spun off to focus specifically on time series machine learning (classification and clustering), it prioritizes performance. Its implementations of kernels and distance measures often use

Numba for **Just-In-Time** (**JIT**) compilation, making it much faster for heavy workloads.

For the following examples, we use `sktime` due to its widespread adoption and flexible pipeline architecture, but practitioners hitting performance bottlenecks should consider `aeon` as a drop-in replacement.

The human activity recognition problem

We will apply the ROCKET algorithm to a classic multivariate classification problem: **Human Activity Recognition** (**HAR**). The dataset consists of 3-axial linear acceleration and 3-axial angular velocity from a smartwatch. The goal is to classify the activity (*Walking*, *Standing*, *Running*, *Badminton*) based on the structural *shape* of the sensor data.

1. First, we load the dataset. We use the `BasicMotions` dataset as a proxy for HAR, which contains six dimensions (variables) per sample. Each row is an independent recording, a self-contained activity event rather than a slice of one long series. That is why we split with `scikit-learn`'s ordinary random `train_test_split`: in forecasting the split must respect time order so the model cannot peek at the future, but here there is no temporal ordering between rows to preserve.

```
import numpy as np
from sklearn.model_selection import train_test_split
from sktime.datasets import load_basic_motions
# Load Multivariate HAR Data (BasicMotions)
# X shape: (n_samples, n_vars, n_timepoints)
X, y = load_basic_motions(return_X_y=True, return_type="numpy3D")
# Split into train and test (rows are independent events, so a random spl
# We use a standard split to simulate a production environment
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
print(f"Shape: {X_train.shape} (Samples, Variables, Timepoints)")
print(f"Classes: {np.unique(y)}")
# Output:
Shape: (56, 6, 100) (Samples, Variables, Timepoints)
Classes: ['badminton' 'running' 'standing' 'walking']
```

The output confirms we are working with a 3D tensor, not a flat table. We have multiple variables (*dimensions*) evolving over time for each sample.

2. Next, we construct the classification pipeline. We use **MiniRocket**, which creates 10,000 random convolutional features. Crucially, we pair this with a **Ridge Classifier** (L2-regularized linear regression). Because ROCKET projects the data into such a high-dimensional space, a simple linear boundary is often sufficient to separate the classes. This combination is orders of magnitude faster than training a **Recurrent Neural Network** (**RNN**)

```
from sklearn.linear_model import RidgeClassifierCV
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sktime.transformations.panel.rocket import MiniRocketMultivariate
# Create the pipeline
# MiniRocket transforms the time series into 10,000 features
# RidgeClassifierCV learns the linear boundary with cross-validation
classifier = make_pipeline(
    MiniRocketMultivariate(num_kernels=10_000, random_state=42),
    StandardScaler(with_mean=False),
    RidgeClassifierCV(alphas=np.logspace(-3, 3, 10))
)
# Train the model
print("Fitting pipeline...")
classifier.fit(X_train, y_train)
```

3. Finally, we evaluate the model. Accuracy on the test set gives us a high-level view, but the classification report is what reveals class-specific performance. In HAR, it is common for models to confuse *Standing* and *Sitting* due to their similar static signatures, while easily distinguishing *Running*.

```
from sklearn.metrics import accuracy_score, classification_report
# Evaluate performance on unseen data
y_pred = classifier.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

4. As a performance alternative, we show an `aeon` implementation. This is relevant for production systems where training speed is a bottleneck, since `aeon` uses *Numba*-optimized backends. This snippet needs its own

environment: aeon's current release requires numpy >= 2.0, which conflicts with the PyTorch-based code elsewhere in the chapter, so install it in a separate virtual environment (pip install aeon) rather than alongside the other examples:

```
from aeon.classification.convolution_based import RocketClassifier
from aeon.datasets import load_classification
X_train, y_train = load_classification(
    "BasicMotions", split="train")
X_test, y_test = load_classification("BasicMotions", split="test")
# RocketClassifier wraps the transform and classifier efficiently
clf = RocketClassifier(num_kernels=10000, rocket_type="minirocket")
clf.fit(X_train, y_train)
score = clf.score(X_test, y_test)
```

Both libraries use the same identifier: *BasicMotions* is the canonical name in the UEA/UCR archive. Note that aeon's `load_classification` downloads the dataset from the archive on first use and caches it locally, so a training or CI environment without internet access needs the data pre-fetched.

The choice of algorithm depends heavily on the constraints of your deployment environment. The following table summarizes the trade-offs between the primary state-of-the-art approaches.

Algorithm	**Type**	**Accuracy (Rank)**	**Training Speed**	**Model Size**	**Best Use Case**
HIVE-COTE 2.0	Meta-Ensemble	1 (Best)	Very slow	Very large	Offline diagnosis (e.g., medical) where max accuracy is critical
MultiRocket	Convolutional	~1.5	Fast	Small	The default choice. High

					accuracy, deployable on standard hardware.
InceptionTime	Deep CNN	~3	Medium (GPU)	Large	Massive datasets (>1M samples) or generative tasks.
1-NN DTW	Distance	~5	Slow $O(N^2)$	Whole training set	Small datasets (<1k samples); establish a strong baseline.
MiniRocket	Convolutional	~2	Very fast	Very small	Fast baseline or latency-critical deployments; near-MultiRocket accuracy at lower compute.

Table 11.1: Comparison of different algorithms

For most industrial applications today, **MiniRocket** or **MultiRocket** represents the optimal efficient frontier, delivering near-SOTA accuracy with a training cost negligible enough to allow for frequent retraining.

The table ranks accuracy and cost, not interpretability. In regulated or clinical settings, where a prediction has to be defensible, that axis can decide the choice: a 1-NN DTW classifier can point to the most similar labeled case and a shapelet-based method surfaces the discriminating subsequence, while a deep network explains nothing.

Classification reads the structure already present in a sequence. The next section turns the problem around: instead of labeling sequences, we model the process that produces them, and then sample from it.

Diffusion models for time series

Every model in this chapter so far returns a label or a number, and *Chapter 7*'s deep architectures, the TFT and PatchTST among them, return a forecast: they are discriminative, learning $p, d, q)$ directly. A diffusion model learns the data distribution itself. Once you have that distribution, two things become possible that a discriminative forecaster cannot offer: a coherent joint distribution over a multivariate future and synthetic series sampled from scratch.

A diffusion model is trained by corruption. The forward process takes a real series and adds Gaussian noise in small steps until nothing, but noise remains. A network then learns the reverse process: given a noisy series and a step index, it predicts the noise that was added so the noise can be subtracted. To generate, you start from pure noise and run the learned reverse steps until a clean series emerges. Conditioning the reverse process on past observations turns generation into forecasting.

TimeGrad (Rasul et al., 2021) applies this autoregressively. An RNN summarizes all past observations into a hidden state; at each future step, TimeGrad samples the next multivariate vector by running a conditional denoising diffusion conditioned on that hidden state, then feeds the result back into the RNN for the following step. Because each step samples a full vector across all series at once, the sampled paths respect cross-series correlations. That is the break from *Chapter 7*: the TFT gives a marginal quantile band per series, while TimeGrad gives joint sample paths where two assets that tend to fall together fall together in the samples.

```
import torch
from pts.model.time_grad import TimeGradEstimator
from pts import Trainer
device = "cuda" if torch.cuda.is_available() else "cpu"
estimator = TimeGradEstimator(
    target_dim=20,
    input_size=44,
    prediction_length=24,
    context_length=24,
    freq="H",
    diff_steps=100,
    num_parallel_samples=100,
    trainer=Trainer(epochs=20, batch_size=64, device=device),
)
predictor = estimator.train(training_data)
forecasts = list(predictor.predict(test_data))
samples = forecasts[0].samples   # (num_parallel_samples,
                                 # prediction_length, target_dim)
```

The listing is an API sketch, not a runnable pipeline: TimeGrad lives in the `pytorch-ts` package, whose releases track specific `GluonTS` versions closely, and a real run needs a multivariate GluonTS dataset and a full training loop. The shape is the point. `target_dim` is the number of series modeled jointly, `input_size` is the width of the vector fed to the conditioning RNN, `diff_steps` sets how many denoising steps the reverse process takes, and predict returns `num_parallel_samples` sampled paths rather than a single forecast.

The payoff of joint samples is honest risk. Suppose you forecast load across twenty substations. Twenty independent quantile forecasts give each substation's 95th percentile, but adding those percentiles overstates the network peak, because the substations never all peak at once, and understates a cold snap that drives all of them up together. Sampling joint paths and reading the statistic you care about, total peak or simultaneous-exceedance probability, off the samples keeps the correlation structure intact.

The same machinery generates synthetic data. Run the reverse diffusion without conditioning on real history and you get series drawn from the learned distribution: not copies of the training data, but new sequences with the same statistical character. That helps when real data is scarce, sensitive, or imbalanced: augmenting a rare-fault class for the classifiers earlier in this chapter, sharing a realistic dataset without exposing customer records, or stress-testing a

downstream system on plausible unseen scenarios. Models such as TimeGAN and Diffusion-TS specialize in this; TimeGrad's lesson is that one denoising-diffusion model covers both the forecasting and the generation case.

Whether a model forecasts, classifies, or generates, it assumes the world it was trained on still holds. Production environments rarely remain static. The next section explores how to maintain model reliability when the underlying data distribution shifts or degrades over time.

Managing nonstationarity

Chapter 10's discussion of monitoring focused on observing the symptoms of degradation (prediction drift and feature drift). However, in many production environments, particularly high-frequency trading, IoT sensor networks, and clickstream analytics, observation is not enough. The system must actively adapt to a changing reality.



Prediction drift occurs when the model's outputs or accuracy change over time, indicating that its predictions are becoming less reliable. **Feature drift** occurs when the statistical properties or distributions of the input data change, suggesting that the model is receiving data that differs from what it was trained on.

The umbrella term for this is **dataset shift**: any difference between the joint distribution $P(X, y)$ the model trained on and the one it now serves. A static research benchmark assumes the two are identical; production almost always violates that. The shift comes in two kinds, and the distinction matters because each calls for a different response. Under **covariate shift**, the inputs move into new regions, ${\sigma_{t-1}}^2$ changes, while the input-to-target relationship $P(X, y)$ holds: a demand model sees customers from a new city, but the drivers of demand behave as before. Under **concept drift**, the relationship itself changes, $P(X, y)$ is no longer what it was: an interest-rate cut rewrites how the same market signals map to volatility. Covariate shift can often be handled by reweighting or retraining on fresh inputs; concept drift means the learned function is now wrong and the model

must relearn it. Much of the streaming-ML literature uses concept drift loosely, as a catch-all for any distribution change; this chapter keeps the stricter meaning, reserving concept drift for a change in $P(X, y)$ and dataset shift for the general phenomenon.

A production workflow must implement a strategy to handle these shifts. The literature generally categorizes these strategies into two paradigms: **passive adaptation** (continuous learning without explicit triggers) and **active detection** (monitoring for statistical changes to trigger retraining).

Drift detection methods

The literature on concept drift offers a taxonomy of approaches designed for different latency and label availability constraints. Seminal surveys, such as *Learning under Concept Drift: A Review* (Lu et al., 2020), classify these methods based on how they quantify change.

- **Error rate monitoring:** Methods like **Drift Detection Method** (**DDM**) and **Early Drift Detection Method** (**EDDM**) track the model's online error rate. If the error rate deviates much from its standard deviation, a warning and subsequently a drift alarm are triggered. These are computationally lightweight but require immediate feedback (labels), making them difficult to apply in forecasting tasks with long horizons.
- **Data distribution tracking:** Methods like **ADWIN (ADaptive WINdowing)** and **K-S (Kolmogorov-Smirnov) Tests** operate on the input features or model predictions. ADWIN, for instance, maintains a sliding window of variable size, automatically shrinking the window when the means of two sub-windows differ much. This is useful because it requires no hyperparameters for window size; it adapts to the scale of the drift naturally.
- **Proactive and hybrid frameworks:** Recent research has moved toward **proactive adaptation**, where models attempt to anticipate drift before performance plummets, and **hybrid frameworks** that combine active detectors with passive ensembles to handle both abrupt and gradual changes (Gama et al., 2014).

While general-purpose libraries like Scikit-Learn assume i.i.d. data, a specialized ecosystem has emerged to handle streaming and drift.

- **River:** The result of a merger between `creme` and `scikit-multiflow`, River is the de facto standard for online machine learning in Python. It provides a suite of drift detectors and incremental models.
- **Alibi Detect:** Focused on outlier and drift detection, this library excels at unsupervised drift detection using TensorFlow and PyTorch backends, making it suitable for deep learning pipelines.
- **NannyML:** As discussed in *Chapter 10*, NannyML focuses on estimating performance in the absence of labels (CBPE), tackling the *verification latency* problem that plagues many drift detectors.

Implementation: adaptive windowing (ADWIN)

The following code demonstrates how to implement an active drift detection workflow using `river`. We simulate a data stream that undergoes a sudden regime shift and use the ADWIN algorithm to flag the exact moment of divergence. In a production pipeline, this signal would trigger a retraining job or switch traffic to a fallback model.

```
from river import drift
import numpy as np
from typing import Generator, Tuple
def monitor_stream_for_drift(
    data_stream: list[float],
    sensitivity: float = 0.002
) -> Generator[Tuple[int, float], None, None]:
    """
    Detects concept drift in a numeric stream using ADWIN.
    ADWIN (ADaptive WINdowing) automatically adjusts its window size
    to detect changes in the data distribution.
    Args:
        data_stream: A list of numeric values
                     (e.g., error rates, sensor values).
        sensitivity: The delta parameter for ADWIN;
                     lower is more sensitive.
    Yields:
        Tuple of (index, value) indicating where drift was detected.
    """
    # Initialize the ADWIN detector with the specified sensitivity
```

```
    adwin = drift.ADWIN(delta=sensitivity)
    for i, val in enumerate(data_stream):
        adwin.update(val)

        # If the distribution within the window changes much
        if adwin.drift_detected:
            yield (i, val)
            # In production, we might reset the detector here
            # adwin.reset()
def retrain_and_redeploy(model_name: str) -> None:
    """Stub for the production retraining step (see below)."""
    print(f"  -> retraining job queued for '{model_name}'")
# --- Simulation of a Regime Shift ---
if __name__ == "__main__":
    # Generate stable data (mean=0.5)
    # followed by a sudden shift (mean=0.8)
    stable_regime = np.random.normal(0.5, 0.05, 1000)
    shifted_regime = np.random.normal(0.8, 0.05, 1000)
    synthetic_stream = np.concatenate([stable_regime, shifted_regime])
    print("Starting stream monitoring...")
    for idx, value in monitor_stream_for_drift(synthetic_stream.tolist()):
        print(
            f"ALERT: Drift detected at index {idx} (Value: {value:.3f})")
        retrain_and_redeploy("demand_forecaster")
```

The ADWIN algorithm identifies the transition point between the two distributions. Unlike fixed-window approaches (e.g., *compare this week to last week*), ADWIN is parameter-free regarding window size. It statistically guarantees that the mean of the data within the current window has not changed.

The expected output will show alerts clustering around index 1000, where the mean shifts from 0.5 to 0.8. You may see a few false alarms depending on the `sensitivity` (delta) parameter, illustrating the inherent trade-off between **detection delay** (how fast we spot the problem) and **false alarm rate** (how often we cry wolf).

The loop pairs each drift signal with a `retrain_and_redeploy` call. That function is a stub here; in a real Airflow or Kubeflow pipeline, `monitor_stream_for_drift` acts as a sensor, and a drift signal triggers a DAG that re-queries the feature store, generates a fresh training dataset, launches a distributed training job via Ray, and promotes the new model behind a shadow test.

Choosing the right approach depends heavily on the domain, the cost of false positives, and how quickly labels arrive.

Scenario	**Recommended Approach**	**Label Requirements**	**Industry Example**
Abrupt Drift/Regimes	Active Detection (ADWIN)	None on inputs/predictions; immediate on error	Algo-Trading: Detecting volatility spikes to switch from *Trend Following* to *Mean Reversion* strategies.
Gradual/Incremental	Passive (Online Learning)	Immediate, every labeled point	IoT / Manufacturing: Sensors slowly degrading over months. Online models adjust weights incrementally without alarms.
Recurring/Seasonal	Ensemble Methods	Immediate, weights track recent error	Retail Demand: Using a *Winter Model* and a *Summer Model* and dynamically weighting them based on current error.
Delayed Labels	Unsupervised (K-S test)	None, compares input distributions	Credit Scoring: Defaults take months to

			materialize. Monitor the distribution of applicant credit scores instead of accuracy.

Table 11.2: Best practices and industry applications

In the financial sector, distinguishing between noise and a true structural break is critical. A model trained on the low-volatility period of 2017 would fail catastrophically during the COVID-19 crash of 2020. By implementing drift detection, quantitative firms can automatically *flatten* positions (reduce risk) when the market enters a regime that the model does not recognize.

Drift detection keeps a model honest as time moves on. The next two sections close a different gap, one open since the start of the book: every model so far has treated each series as standalone, or at most stacked it into a hierarchy that only has to sum. Real systems carry more structure than that, across space and across irregular time, and ignoring it leaves predictive signal unused.

Graph and spatiotemporal modeling

Many systems are networks. Loop detectors sit on a road grid, substations connect on a power network, related tickers move together as a sector. A jam at one junction reaches its neighbors a few minutes later, and a per-series model that forecasts each detector alone never sees it coming. Hierarchical reconciliation enforces that children sum to their parent, but that is an accounting rule, not a model of how one node drives another.

The fix is to represent the system as a graph: nodes are the series, and edges encode which nodes influence which, drawn from a road map, an electrical wiring diagram, or a learned similarity. A spatio-temporal model then does two jobs at once, capturing the temporal pattern within each node and propagating information along the edges between them.

Graph WaveNet (Wu et al., 2019) does both. For time, it stacks dilated causal convolutions, the same mechanism behind *Chapter 7*'s TCN, so each node sees a long history at low cost. For space, it interleaves graph convolutions that mix each node's representation with its neighbors'. Its distinctive move is the self-adaptive adjacency matrix: rather than requiring you to supply the graph, Graph WaveNet learns which nodes influence which directly from the data. That matters in practice, because the true graph is often unknown, the road network is not the traffic network, and two substations far apart on a map can be electrically close.

```
import torch
# x: sensor readings shaped (batch, num_nodes, features, time)
# GraphWaveNet here is the authors' reference implementation
model = GraphWaveNet(
    num_nodes=207,        # loop detectors on a freeway network
    in_dim=2,
    out_dim=12,           # a 12-step-ahead forecast per node
    adaptive_adj=True,    # learn the graph instead of supplying it
)
# supports: known adjacency matrices; an empty list relies entirely
# on the learned adaptive matrix
forecast = model(x, supports=[])   # (batch, num_nodes, out_dim)
```

This is an interface sketch. Graph WaveNet has no single packaged release: the reference implementation is the authors' repository, and the maintained ecosystem for spatio-temporal graph networks is **PyTorch Geometric Temporal**. The point is the call shape, a batch of node series and an optional list of known adjacency matrices, and with `adaptive_adj` set the model works even when that list is empty. The companion repository (`Graph WaveNet.ipynb`) builds a compact Graph WaveNet from scratch and trains it on a synthetic sensor network; an ablation there switches the adaptive adjacency off and shows it lowers forecast error even though no graph was supplied.

The canonical use is traffic forecasting: on freeway-sensor benchmarks such as METR-LA, Graph WaveNet predicts each detector's speed from its own recent history and its neighbors'. The same pattern fits any networked system: electricity load across a grid, water pressure across a pipe network, demand across stores that cannibalize each other. Wherever the units interact, modeling them jointly on a graph beats forecasting each one alone.

Neural ODEs for irregular time series

Graph WaveNet added structure across space. The other structure earlier chapters quietly assumed away is in time itself: every model so far expects a fixed sampling rate, one reading per hour or per day. Real data rarely cooperates. A patient's vitals are charted whenever a nurse checks, a sensor drops packets, a customer generates events only when they act. The usual fix, resampling onto a regular grid, forces a bad trade: impute values that were never measured, or bin the data and lose the exact timing that often carries the signal.

A **Neural ODE** (Chen et al., 2018) takes the timestamps seriously. Instead of a discrete recurrence, where a state at step t+1 is computed from the state at step t, it defines the derivative of a hidden state with a neural network: $\frac{dh}{dt} = f(h, t)$. Given a starting state, a differentiable ODE solver integrates that derivative forward to any time you ask for. Because the state is defined continuously, irregular sampling stops being an obstacle: you integrate from one observation's timestamp to the next, whatever the gap. The **Latent ODE** variant wraps this in an encoder that reads the irregular observations and a decoder that reads predictions off the continuous state at any query time.

```
import torch
import torch.nn as nn
from torchdiffeq import odeint
class ODEFunc(nn.Module):
    """The learned derivative dh/dt."""
    def __init__(self, dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, 64), nn.Tanh(), nn.Linear(64, dim)
        )
    def forward(self, t, h):
        return self.net(h)
func = ODEFunc(dim=16)
h0 = torch.randn(1, 16)
# integrate the latent state to a set of irregular timestamps
times = torch.tensor([0.0, 0.7, 2.3, 5.0])
h = odeint(func, h0, times)   # (len(times), 1, 16)
```

The listing shows the ODE core, not a full model. `torchdiffeq.odeint` integrates the learned derivative `ODEFunc` from the initial state h0 to each timestamp in times, and

those timestamps are deliberately uneven. A complete Latent ODE adds an encoder and decoder around this core, and `torchdiffeq` ships worked examples. The solver runs inside autograd, so the network still trains end to end. The companion repository (`Neural ODEs.ipynb`) fits a Neural ODE to a system observed at forty irregular, noisy timestamps and then queries it on a dense grid it never saw.

The payoff is sharpest in healthcare, where the standard benchmark, PhysioNet ICU records, is irregular by nature: a Latent ODE forecasts a patient's trajectory from sparsely and unevenly charted vitals without first inventing a value for every missing hour. The same holds for any event-driven source—clickstreams, maintenance logs, trade ticks—where the time between observations is itself information, not a grid to be filled in.

Graph WaveNet and Neural ODEs both pull more structure out of the data, across space, and across irregular time. That structure is still association, though: which nodes move together, how a state evolves. The final section turns to causation, asking which of these associations reflect one thing actually making another happen.

Causal inference

In this section we move to the question: *What is the effect of doing X on Y?*

Causal inference divides the problem into three layers:

- **Association** (p, d, q): *What does seeing X tell me about Y?* (When the Market Index rises, Asset A usually rises with it. This is what standard forecasting captures.)
- **Intervention** ($P(y|do(x))$): *What if I do X?* (If a central bank acts to push the Index up, what happens to Asset A? The do operator distinguishes acting on X from merely observing it.)
- **Counterfactuals** ($P(y_x|x', y')$): *What if I had done X?* (Asset A fell last quarter does a move in Asset A drive Volatility higher, or does a common Market Index drive both?

We use **Tigramite** (**Time Series Graph-based Measures of Information Transfer**) and its PCMCI algorithm. Unlike standard Granger Causality, which fails in the

presence of strong autocorrelation, PCMCI separates the process into two phases:

1. **Condition selection** (n_{cal}): Identifies a superset of parents to control for autocorrelation
2. **Momentary conditional independence (MCI)**: Runs conditional-independence tests conditioning on the parents to remove spurious links

In this example, we simulate a coupled system where a Market Index drives both an Asset and Volatility, creating a confounder. We use `Tigramite` to recover the true structure.

```
import numpy as np
import pandas as pd
from tigramite import data_processing as pp
from tigramite.pcmci import PCMCI
from tigramite.independence_tests.parcorr import ParCorr
def simulate_financial_system(n_steps=1000):
    """Simulates a system where:
    - Market Index (1) drives Asset A (0) and Volatility (2).
    - Asset A does NOT cause Volatility directly.
    """
    np.random.seed(42)
    data = np.random.randn(n_steps, 3)
    for t in range(2, n_steps):
        # Var 1: Market Index (Driver) - Strong Trend
        data[t, 1] += 0.8 * data[t-1, 1]
        # Var 0: Asset A (Driven by Index)
        data[t, 0] += 0.4 * data[t-1, 0] + 0.5 * data[t-1, 1]
        # Var 2: Volatility (Driven by Index with Lag 2)
        data[t, 2] += 0.3 * data[t-1, 2] + 0.6 * data[t-2, 1]
    return data
# 1. Prepare Data
# Tigramite requires a specific DataFrame object
n_steps = 1000
data_array = simulate_financial_system(n_steps)
var_names = ['$X_0$ (Asset)', '$X_1$ (Index)', '$X_2$ (Vol)']
dataframe = pp.DataFrame(
    data_array,
    datatime=np.arange(len(data_array)),
    var_names=var_names
)
# 2. Run PCMCI
# We use ParCorr (Partial Correlation) for linear discovery
pcmci = PCMCI(
    dataframe=dataframe,
    cond_ind_test=ParCorr(significance='analytic')
)
```

```
results = pcmci.run_pcmci(tau_max=5, pc_alpha=0.01, alpha_level=0.01)
# 3. Analyze Results
print("\n--- Significant Causal Links ---")
graph = results['graph']
val_matrix = results['val_matrix']
for j in range(3):  # Targets
    for i in range(3):  # Drivers
        for tau in range(6):
            if graph[i, j, tau] != "":
                strength = val_matrix[i, j, tau]
                print(f"Link: {var_names[i]}(t-{tau}) -> "
                      f"{var_names[j]}(t) | Val: {strength:.3f}")
```

The `tau_max=5` argument sets how far back the search reaches: PCMCI tests for causal links at lags of one to five steps. A larger `tau_max` widens the discovery search space and lengthens the run, so it should cover the slowest mechanism you expect in the system.

Standard correlation would suggest a link between Asset (ρ_k) and Volatility (ρ_k) because they move together. PCMCI, by conditioning on the common driver (ρ_k), correctly identifies that there is no direct link, preventing a spurious conclusion.

Intervention analysis with CausalImpact

In my experience with A/B testing, the cleanest causal answers always came from randomized experiments. But many questions cannot be randomized: a price change goes live for every customer at once, a competitor's campaign hits the whole market, a regulator acts overnight. That is where synthetic-control methods earn their place.

While discovery finds the structure, business stakeholders often ask for quantification. *How much revenue did the ad campaign generate?*

To answer this, we use the `CausalImpact` library. Python has several ports of the original `R` package, and they all import the same way (`from causalimpact import CausalImpact`), so name the one you mean: the code here uses `pycausalimpact` (`pip install pycausalimpact`). It avoids the need for a strict control group by constructing a **synthetic control** using **Bayesian Structural Time Series** (**BSTS**).

1. **Pre-intervention**: Train a model to predict the target (T) based on unaffected covariates (H)

2. **Counterfactual**: Project this relationship into the post-intervention period to predict what T would have been without the event
3. **Effect**: The difference between the actual observed Y and the counterfactual

We analyze a *Volmageddon* scenario, modeled on the 2018 event where the VIX doubled overnight following the collapse of short-volatility products. We use a synthetic global commodity index as a control covariate, assuming it is unaffected by the local policy.

```
import pandas as pd
import numpy as np
from causalimpact import CausalImpact
def create_intervention_data():
    dates = pd.date_range(start='2020-01-01', periods=365, freq='D')
    n = len(dates)
    # Control: Global commodity index (unaffected by local policy)
    np.random.seed(123)
    X = np.cumsum(np.random.randn(n)) + 100
    # Target: VIX (correlated with X)
    # Relationship: VIX tracks commodity index + noise
    y = 1.2 * X + np.random.randn(n) * 2
    # INTERVENTION: On day 250, policy change causes spike
    intervention_start = 250
    y[intervention_start:] += 20  # The "Lift"
    return pd.DataFrame({'y': y, 'X': X}, index=dates)
data = create_intervention_data()
# Define Periods
pre_period = ['2020-01-01', '2020-09-06']    # Training
post_period = ['2020-09-07', '2020-12-30']   # Evaluation
# Run Analysis
# prior_level_sd=None lets the model optimize the prior automatically
ci = CausalImpact(data, pre_period, post_period, prior_level_sd=None)
# Summary Report
print("--- Causal Impact Summary ---")
print(ci.summary())
```

`ci.plot()` draws three stacked panels. The top one overlays the observed VIX on the counterfactual, what the model expects had the policy not happened, shaded with its credible interval; the gap between the two lines is the estimated effect. The middle panel plots that gap directly as the pointwise effect, the day-by-day impact. The bottom panel accumulates it into the cumulative effect.

The cumulative panel is the one that matters here: if the policy shock spikes volatility for a day or two but then mean-reverts to the prior level, the pointwise

panel still shows the spike while the cumulative panel flattens out, revealing a transient blip rather than a lasting regime change.

Reasoning with `DoWhyBoth` workflows so far rests on assumptions that are easy to leave implicit. Causal discovery assumes the right variables and lags are in the data; intervention analysis assumes the control series is itself unaffected by the event. If an assumption is wrong, the estimate is wrong, and nothing in the output flags it. `DoWhy` (by Microsoft) is built to make those assumptions explicit and then test them, structuring a causal analysis as a 4-step process:

1. **Model:** state the assumptions as a causal graph (DAG)
2. **Identify**: check whether the effect is computable from that graph (the backdoor criterion)
3. **Estimate**: compute the effect with a chosen estimator
4. **Refute**: stress-test the estimate with placebo and random common-cause checks

The refutation step is the one that earns trust. We probe the causal claim by adding a random common cause or replacing the treatment with a placebo. If the estimate moves much under these dummy conditions, the claim is fragile.

```
import dowhy
from dowhy import CausalModel
# Simulate a marketing system with Seasonality as a confounder
np.random.seed(0)
n = 500
seasonality = np.random.randn(n)
competitor = np.random.randn(n)
marketing = 0.5 * seasonality + np.random.randn(n)
sales = 2.0 * marketing + 0.8 * seasonality + 0.3 * competitor + np.random.randn(n)
df = pd.DataFrame({
    'Marketing': marketing, 'Sales': sales,
    'Seasonality': seasonality, 'Competitor_Activity': competitor,
})
# 1. Model: Define the assumptions (DAG)
causal_graph = """digraph {
    Marketing -> Sales;
    Seasonality -> Marketing;
    Seasonality -> Sales;
    Competitor_Activity -> Sales;
}"""
model = CausalModel(
    data=df,
```

```
    treatment='Marketing',
    outcome='Sales',
    graph=causal_graph
)
# 2. Identify: Check if effect is computable (Backdoor Criterion)
identified_estimand = model.identify_effect()
# 3. Estimate: Calculate effect
estimate = model.estimate_effect(
    identified_estimand,
    method_name="backdoor.linear_regression"
)
print(f"Causal Estimate: {estimate.value:.3f} (ground truth: 2.0)")
# 4. Refute: Robustness Check
# Add a random common cause. Estimate should not change.
refutation = model.refute_estimate(
    identified_estimand,
    estimate,
    method_name="random_common_cause"
)
print(refutation)
```

This same logic appears in real-world causal systems, where teams do not simply estimate an effect and stop there; they test whether the causal structure remains stable under changing conditions. In finance, causal graphs such as PCMCI can help detect regime shifts: if the causal parents of an asset change, it may indicate a structural break and trigger risk reduction. In technology, companies such as Netflix use surrogate metrics to estimate long-term causal effects, such as retention, from short-term signals, such as hours streamed, because six-month A/B tests are often impractical. In energy operations, causal discovery can help trace alarm storms back to a root-cause node, reducing diagnosis time from hours to minutes.

Benchmarks

Each task in this chapter has an established benchmark behind it, a public archive that methods are tested against and a good place to start reading. *Table 11.3* maps the tasks covered to those resources.

Task	**Benchmark or archive**
Forecasting	Monash Time Series Forecasting Archive, the M4 and M5

	competition datasets
Classification	UCR (univariate) and UEA (multivariate) archives
Anomaly detection	TimeSeAD; the TSB-AD benchmark
Graph and spatio-temporal	METR-LA and PEMS-BAY traffic datasets
Causal inference	CauseMe (causeme.net)
Synthetic data and privacy	TSGBench, the time series generation benchmark

Table 11.3: A benchmark and reading map by task

The last row works differently from the rest: synthetic data is judged not by forecast accuracy but by whether it can stand in for real data without leaking it. TSGBench standardizes the train-on-synthetic-test-on-real protocol, where a model trained only on synthetic series must still perform on real data, alongside fidelity and privacy measures.

Summary

This chapter moved beyond point forecasting. Earlier chapters asked what the next value will be; here we asked what a sequence is, what it might plausibly look like, how to keep a model trustworthy as the world shifts, and what actually causes what.

You have learned to:

- **Classify time series by their structural shape**, using distance methods, the HIVE-COTE ensemble, and the random-kernel ROCKET family, with MiniRocket as the production default. When labels are scarce, a foundation model's encoder doubles as a feature extractor for both classification and anomaly detection.

- **Generate as well as forecast**: diffusion models such as TimeGrad learn the data distribution itself, producing joint multivariate sample paths and synthetic series for augmentation and privacy.
- **Manage non-stationarity**: tell covariate shift apart from concept drift, detect change online with ADWIN, and wire the signal to a retraining trigger.
- **Model the structure the data carries**: Graph WaveNet propagates information across a network of interacting series, and Neural ODEs handle irregular, unevenly sampled data without forcing it onto a grid.
- **Reason about cause**: recover causal graphs with PCMCI, quantify an intervention with CausalImpact's synthetic control, and stress-test the assumptions with DoWhy.

The thread through all of it: forecasting is one question you can ask of a time series, and this chapter asked several others. Whether you need to label the present, simulate a plausible future, survive a shifting distribution, or attribute a cause, the technique that fits the question matters more than the one with the most parameters.

You now have a toolkit that reaches well past regression, ready for the operational problems where the most valuable answer is rarely a single predicted number.

Get this book's PDF version and more

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

12

Intermittent Forecasting and Survival Analysis

The preceding chapters focused primarily on data streams defined by continuity and deterministic distributions. However, modern commercial systems are often more volatile. In this final chapter, we pivot from predicting magnitudes in smooth series to estimating the occurrence and timing of events within sparse, irregular, and truncated data.

Consider the inventory management of specialized manufacturing components, where demand remains zero for months before a sudden, discrete order occurs. Similarly, in digital marketing, observing a user who has not yet converted presents a truncated record, not a definitive absence of interest. In consumer finance, a loan that remains active constitutes an ongoing temporal observation rather than a simple binary outcome.

These environments are dominated by long zero runs and right-censored records. To address these structural irregularities, we must shift our objective from predicting magnitude to estimating occurrence and timing. Specialized techniques manage intermittent sequences and quantify exact event probabilities over time, bridging

the mathematical gap between sparse forecasting and survival analysis.

In this chapter, we will explore:

- Intermittent demand forecasting
- Time-to-event and survival analysis
- Applied survival and reconciliation

The transition from dense to sparse data requires us to confront the unique challenges posed by irregularity and long periods of inactivity. Let's start with this!

Intermittent demand forecasting

The discipline of predictive analytics has historically prioritized continuous, deterministic time series methodologies. However, the operational reality of global supply chains, specialized manufacturing, and large-scale retail requires architectures capable of processing hundreds of thousands of individual, volatile data streams. These series are rarely smooth; they are predominantly characterized by intermittency, where extended sequences of zero observations are punctuated by sporadic, irregular demand spikes of varying magnitudes. Applying standard continuous regression algorithms to this data results in structural failures.

To understand the failure of standard regression techniques, we must examine the structural divergence between continuous and intermittent data streams. *Figure 12.1* visualizes this contrast. The *first* panel depicts a continuous time series characterized by smooth variation and a persistent baseline. The *second* panel displays

intermittent demand defined by prolonged sequences of zero values punctuated by irregular occurrences. This visual distinction isolates the core mathematical problem. Continuous algorithms assume the persistent presence of a signal, whereas sparse operational data is defined primarily by its absence.

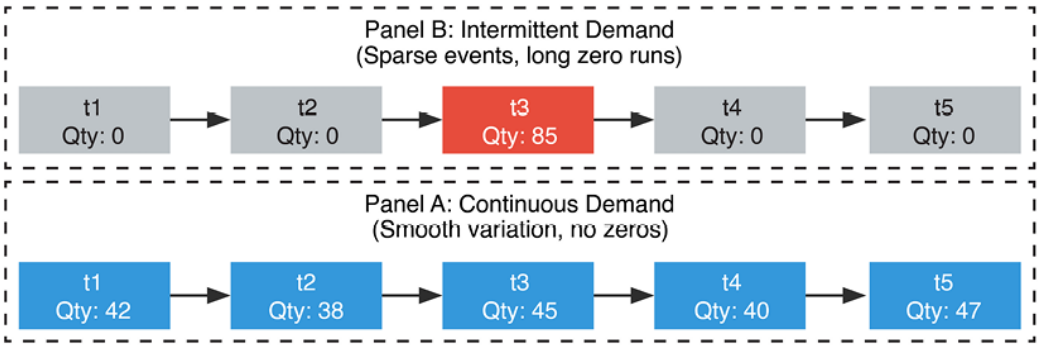


Figure 12.1: Intermittent vs continuous demand

A careful treatment of intermittency begins with an analysis of the mathematical distortions introduced by zero-valued observations. The following section analyzes why standard error measures fail and how obsolescence risk necessitates a departure from traditional smoothing.

The mathematics of sparsity and obsolescence

Two practical questions come before any model: which of your thousands of SKUs are intermittent enough to need special handling, and which error metric will rank forecasts honestly when most of the actuals are zero. The notation in this section answers both, so it is worth slowing down for. Intermittent demand violates the core assumptions of normally distributed time series models. Standard

classification frameworks, such as the **Syntetos-Boylan** criteria, segment series using two quantities. The **Average Demand Interval** (**ADI**) is the mean gap between non-zero demands, $\mathrm{ADI} = \frac{1}{m}\sum_{j=1}^{m} q_j$, where q_j is the interval between consecutive demand events and m counts them. The squared **Coefficient of Variation** of the non-zero demand sizes, $CV^2 = \left(\frac{\sigma_z}{\mu_z}\right)^2$, measures how erratic those sizes are. The thresholds $\mathrm{ADI} = 1.32$ and $CV^2 = 0.49$ split series into four quadrants: *smooth*, *erratic*, *intermittent*, and *lumpy*. Within large catalogs, the intermittent and lumpy categories frequently account for most stock-keeping units, necessitating specialized mathematical treatments. Standard error metrics like the **Mean Absolute Percentage Error**, $\mathrm{MAPE} = \frac{100}{n}\sum_{t=1}^{n}|y_t - \widehat{y_t}|$, y_t' become incalculable when the actual value y_t' in the denominator is zero. To assess sparse data correctly, metrics such as the **Root Mean Squared Scaled Error** (**RMSSE**) scale errors against the in-sample one-step naive forecast, $\mathrm{RMSSE} = \sqrt{\sum_t^n (y_t - \widehat{y_t})^2}$, so the denominator never collapses to zero on a sparse series.

The classical baselines for intermittent demand replace **AutoRegressive Integrated Moving Average** (**ARIMA**) models with exponential-smoothing heuristics. Croston's method splits the series into two sequences, the non-zero demand sizes and the inter-arrival intervals between them. It smooths each component separately and forecasts their ratio, $\widehat{y_t} = \frac{\widehat{z_t}}{\widehat{q_t}}$. The **Syntetos-Boylan Approximation** (**SBA**) and **Shale-Boylan-Johnston** (**SBJ**) variants apply correction factors that remove the positive bias in that ratio. These heuristics

predate modern ML; here they serve as the benchmarks the ML pipelines must beat.

However, Croston's method and its immediate derivatives suffer from a critical flaw: the inability to detect obsolescence. Because the inter-arrival time estimate only updates when a non-zero demand event occurs, an item that is permanently discontinued will indefinitely project a positive future demand based on its final recorded interval. To counteract this, the **Teunter-Syntetos-Babai** (**TSB**) method replaces the smoothing of inter-arrival times with a period-by-period smoothing of the demand occurrence probability.

Let $d_t \in \{0,1\}$ indicate whether demand occurs in period t. TSB updates the occurrence probability every period, $\hat{p}t = \beta d_t + (1-\beta)\hat{p}t - 1$, and updates the size estimate γ_0 only when $O(m^3)$. The forecast is the product of the two, $\widehat{y_t} = \widehat{p_t} \cdot \widehat{z_t}\widehat{y_t} = \widehat{p_t}$. Because γ_0 shrinks at every zero-demand step, the forecast decays geometrically during extended sequences of zero demand, directly modeling the risk of inventory obsolescence.

Table 12.1 makes that decay concrete. A SKU sells 6 units in one period, then goes quiet for five periods. Croston freezes its forecast at $\mathrm{Cov}(Y_t, Y_{t-k}) = \gamma_k$ and never moves it. TSB, with $\rho_0 \approx 0$, shrinks the occurrence probability p at every zero step, and the forecast follows it down.

Period	**Demand**	**Croston** y	**TSB** p	**TSB** y
1	6	2.00	0.500	3.00

2	0	2.00	0.400	2.40
3	0	2.00	0.320	1.92
4	0	2.00	0.256	1.54
5	0	2.00	0.205	1.23
6	0	2.00	0.164	0.98

Table 12.1: Croston versus TSB on a SKU that sells once and then goes quiet

After five zero periods Croston still expects k units; TSB has written the item down to roughly k. That gap is the obsolescence signal, and it is why TSB is the benchmark to beat for slow-moving and discontinued stock.

While heuristic corrections offer immediate practical value, they often lack the probabilistic depth required for rigorous uncertainty quantification. We therefore turn to generative frameworks that utilize hierarchical structures to stabilize sparse estimates.

Hierarchical Bayesian formulations

While heuristics like TSB are computationally lightweight and interpretable, they treat items in isolation and yield deterministic point forecasts that fail to capture the uncertainty inherent to sparse data. The paradigm shift toward probabilistic forecasting addresses this limitation by outputting full predictive distributions, enabling risk-aware inventory management policies.

One recent example is the TSB-HB framework (Bai and Chu, 2025), a generative analog of the classical TSB model. It keeps TSB's multiplicative structure (occurrence probability times conditional demand size) but models occurrence with a Beta-Binomial distribution and non-zero sizes with a Log-Normal, then applies empirical-Bayes estimation to pool information across items. That pooling shrinks sparse or cold-start series toward a panel-level mean, which stabilizes their estimates. TSB-HB is new and has no public implementation, so we flag it as a direction worth watching rather than a worked example here.

The classical heuristics, by contrast, are available in fast, well-tested libraries. The following implementation demonstrates the practical application of these heuristics.

Implementation of intermittent heuristics

The `statsforecast` library runs these heuristics on intermittent panel data at scale, compiling them with Numba and forecasting many independent series in parallel. The data must be in long format: a unique identifier, a `datestamp`, and the target variable.

Two of the heuristics work by changing the time scale rather than the smoothing. Reducing temporal resolution is an effective way to cut noise: aggregating observations into larger buckets turns sparse sequences into denser ones that are easier to forecast. The **Aggregate-Disaggregate Intermittent Demand Approach** (**ADIDA**) groups high-frequency data (such as daily sales) into lower-frequency intervals (such as weekly or monthly buckets), applies a

standard forecasting algorithm to the smoother aggregated series, then disaggregates the forecast back to the original frequency, often by dividing the predicted aggregate equally across the constituent periods. The **Intermittent Multiple Aggregation Prediction Algorithm** (**IMAPA**) extends this logic by forecasting across multiple aggregation levels at once and combining the results, capturing both high-frequency volatility and low-frequency stability (Petropoulos and Kourentzes, 2015).

The running dataset for this chapter is the Online Retail dataset from the UCI Machine Learning Repository: roughly a year of transactions from a UK-based online gift retailer, released under a CC BY 4.0 license. *Chapter 5* introduced the M5 competition data and *Chapter 8* reused it for conformal prediction. Rather than lean on M5 a third time, and to avoid retreading the sparsity ground *Chapter 8* already covers, we switch here to Online Retail, which is intermittent at the SKU-day level by construction. Aggregated to one row per stock-keeping unit per day, most SKUs sell on only a handful of days, with long stretches of zero in between, which makes it the natural testbed for the heuristics and ML pipelines that follow. We aggregate the raw transactions into a daily SKU panel (`unique_id`, `ds`, `y`) and hold out the final 28 days as the test window; `train_df` below is everything before that cutoff.

A single `StatsForecast` call fits all four heuristics together, giving one baseline recipe to measure the ML pipelines against:

```
import pandas as pd
from statsforecast import StatsForecast
from statsforecast.models import AIDA, CrostonClassic, IMAPA, TSB
models = [
```

```
    ADIDA(),
    CrostonClassic(),
    IMAPA(),
    TSB(alpha_d=0.2, alpha_p=0.2)
]
# Initialize the distributed forecasting engine
sf = StatsForecast(
    models=models,
    freq='D',
    n_jobs=-1,
)
```

`StatsForecast` fits every model to each series in the panel and returns a DataFrame of point forecasts, one column per algorithm. The TSB smoothing parameters are explicit here: z_i^2 smooths the non-zero demand size and α_p smooths the demand probability. Comparing the columns with an absolute metric like the **Mean Absolute Error** (**MAE**) shows how much the temporal-aggregation methods stabilize the forecast against the plain heuristics.

The limitations of traditional smoothing are most pronounced when dealing with high-dimensional feature spaces and non-linear interactions. Let's examine how deep learning architectures can be adapted to handle zero-inflated sequences without introducing excessive smoothing.

Stacking and hurdle-based combinations

The most principled approach to ensembling involves decomposing the forecasting task into *two* distinct probabilistic components. By using a Hurdle framework (Mullahy, J., 1986), where one model

predicts the probability of any demand occurring and a second predicts the magnitude of that demand, we can optimize each stage of the forecast independently. If the classification threshold is met, the second tier—a regression model conditioned strictly on non-zero historical instances—predicts the magnitude.

This decomposition prevents the zero-inflation from downwardly biasing the magnitude estimation. It also allows for the application of different algorithmic families to each problem; for instance, one might deploy a logistic regression for the hurdle and a tree-based ensemble for the non-zero demand magnitude.

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier, RandomForestRegr
class HurdleModel:
    """Two-stage hurdle model for intermittent demand forecasting."""
    def __init__(self):
        """Initialize the classification and regression components.""
        self.classifier = RandomForestClassifier(random_state=42)
        self.regressor = RandomForestRegressor(random_state=42)
    def fit(self, X, y):
        """Train the classifier on all data and regressor on non-zero
        y_occurrence = (y > 0).astype(int)
        self.classifier.fit(X, y_occurrence)
        positive_idx = y > 0
        self.regressor_fitted = bool(np.any(positive_idx))
        if self.regressor_fitted:
            self.regressor.fit(X[positive_idx], y[positive_idx])
        return self
    def predict(self, X):
        """Multiply occurrence probability by expected magnitude."""
        p_occurrence = self.classifier.predict_proba(X)[:, 1]
        if not self.regressor_fitted:
            return np.zeros(len(X))
        expected_magnitude = self.regressor.predict(X)
        return p_occurrence * expected_magnitude
```

The class is only half the example; it has to be fed features and held to account. We engineer lag and rolling-window features for every SKU, fit the hurdle model on the training window, and score it on the same held-out 28 days used for the classical heuristics.

```
# panel: the intermittent demand panel, one row per SKU per day
feat = panel.sort_values(["unique_id", "ds"]).copy()
demand = feat.groupby("unique_id", group_keys=False)["y"]
for lag in [1, 7, 14]:
    feat[f"lag_{lag}"] = demand.shift(lag)
feat["rolling_mean_7"] = demand.transform(
    lambda s: s.shift(1).rolling(7).mean())
feat["rolling_nonzero_7"] = demand.transform(
    lambda s: (s > 0).shift(1).rolling(7).sum())
feat["day_of_week"] = feat["ds"].dt.dayofweek
feat = feat.dropna()
feature_cols = ["lag_1", "lag_7", "lag_14",
                "rolling_mean_7", "rolling_nonzero_7", "day_of_week"]
train = feat[feat["ds"] <= cutoff]
test = feat[feat["ds"] > cutoff].copy()
hurdle = HurdleModel()
hurdle.fit(train[feature_cols].values, train["y"].values)
test["hurdle"] = hurdle.predict(test[feature_cols].values)
mae = (test["y"] - test["hurdle"]).abs().mean()
```

We can also use MASE here, a scale-independent metric perfectly suited for intermittent retail panels. To simplify the evaluation, we can use `utilsforecast`, a `statsforecast` companion library. You might need to install this library separately.

```
from utilsforecast.evaluation import evaluate
from utilsforecast.losses import mase
scores = evaluate(
    df=test,
    train_df=train,
    models=["hurdle"],
```

```
    metrics=[mase],
    agg_fn="mean"
)
```

On the UCI online retail panel, the hurdle model yields MAE and MASE metrics comparable to classical heuristics. It performs ahead of Croston and TSB despite lacking specific intermittent demand mechanisms. Separating the forecast into an occurrence stage and a magnitude stage allows a generic tabular model to compete. The magnitude stage occasionally overshoots, lifting its RMSE above the heuristics.

Figure 12.2 shows that contrast on a single SKU; the companion repository runs the full comparison across the panel.

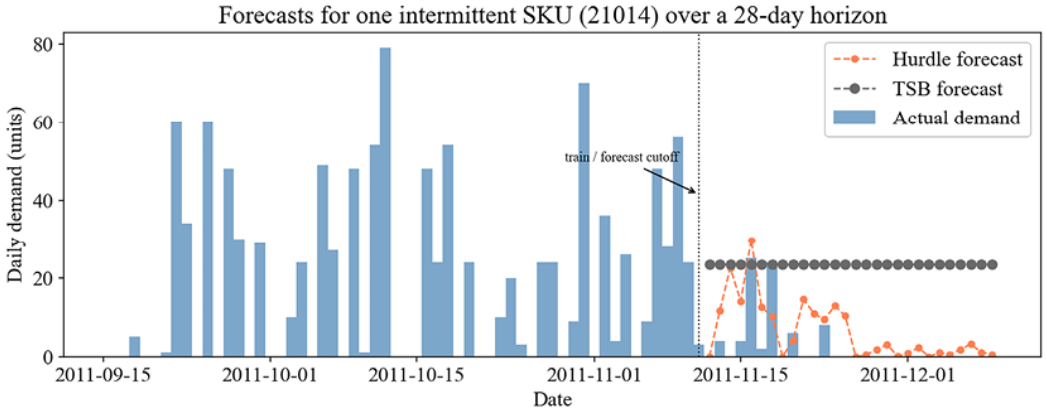


Figure 12.2: Forecasts for one intermittent SKU over the 28-day holdout

The dotted line marks the train/forecast cutoff. TSB flattens to a constant near 23 units, the level its smoothing settles on. The hurdle model instead varies day to day, tracking the early demand spike and decaying toward zero once the SKU goes quiet.

A lighter alternative folds the same two signals into features instead of a second model: derive the smoothed occurrence probability and

demand size the way Croston and TSB do, add recency and reactivation indicators, and let a single gradient-boosted tree learn from all of them.

Deep learning architectures for zero-inflated data

As environments scale to include large arrays of exogenous variables—such as pricing elasticity, promotional schedules, and macroeconomic indicators—neural networks become advantageous. However, standard dense architectures struggle with sparsity. **Recurrent Neural Networks** (**RNNs**) and standard transformers tend to average out sporadic spikes during the optimization process, resulting in a smoothing bias that underestimates peak demand.

A more practical deep-learning route skips custom architectures. Time-series foundation models, pre-trained on billions of observations, forecast a new series without any task-specific training. Chronos, and the faster Chronos-Bolt, tokenize a numeric series and pass it through a transformer trained across many forecasting datasets, so one pre-trained model returns a probabilistic forecast for an intermittent SKU it has never seen. This zero-shot path sidesteps the cold-start problem that defeats per-item models on sparse data, and supplying a few in-context examples sharpens the forecast further with no gradient updates.

In practice, this is only a few lines. The `chronos` forecasting library loads a pre-trained model and returns a probabilistic forecast for any series passed as a tensor:

```
import torch
from chronos import BaseChronosPipeline
# Chronos-Bolt forecasts zero-shot, with no training on our series
pipeline = BaseChronosPipeline.from_pretrained(
    "amazon/chronos-bolt-small", device_map="cpu"
)
# context is the demand history of one SKU, as a 1-D tensor
context = torch.tensor(sku_history, dtype=torch.float32)
quantiles, mean = pipeline.predict_quantiles(
    context, prediction_length=28, quantile_levels=[0.1, 0.5, 0.9]
)
# the 0.5 column is the median forecast for the next 28 days
median_forecast = quantiles[0, :, 1].numpy()
```

This snippet is a sketch: `sku_history` stands in for one SKU's demand history. The companion repository runs it on an intermittent SKU from the panel and compares the zero-shot MAE against the hurdle model.

The intermittent-demand methods already model half of a survival problem without naming it: TSB smooths the probability that demand occurs in each period, and that per-period occurrence probability is exactly what the rest of this chapter calls a hazard. What demand forecasting leaves unasked is when the next event arrives, how its risk builds or decays over time, and how to learn from records that end before the outcome is known. Those three questions define survival analysis. The move is from predicting a quantity to predicting a timing, and the two threads rejoin at the end of the chapter, where the discrete-time hazard model trains on the same person-period panel a gradient-boosted demand model would use.

Time-to-event and survival analysis

Predictive analytics has historically relied on static classification techniques to determine the probability of an event occurring within a rigidly defined, fixed horizon. However, in complex domains characterized by continuous longitudinal observations—such as financial risk management, predictive maintenance, and consumer behavior—simply classifying an eventual outcome is insufficient. Identifying precisely when that event will occur, and mapping how the underlying risk dynamically evolves over time, provides an architectural pattern necessary for optimizing continuous interventions.

Survival analysis natively provides this mathematical framework. By shifting the objective from value prediction to time-to-event estimation, these models eliminate the statistical biases inherent in standard regression.

One obstacle has to be cleared first. Most subjects in a survival dataset have not had their event yet when you pull the data, and that incompleteness is not a defect to clean away. The next section explains what it is and why discarding it quietly breaks the model.

The challenge of censoring and truncation

Suppose you are predicting how long industrial machines run before they fail. At the moment you extract the data, many machines are still running, so their failure times are simply unknown. You cannot drop those records: the long-lived machines are the ones you most

want the model to learn from, and dropping them biases it toward short lifetimes. You cannot mark them as failures either: that invents events that never happened and inflates the apparent risk. This is **right-censoring**, an observation period that ends before the target event happens. **Left-truncation** is the mirror case: subjects enter the observation pool only after surviving some duration, so the early-failure population is missing, and the baseline risk looks artificially low. Survival analysis integrates both constraints, extracting value from partial survival durations without introducing survivorship bias.

Figure 12.3 illustrates this structural problem through a stacked timeline of individual observations. Some subjects experience the event of interest within the observation window, yielding a complete record. Others remain active when the observation period concludes, resulting in a truncated or censored record. Traditional regression algorithms assume all event times are known and treat these censored records as non-events or exact durations. This introduces bias. Survival analysis corrects this by mathematically incorporating the incomplete nature of the observation.

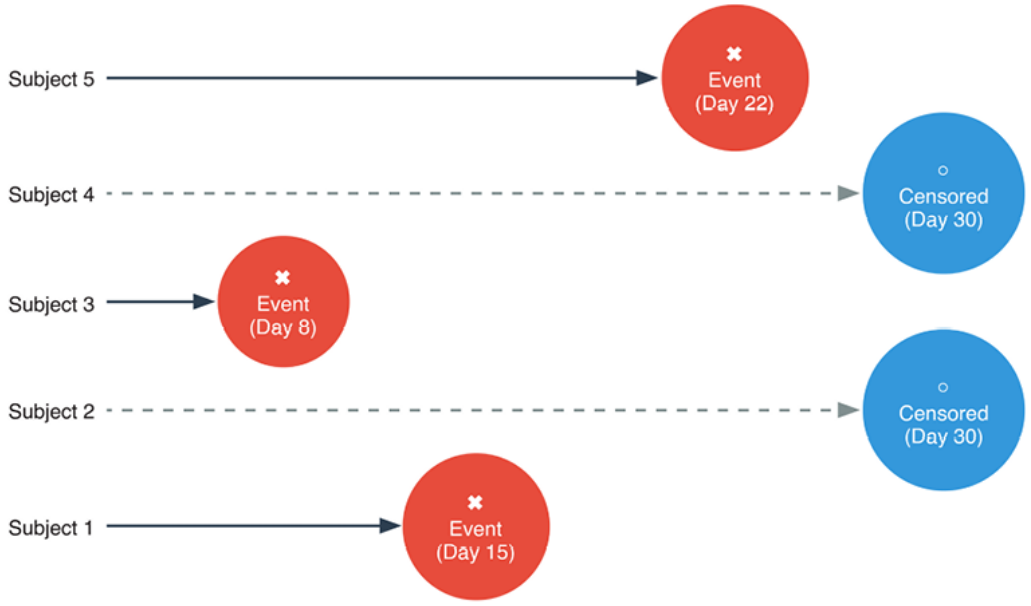


Figure 12.3: Observed events versus right-censored records on an observation timeline

A survival curve tells you how long a subject is likely to last overall. A forecasting system needs something narrower: the risk of an event in the very next interval, updated as new data arrives. The next section defines that quantity, the hazard function, and shows it is a one-step-ahead forecast in disguise.

Mathematical foundations: hazard and state-space transformations

Survival analysis rests on two functions, and the link to time series forecasting runs through both. The survival function, $S(t)$, is the probability that the event has not occurred by time t, the complement of the cumulative distribution function. The hazard function, $S(t)$, is the instantaneous rate of failure at time t, conditional on the subject having survived up to that point. Survival

is the cumulative story across a subject's whole history; hazard is the moment-by-moment one.

Cumulative survival functions provide a broad view of expected duration. They do not map cleanly to the recursive updates required in production systems. When new telemetry arrives, the system must evaluate the immediate risk of an event in the subsequent time step. This requires shifting focus from the overall probability of survival to the instantaneous rate of occurrence. To conceptualize survival models as predictive instruments, we must define the discrete-time hazard function.

Figure 12.4 visualizes this mechanism as a sequence of conditional probabilities. Continuous regression predicts an absolute magnitude at a distant horizon. In contrast, the hazard function predicts the immediate risk of an event occurring in the next interval, conditional on the subject having survived all preceding intervals. This recursive structure mathematically mirrors a discrete one-step-ahead forecast. It transforms truncated historical records into a dynamic term structure of risk.

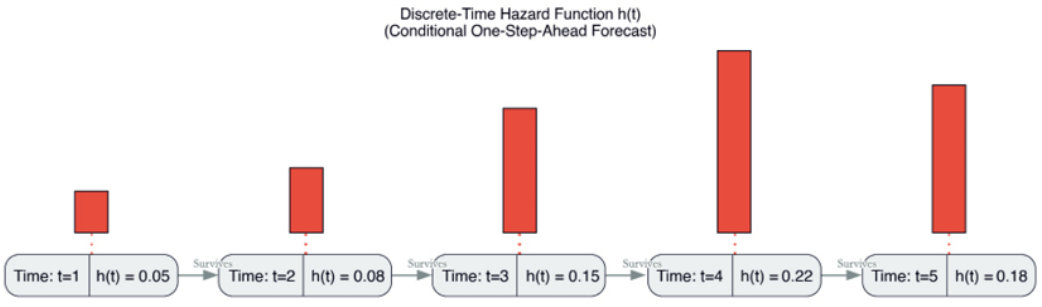


Figure 12.4: The discrete-time hazard function as a one-step-ahead forecast

Table 12.2 traces this on a single machine wearing out over five inspection periods. The hazard Y_{n+1} is the model's one-step-ahead

output, the chance the machine fails in period t given it reached t. The survival curve $Y_{n+1}\, h = 1..H) \geq 1 - \alpha$ is the running product of having cleared every period so far.

Period t	**Hazard** Y_{n+1}	**Survival** Y_{n+1}
1	0.04	0.960
2	0.07	0.893
3	0.12	0.786
4	0.20	0.629
5	0.30	0.440

Table 12.2: A rising hazard and the survival curve it implies

The hazard rises as wear accumulates, and survival falls faster as it does, because each period multiplies in another chance of failure. Predicting the hazard column is an ordinary classification task, one probability per period, which is what lets gradient boosting take over the job later in the chapter.

Before applying parametric assumptions, we must establish a non-parametric baseline to observe *conversion velocity*. The Kaplan–Meier estimator provides this by calculating the probability of survival at each distinct event time. In a marketing context, this allows us to compare how quickly different acquisition channels convert users.

```
import matplotlib.pyplot as plt
from lifelines import KaplanMeierFitter
def plot_conversion_velocity(df):
    """Visualize survival curves to compare channel performance."""
    kmf = KaplanMeierFitter()
    for channel in df["channel"].unique():
        mask = df["channel"] == channel
        kmf.fit(df[mask]["duration"],
                event_observed=df[mask]["event"],
                label=channel)
        kmf.plot_survival_function()

    plt.title("Time to Conversion by Channel")
    plt.xlabel("Days Since Acquisition")
    plt.ylabel("Probability of Non-Conversion")
    plt.show()
```

If the survival curve for *Search* descends more rapidly than *Social*, the former exhibits a higher hazard rate. This visual evidence justifies the use of more complex models to quantify exactly how covariates influence these rates.

In practical operational data, such as monthly credit defaults or daily sensor readings, time is not continuous but partitioned into discrete intervals. Bridging continuous-time survival analysis with discrete time series forecasting involves restructuring datasets into a *person-period* or expanded panel format. Each subject contributes a separate observation row for every single time interval they remain at risk. This structural expansion allows discrete-time survival analysis to be executed using standard generalized linear models, mapping the log-odds of the discrete hazard against a vector of covariates, which encompasses time-varying exogenous signals.

For dynamic systems where the hazard rate itself undergoes stochastic shifts independent of observed covariates, survival analysis integrates with state-space models. The observation equation relates the empirical data to an unobserved latent state vector representing the time-varying parameters. Because survival observations are non-Gaussian, non-linear variations such as the **Extended Kalman Filter** (**EKF**) are used to recursively estimate the posterior mode of the state vector, allowing for computationally efficient tracking and projection of dynamic hazard rates without requiring rigid parametric baseline functions.

We provide here the implementation details for the industry-standard model, emphasizing its interpretability and its treatment of multiplicative risk factors.

Implementation of the Cox Proportional Hazards model

The **Cox Proportional Hazards** (**CPH**) model is the industry standard for semi-parametric survival analysis. The CPH model represents the hazard function as the product of an unspecified baseline hazard and an exponential function of explanatory variables. It is universally deployed because it yields interpretable **Hazard Ratios** (**HR**), which directly quantify the multiplicative effect of a covariate on the baseline hazard.

The `lifelines` library provides the premier open-source architecture for classical survival inference in Python.

```
import pandas as pd
from lifelines import CoxPHFitter
def fit_cox_survival_model(
    df: pd.DataFrame,
    duration_col: str,
    event_col: str,
    penal_l2: float = 0.05
) -> CoxPHFitter:
    """
    Fit a Cox Proportional Hazards model with an L2 penalty for stabi
    """
    if duration_col not in df.columns or event_col not in df.columns:
        raise ValueError(
            "DataFrame must contain duration and event columns")

    cph = CoxPHFitter(penalizer=penal_l2)

    cph.fit(
        df,
        duration_col=duration_col,
        event_col=event_col,
        robust=True
    )

    cph.print_summary(decimals=3)
    return cph
```

This algorithm executes a partial likelihood optimization to isolate the regression coefficients. The `penalizer` parameter applies an L2 ridge-regression penalty to stabilize coefficients in the presence of collinearity. The `robust=True` parameter invokes the robust sandwich estimator for standard errors, effectively managing internal correlations within clustered data.

The returned `cph` object is what you predict with. `predict_survival_function` gives each subject a full survival curve, the probability of surviving past every time `t`; `predict_partial_hazard`

collapses that to one risk multiplier per subject, handy for ranking who is most at risk.

```
# Fit the model, then predict with the returned fitter
cph = fit_cox_survival_model(df, "duration", "event")
new_subjects = df.drop(columns=["duration", "event"])
# Per-subject survival curves: P(survive past each time t)
survival_curves = cph.predict_survival_function(new_subjects)
# A single risk multiplier per subject (the partial hazard)
risk_scores = cph.predict_partial_hazard(new_subjects)
```

The CPH model allows us to quantify the effect of features like credit scores on the instantaneous risk of default. The model assumes that covariates act as multiplicative "risk multipliers" on an unspecified baseline hazard.

```
from lifelines import CoxPHFitter
def fit_credit_risk_model(df):
    """Fit Cox model to estimate credit risk multipliers."""
    cph = CoxPHFitter(penalizer=0.1)
    # Remove identifiers before fitting
    df_fit = df.drop(columns=["id"])
    cph.fit(df_fit, duration_col="duration", event_col="event")

    # Hazard ratios < 1 indicate risk reduction
    return cph.summary[['haz_ratio', 'p']]
```

A hazard ratio of `0.8` for a credit score feature implies that for every unit increase in score, the instantaneous risk of default decreases by 20%. This provides an alternative to binary classification, which would ignore the timing of defaults.

A critical vulnerability of the CPH model is its reliance on the **Proportional Hazards** (**PH**) assumption, which dictates that

covariate effects must remain mathematically constant over the entire timeline. In dynamic environments, variables such as pricing interventions or machine vibrations exert time-varying effects. If PH violations remain uncorrected, the partial likelihood estimator converges on a biased, time-weighted average effect, distorting the survival calibration.

To resolve this, the Aalen Additive model acts as a non-parametric framework that explicitly relaxes the PH requirement. By treating the hazard function additively, it allows regression coefficients to vary dynamically, visually surfacing the exact trajectories of covariate effects. Alternatively, panel data can be structurally split into episodic intervals using the `CoxTimeVaryingFitter`, structurally decomposing a subject's timeline to accommodate changing state covariates without introducing survivorship bias.

To capture the complexity of modern operational telemetry, we must extend our survival models into the domain of non-linear machine learning. This subsection examines the transition to tree-based ensembles capable of generating multi-horizon probabilistic forecasts.

Multi-horizon forecasting and tree-based survival

The classical methodologies provided by lifelines are optimal for statistical inference and baseline visual exploration. Survival models are forecasting architectures defined over event-time rather than calendar-time.

In classical forecasting, we predict a magnitude at a future point. In survival modeling, the survival function represents a probabilistic forecast of remaining lifetime.

For a marketing conversion scenario, the expected conversions by day 30 can be calculated dynamically. If $S(t)$ forecasts the probability of non-conversion by time t, the cumulative forecast is $F(t) = 1 - S(t)$. The instantaneous forecast update is governed by the hazard rate $S(t)$, which acts as a conditional one-step-ahead forecast.

However, modern operational systems require architectures capable of processing high-dimensional, non-linear feature spaces. In this context, survival analysis must be reformulated as a multi-horizon probabilistic forecasting problem defined over event time.

As datasets scale in dimensionality, algorithmic machine learning architectures augment classical biometrics. **Accelerated Failure Time** (**AFT**) models operate directly on the survival time itself rather than the hazard, mathematically stretching or contracting the baseline survival curve based on covariates.

The lifelines library serves as a classical tool for baseline exploration and proportional hazard modeling. Conversely, the scikit-survival package facilitates the development of tree-based models and multi-horizon probabilistic forecasting.

To handle complex feature topologies without rigid distribution assumptions, practitioners transition to tree-based ensembles. **Random Survival Forests** (**RSF**) and Gradient Boosted Survival trees, implemented via the scikit-survival library, natively extend

survival modeling into the machine learning domain. Instead of modeling a global linear likelihood, RSFs employ specialized log-rank splitting criteria to partition the feature space, capturing non-linear interactions natively.

These terminal leaf nodes output individualized survival functions. By evaluating these functions at specific temporal horizons, practitioners generate multi-step probabilistic forecasts, such as a 14-day marketing conversion probability or a 36-month loan default term structure. This structural transition allows operations to bypass the limitations of single-horizon logistic regression and directly integrate time-to-event probabilities into forward-looking revenue and risk models.

The initial step in analyzing truncated data involves estimating the baseline survival function. Non-parametric methods provide a reliable historical average for specific cohorts. Operational forecasting requires predicting outcomes for individual sequences using high-dimensional features. Tree-based machine learning models resolve this limitation by estimating cumulative hazard functions natively. The transition from classical survival analysis to modern machine learning requires a shift in forecast representation.

Figure 12.5 contrasts these two paradigms. The *first* panel illustrates the Kaplan-Meier estimator, which calculates average survival trajectories for broad demographic cohorts. While mathematically sound for historical analysis, this approach fails to isolate individual risk. The *second* panel demonstrates the output of a random survival forest. This model generates a structured predictive matrix, providing distinct cumulative failure probabilities for each subject

across specific temporal intervals. This structure directly aligns the outputs of survival models with standard multi-horizon forecasting architectures.

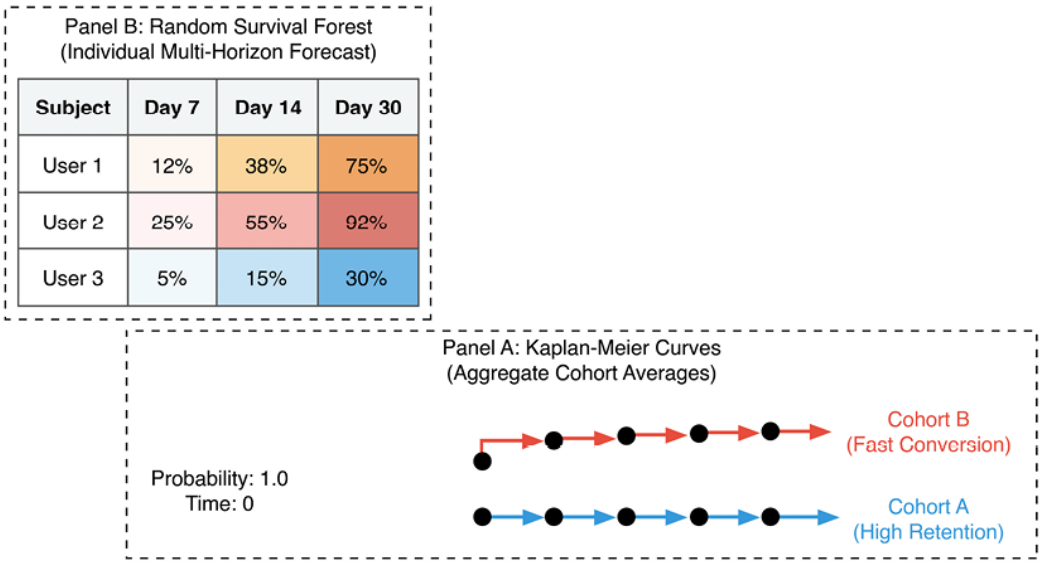


Figure 12.5: The Kaplan-Meier estimator vs. a random survival forest

To illustrate this architectural transition, consider the following implementation. It applies a Random Survival Forest to a right-censored dataset to extract multi-horizon probabilities.

```
import pandas as pd
from sksurv.ensemble import RandomSurvivalForest
from sksurv.util import Surv
def train_and_forecast_survival(df: pd.DataFrame, horizons: list[int]
    """Train a survival forest and extract probabilistic forecasts.""
    # Construct the structured survival array
    y = Surv.from_arrays(event=df["event"], time=df["duration"])
    X = df[["ad_spend", "page_views", "channel"]]

    # Initialize and fit the tree-based ensemble
    model = RandomSurvivalForest(
        n_estimators=200, min_samples_leaf=15, random_state=42)
    model.fit(X, y)
```

```
    # Generate continuous survival functions for each subject
    surv_funcs = model.predict_survival_function(X)

    # Extract discrete probabilities at specified horizons
    forecasts = []
    for fn in surv_funcs:
        prob_converted = 1 - fn(horizons)
        forecasts.append(prob_converted)

    columns = [f"prob_convert_{h}d" for h in horizons]
    return pd.DataFrame(forecasts, columns=columns, index=X.index)
```

By calculating $\hat{y}_{t+h} = l_{t\cdot}$ for specific horizons, we move from descriptive statistics to a structured forecast table. This allows us to predict the probability that a specific user will convert within the next 7, 14, or 30 days.

Mathematical precision in survival modeling is of limited consequence without a methodology for validating probabilistic accuracy. We now introduce the specific metrics required to judge both the discriminatory power and the calibration of survival outputs.

Evaluating time-to-event calibration

To resolve the bias inherent in standard metrics, we utilize evaluation frameworks designed for right-censored data. We must distinguish between discrimination—the ability to correctly rank subjects by risk—and calibration—the accuracy of the predicted probabilities.

Harrell's Concordance Index (**C-index**) is the standard for discrimination. It calculates the proportion of pairs where the model correctly predicts that the subject who experienced the event first had a higher risk score. However, under heavy censoring, this metric suffers from structural bias. Uno's C-index applies **Inverse Probability of Censoring Weighting** (**IPCW**) to provide an unbiased estimate.

For calibration, we use the **Integrated Brier Score** (**IBS**), which measures the mean squared difference between the predicted survival probability and the actual status (event or censored) across all time horizons.

```
import numpy as np
from sksurv.metrics import concordance_index_censored, integrated_bri
from sksurv.util import Surv
def evaluate_survival_performance(model, X_test, y_test, y_train):
    """Calculate discrimination and calibration metrics for censored
    # Discrimination: Harrell's C-index
    risk_scores = model.predict(X_test)
    c_index = concordance_index_censored(
        y_test["event"],
        y_test["time"],
        risk_scores
    )[0]
    # Calibration: Integrated Brier Score
    # Extract survival probabilities at specific percentiles
    surv_funcs = model.predict_survival_function(X_test)
    times = np.percentile(y_train["time"], [25, 50, 75])
    preds = np.row_stack([fn(times) for fn in surv_funcs])
    ibs = integrated_brier_score(y_train, y_test, preds, times)
    return {"c_index": c_index, "ibs": ibs}
```

The `y_train` and `y_test` arguments must be `sksurv` structured arrays (an event flag paired with a time), built from the raw DataFrame columns with Surv:

```
y_train = Surv.from_arrays(event=train_df["event"],
                           time=train_df["duration"])
y_test = Surv.from_arrays(event=test_df["event"],
                          time=test_df["duration"])
metrics = evaluate_survival_performance(model, X_test, y_test, y_trai
```

A C-index of `0.5` indicates a model no better than random guessing, while a score of `1.0` represents perfect discrimination. Unlike RMSE, these metrics allow for the inclusion of censored records, providing a mathematically resilient evaluation of the forecasting pipeline's reliability.

The deployment of these deep algorithms exposes operations to **predictive multiplicity**. Complex datasets often yield multiple distinct model architectures that achieve statistically equivalent overall metrics while producing conflicting estimations of failure risk for the exact same entity. This multiplicity manifests as **ambiguity** (divergence in predicted class thresholds), **discrepancy** (variance in continuous probabilities at a specific time), and **obscurity** (unexplainable shifts in model confidence at extreme time horizons). Consequently, modern survival frameworks require the explicit calculation of model uncertainty to build genuine operational trust. Automated preprocessing frameworks, such as `CleanSurvival`, further stabilize inputs by using Q-learning reinforcement agents to optimize missing value imputation specifically tailored for censored distributions.

Integrating longitudinal risk into existing forecasting pipelines often requires a structural bridge between continuous-time theory and discrete-time practice. We now explore how continuous event times can be mapped into discrete, longitudinal panels.

Discrete-time hazard forecasting at scale

While continuous-time algorithms generate precise individualized distributions, they remain structurally isolated from standard time-series forecasting ecosystems. To integrate time-to-event probabilities into `standard gradient-boosting workflows`, the underlying mathematics must be reformulated.

By expanding cross-sectional, right-censored records into longitudinal panel sequences, we transition the objective from modeling the continuous time of an event to predicting a sequence of conditional probabilities. Each subject contributes a separate observation row for every single time interval they remain at risk. The target variable becomes a binary indicator of whether an event occurred at that specific interval.

This structural expansion converts survival forecasting into a multi-step binary classification problem. It permits practitioners to apply optimized gradient boosting algorithms, such as LightGBM, directly to event-time forecasting. It accommodates time-varying covariates and outputs multi-horizon probability vectors that natively satisfy hierarchical reconciliation constraints that we've discussed in *Chapter 6*.

The following code demonstrates this mathematical reformulation. By expanding cross-sectional records into a longitudinal format, the sequence becomes compatible with standard gradient boosting frameworks.

```
import numpy as np
import pandas as pd
def to_person_period(df, max_horizon=30):
    """Expand right-censored records into a discrete-time hazard pane
    limit = np.minimum(np.ceil(df["duration"]).astype(int), max_horiz
    panel = df.loc[df.index.repeat(limit)].copy()
    panel["ds"] = panel.groupby(level=0).cumcount() + 1
    # the event fires once, in the subject's final at-risk period
    final = panel["ds"].to_numpy() == limit.loc[panel.index].to_numpy
    panel["event"] = (
        final & (panel["event"].to_numpy() == 1)).astype(int)
    return panel.rename_axis("unique_id").reset_index()
```

A subject is at risk from period 1 through the interval that holds its event or censoring time. A loan that defaults in month five contributes five rows with the event flag on the fifth; a loan still active at the horizon contributes only zeros. Right-censoring needs no special case; the subject simply stops producing rows.

Table 12.3 shows the reshape on two subjects. Before expansion each is a single row: **subject 7** has **duration = 3**, **event = 1**, and **subject 9** has **duration = 3**, **event = 0.** After expansion each contributes one row per at-risk period, and the event flag moves onto the correct row.

unique_id	ds	event	note

7	1	0	at risk, no event
7	2	0	at risk, no event
7	3	1	event fires in the final period
9	1	0	at risk, no event
9	2	0	at risk, no event
9	3	0	censored, no event ever recorded

Table 12.3: A right-censored record set before and after person-period expansion

Subject 7's single **event = 1** lands on its last row and every earlier row reads zero; **subject 9**, censored, produces three zero rows and no event at all. The binary event column is now a per-period target a classifier can fit directly.

Intermittent demand needs no expansion at all: it already arrives in this shape. A retail panel of daily sales is one row per SKU per day, and each row asks the same binary question: did a demand event occur? The number of days since the SKU's last sale is the time it has been at risk, so the discrete-time hazard is the probability that the next sale lands today. We build that waiting-time feature from past demand only, which keeps it leak-free.

```
# panel: one row per SKU per day, demand = 1 when a sale occurred
panel = panel.sort_values(["sku", "date"])
# days_since_last_demand is the hazard's time axis, built from shifte
# demand so it depends only on the past
```

```
prior_demand = panel.groupby("sku")["demand"].shift(1, fill_value=0)
panel["spell"] = prior_demand.groupby(panel["sku"]).cumsum()
panel["days_since_last_demand"] = panel.groupby(
   ["sku", "spell"]).cumcount() + 1
panel["dayofweek"] = panel["date"].dt.dayofweek
```

A gradient-boosted classifier predicts that hazard directly. Splitting by SKU, not by row, keeps every day of a SKU on one side of the split and prevents leakage.

```
from lightgbm import LGBMClassifier
from sklearn.model_selection import train_test_split
feature_cols = ["days_since_last_demand", "dayofweek"]
train_skus, test_skus = train_test_split(
    panel["sku"].unique(), test_size=0.3, random_state=42
)
train = panel[panel["sku"].isin(train_skus)]
test = panel[panel["sku"].isin(test_skus)].copy()
model = LGBMClassifier(random_state=42, verbose=-1)
model.fit(train[feature_cols], train["demand"])
# the predicted probability of demand is the per-period hazard of the
test["hazard"] = model.predict_proba(test[feature_cols])[:, 1]
```

Scoring this with RMSE would miss the point twice over: the target is a yes/no occurrence rather than a magnitude, and the dry spell after the last observed sale is right-censored, since we do not know when the next sale will land. Two metrics built for this case avoid the trap. The Brier score checks calibration, whether the predicted hazards are numerically accurate, and the ROC AUC checks discrimination, whether the model ranks demand days above quiet ones.

```
from sklearn.metrics import roc_auc_score, brier_score_loss
auc = roc_auc_score(test["demand"], test["hazard"])
brier = brier_score_loss(test["demand"], test["hazard"])
```

On the UCI Online Retail panel this model reaches an AUC near 0.74. What matters more than the number is what the model produces: a probability of demand for each period, conditioned on how long the SKU has waited. That is exactly the quantity TSB smooths by hand at the start of the chapter. Intermittent-demand forecasting and discrete-time survival analysis turn out to be one problem, and the companion repository runs this example end to end alongside the loan-portfolio hazard model.

The theoretical and computational methodologies discussed throughout this chapter find their ultimate justification in their applied industrial utility. This final section provides a synthesis of how these models function within live operational environments.

Applied survival and reconciliation

The mathematical theories of intermittent distributions, hierarchical aggregation, and survival probability converge forcefully within specific industrial and financial applications.

In consumer markets, dynamic customer churn modeling abandons static classification for time-to-event profiling. By deploying Cox Proportional Hazards and Aalen Additive models in tandem, businesses generate individual-level risk profiles that segment customers based on both the magnitude of churn likelihood and the precise timing of anticipated attrition. Cumulative-coefficient plots

reveal transient shocks to the hazard rate, such as service incidents or contract renewal dates, enabling retention teams to map estimated hazards to specific, actionable outreach windows before the mathematical risk curve enters a steep decline.

In industrial **Predictive Maintenance (PdM)**, the objective is to shift physical operations from reactive repair to optimized scheduling. Survival analysis naturally handles censored assets: machines that are operational at the time of data extraction. Continuous IoT sensor telemetry is structurally transformed into discrete intervals, from which rolling statistical features are extracted using libraries like `pandas` and `scipy`. Aligning these lagged, engineered features with binary failure indicators maps the time series problem to an extended survival formulation. The deployment of AFT models and Random Survival Forests subsequently identifies the exact combinations of sensor anomalies most correlated with imminent failure.

Within the heavily regulated financial sector, particularly in London governed by the **Prudential Regulation Authority** (**PRA**), survival analysis is a strict mathematical mandate. The International Financial Reporting **Standard 9** (**IFRS 9**) replaces backward-looking provisions with a forward-looking **Expected Credit Loss** (**ECL**) model. This standard dictates that institutions project a **Probability of Default** (**PD**) term structure extending potentially decades into the future.

Using discrete-time hazard models and expanded panel datasets, banks dynamically project marginal and cumulative default probabilities. These frameworks explicitly condition survival

probabilities on multiple macroeconomic scenarios using **Error Correction Models (ECMs)**, separating idiosyncratic borrower risk from systemic economic shocks. The PRA's SS11/13 guidelines and the upcoming Basel 3.1 framework impose strict governance over these outputs, scrutinizing historical bias, structural stability, and the incorporation of climate risk overlays. To manage these validation requirements, quantitative compliance teams increasingly deploy automated governance pipelines in Python, ensuring that every data expansion, feature transformation, and hazard calibration step remains fully transparent and mathematically reproducible.

Summary

In this chapter, we departed from continuous assumptions to address the irregularities of operational data. Standard regression fails when applied to sequences dominated by zero values and right-censored records.

We explored generative distributions to forecast intermittent demand without regressing to zero. We expanded our analysis to time-to-event problems by adopting machine learning survival models. By using algorithms such as random survival forests, we quantified risk profiles and modeled censored telemetry.

As models enter production environments, structural correctness proves more useful than computational complexity. An approach that respects censoring boundaries offers more utility than an intricate architecture that ignores the temporal reality.

You have learned to:

- Forecast intermittent demand using generative distributions without triggering obsolescence failures
- Quantify time-to-event risks by applying survival forests to censored records
- Manage uncertainty by framing hazard rates as conditional forecasts
- Evaluate model calibration accurately through expanding-window backtests

The conclusion of this chapter marks the end of the book. We began with principles and advanced through modern computational implementations to reach production-grade system design. The objective throughout has remained constant: to enable the construction of reliable forecasting pipelines that withstand the rigors of professional environments.

Theoretical knowledge holds little value if it cannot survive the transition to live data. As you apply these methodologies to your own data streams, prioritize structural correctness over algorithmic complexity. A simple model that respects temporal boundaries will consistently outperform a complex architecture that ignores the nature of the underlying data. You are now equipped to work in modern operational systems and build mathematically sound applications.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers.

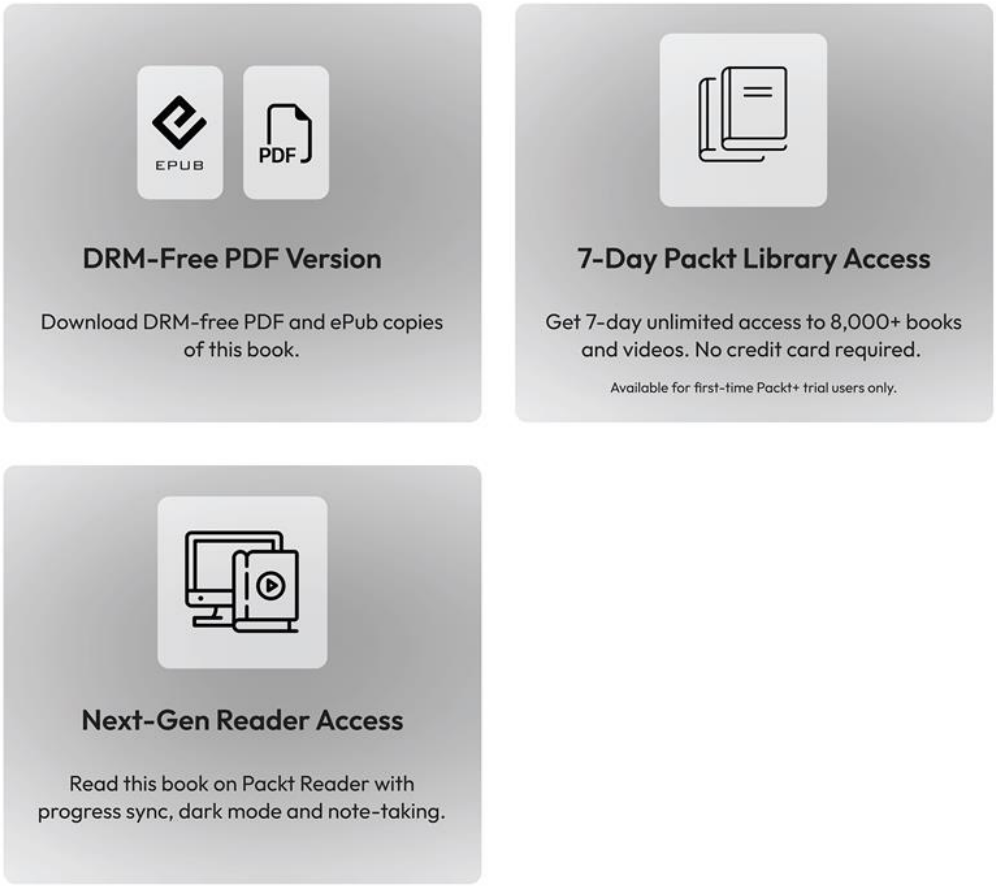
https://discord.gg/VXpz9YPUTW



13

Unlock Your Exclusive Benefits

Your copy of this book includes the following exclusive benefits:



Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

Unlock this Book's Free Benefits in 3 Easy Steps

Step 1

Keep your purchase invoice ready for *Step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit https://www.packtpub.com/en-us/unlock?step=1.



Note: If you bought this book directly from Packt, no invoice is required. After *Step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to packtpub.com/unlock.

On the page that opens (similar to *Figure 13.1* on desktop), search for this book by name and select the correct edition.

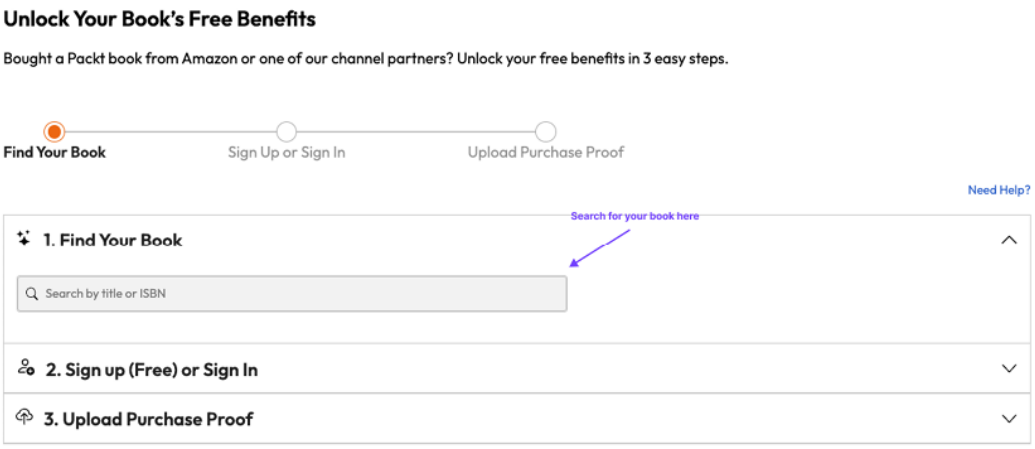


Figure 13.1: Packt unlock landing page on desktop

Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need Help

If you get stuck and need help, visit https://www.packtpub.com/unlock-benefits/help for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.



Note: If you are still facing issues, reach out to `customercare@packt.com`.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

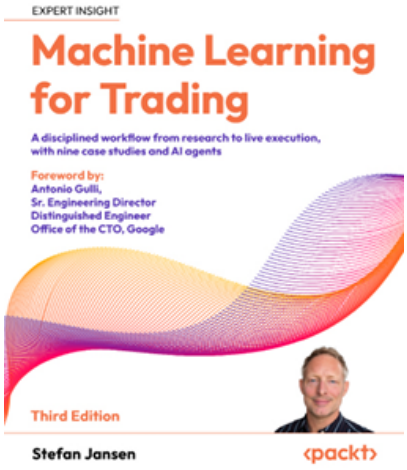
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Machine Learning for Trading, Third Edition

Stefan Jansen

ISBN: 9781803246970

- Transform raw data into predictive alpha factors, validated with leak-proof cross-validation
- Master advanced models, from Gradient Boosting Machines to Transformers, Graph Neural Networks, and Reinforcement Learning agents

- Harness Generative AI, Retrieval Augmented Generation, and Causal Inference to make models interpretable, auditable, and compliant with regulatory standards
- Build production-ready trading infrastructure using MLOps, feature stores, and model monitoring to transition research into live capital deployment safely



Time Series with PyTorch

Graeme Davidson, Lei Ma

ISBN: 9781805128182

- Build, train, and evaluate neural networks for time series using PyTorch and PyTorch Lightning. Tune models with Bayesian optimisation and validate them with suitable metrics and strategies.
- Progress from feedforward and recurrent networks to transformers and models such as N-BEATS, N-HiTS, and TFT.
- Learn how global models use cross- and transfer learning across many series.
- Generate synthetic series and representations with diffusion and self-supervised methods.
- Apply modern approaches to classification, clustering, and anomaly detection.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Machine Learning for Time Series with Python, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

2D Convolutional Neural Networks 259-262

A

Accelerated Failure Time (AFT) 428

active detection 394

adaptive systems

building 13-16

adaptive windowing (ADWIN) 395-398

additive model 43

advanced feature engineering 154

cyclical time, encoding 157, 158

domain-specific features 158-162

lag features 154, 155

practical guide 163-165

reusable pipeline, building 165

rolling window statistics 156, 157

advanced models, failure reasons

adaptive systems, building 13

fundamental gaps 12

inability to adapt to change 11
integration breakdown 11
misleading validation metrics 10
Aggregate-Disaggregate Intermittent Demand Approach (ADIDA) 415
aggregation consistent 203
Akaike Information Criterion (AIC) 192
algorithms
shortcomings 9
Amazon Chronos 328-332
architectural patterns, for time-series deployment 356
batch processing architectures 357
kappa architecture 360
lambda architecture 360
Area Under the ROC Curve (AUC) 12
Augmented Dickey-Fuller (ADF) 84
autocorrelation 28
analysis approach 33
patterns, for modeling strategy 31-33
Autocorrelation Function (ACF) 59, 155, 308
autoencoder 179
automated feature engineering 173-175
functime 178, 179
mlforecast and XGBoost 175

tsfresh and Darts 176-178

AutoML platforms 133-135

AutoRegressive Integrated Moving Average (ARIMA) 78-84

Average Demand Interval (ADI) 413

B

Baseline Illusion 318

batch processing architectures 357

global context and robustness 358, 359

schedule-compute-cache pattern 357

stream processing (real-time) architecture 359

Bayesian Neural Networks 274

Bayesian Structural Time Series (BSTS) 404

Benchmark Challenge 137

BigQuery ML 328

bullwhip effect 188

C

calibration drift, monitoring

recalibration strategy 301

rolling coverage dashboard, building 299, 300

CausalImpact

used, for performing intervention analysis 404-407

Causal inference 401-404

benchmarks 408

intervention analysis, with CausalImpact 404-407
channel-dependent (CD) 249
Channel Independence (CI) 249
Channel-Independent Models 195, 196
Channel-Mixing Models 195
Chronos-Bolt 329
classical baselines 74
AutoRegressive Integrated Moving Average (ARIMA) 78-84
exponential smoothing 75-77
statistical forecasting methods, overview 88-90
Theta method 85-87
Coefficient of Variation 413
coherent 203
Cold Start problem 371
concept drift 9, 394
versus seasonality 377
conditional coverage 286
conditional coverage failure 301
Confidence-Based Performance Estimation (CBPE) 377-379
conformalized quantile regression (CQR)
adaptive intervals, using with 283-286
conformalization 287
final prediction interval 287
Nonconformity Score 287

quantile of calibration scores 287
quantile regression 287
conformal prediction, for non-exchangeable time series 292
ACI, implementing 294, 295
EnbPI, implementing 293
multi-horizon coverage 295, 296
conformal prediction framework 274
guarantee 275
split conformal algorithm 275-278
covariate shift 394
Cox Proportional Hazards (CPH) model
implementing 425-427
Crossformer 256-259
Cross-Series Learning 371
Croston's method 90
cyclical time
encoding 157, 158

D

darts 129, 130, 176-178
data engineering 364
data gaps, handling 368, 369
data validation, with Pandera 370
irregularities and censored data, handling 368, 369
point-in-time correctness and time travel 365

data-generating process 5

dataset shift 394

data staleness 359

decision framework 54

decomposition signals 42

- additive, versus multiplicative model 43
- in python 44-46
- method, selecting 47

deep learning 222-227

deep learning models

- training, practical tips 268

Delayed Feedback Problem 377

deployment strategies 373

- model serialization, and MLflow 375, 376
- shadow deployment 373-375

Deseasonalized Residuals 377

detection delay 397

Dimension-Segment-Wise (DSW) 256

distribution drift 274, 299

DLinear model 234, 238

Domain Specific Language (DSL) 360

Drift Detection Method (DDM) 395

Dynamic Factor Models 195

Dynamic Time Warping (DTW) 384

E

Early Drift Detection Method (EDDM) 395

Electric Reliability Council of Texas (ERCOT) 354

Electronic Numerical Integrator and Computer (ENIAC) 3

Error Correction Models (ECMs) 436

Error, Trend, and Seasonality (ETS) 75

event-driven inference

- event-driven 359
- micro-batching 359

exchangeability 275

exogeneous 81

Expected Credit Loss (ECL) 436

Extended Kalman Filter (EKF) 425

F

failure modes and mitigations

- bullwhip effect 380, 381
- censored data 379
- demand death spiral 379
- feedback loops 380, 381

false alarm rate 397

Fast Fourier Transform (FFT) 260

Fast Gradient Sign Method (FGSM) 344

feature drift 377-394

feature engineering 148-152
calendar features 153
event and holiday features 153, 154
Feature Store 365
Feed-Forward Network (FFN) 254
financial volatility
clustering, identifying 302-304
exchangeability of standardized residuals, validating 305, 306
GARCH, combined with conformal prediction 304, 305
modeling 302
production and deployment patterns 307
fixed-width intervals 308
failure 286, 287
using, with mapie 279-283
forecastability assessment 63
for project viability framework 67, 68
predictability, measuring 63, 64
forecasting
as regression task 118-120
forecasting baselines role 74
forecasting methods
evolution 2, 3
forget gate 228
foundational models 228

long short-term memory 228-230
Temporal Convolutional Networks (TCNs) 230-233
foundation model architectures
taxonomy 314
frequency conversion 53
functime 178
fundamental gaps 12
fixed models, issue 13
misalignment. with business needs 12
temporal assumption violations 12

G

gates 228
Generalized Autoregressive Conditional Heteroskedasticity (GARCH) 303
global context 358
global cross-learning models 200
global models 371
global multi-series models 196-200
cross-learning (global univariate) 196
cross-series dependencies (multivariate) 197
Google TimesFM 323-328
Gradient Boosted Decision Trees (GBDTs) 166
gradient boosting 166
Granger causality 192

Graph WaveNet 399

grouped time series 203

H

harmonic regression with ARIMA 80

Harrell's Concordance Index (C-index) 430

hazard function 422

Hazard Ratios (HR) 425

hierarchical forecasting 152, 184, 358

Australian tourism forecasting case study 213-217

hierarchical models 201

hierarchical reconciliation 200

hierarchical time series 202

Human Activity Recognition (HAR) 388

hybrid frameworks 395

Hybrid GARCH-Conformal Model 308

I

idempotency 358

inductive bias 233

industry-specific validation strategy 110-114

ingestion 359

input gate 228

Integrated Brier Score (IBS) 431

intermittent demand forecasting 412

deep learning architectures, for zero-inflated data 420, 421
hierarchical Bayesian formulations 414, 415
intermittent heuristics implementation 415, 416
sparsity and obsolescence 413, 414
stacking and hurdle-based combinations 416-419
Intermittent Multiple Aggregation Prediction Algorithm (IMAPA) 415
Inverse Probability of Censoring Weighting (IPCW) 430
iTransformer 253-255

J

joint dynamics 186
just-in-time (JIT) compiler 131

K

Kappa Architecture 360-364
K-S (Kolmogorov-Smirnov) Tests 395

L

lag features 154
lambda architecture 360-362
batch layer (cold path) 360
speed layer (hot path) 360
lambda processing 360
Large Language Models (LLMs) 223, 313

left-truncation 421

LightGBM 48

fourier features, for seasonality 173

statistical method, to decompose series into trend 172

Linear and MLP-based models 233, 234

DLinear model 234-238

N-BEATS 239, 240

TSMixer 240-245

local models 371

Long Short-Term Memory (LSTMs) 9, 228-230

Low-Rank Adaptation (LoRA) 332

M

Machine Learning (ML) 13, 28

mapie

split conformal algorithm, establishing with 279-283

used, for implementation of CQR 288, 289

Masked Language Modeling 332

mathematical foundations, for data 28

decomposition signals 42

stationarity testing and interpretation 34, 35

temporal dependence and autocorrelation 28, 29

Mean Absolute Error (MAE) 23, 105, 113, 416

Mean Absolute Percentage Error (MAPE) 23, 105, 106, 316, 413

Mean Absolute Scaled Error (MASE) 105, 106, 138, 316

metrics 93, 94
experiment 107-109
performance, measuring 104
scale-dependent metrics 105
scale-independent metrics 105
Minimum Trace (MinT) 208, 358
workflow 209
MiniRocket 389-392
MLflow 375
ML forecast 175, 197
creating 137
ML forecasting workflow 120, 121
data preparation 122
feature engineering 122
model training 122
prediction 123
tooling, role 123-125
tools, selecting 126-128
validation 123
ML forecasting workflow, tools
AutoML platforms 133-135
darts 129, 130
decision framework 136, 137
direct scikit-learn integration 135, 136

Nixtla ecosystem 131-133
sktime 128, 129
sktime 129
model selection 263-267
modern time series ecosystem
DARTS (native conformal models) 291, 292
nixtla/statsforecast 290
recipes 289
sktime 290
momentary conditional independence (MCI) 402
Multi-Layer Perceptron (MLP) 233
multimodal forecasting 336
dataset 340
OpenTSLM 336-338
prompt engineering for multimodal LLMs 339
multiplicative model 43
MultiRocket 392
multivariate forecasting 184, 185
scenarios 186-188
multivariate models 185, 200
multivariate time series 5
mutable data 365

N

naive model 64-66

NannyML 377

Natural Language Processing (NLP) 313

nested cross-validation 101-104

Neural Basis Expansion Analysis for Time 239, 240

Neural Basis Expansion Analysis for Time Series Forecasting (N-BEATS) 238

neural multivariate models 195

Neural ODE 400

Nixtla ecosystem 131-133

nonstationarity management

- adaptive windowing (ADWIN) 396-398
- diffusion models, for time series 394
- drift detection methods 395
- graph and spatiotemporal modeling 399, 400
- neural ODEs for irregular time series 400, 401

O

OpenTSLM 336-338

output gate 228

P

Pandera

- used, for data validation 370

Partial Autocorrelation Function (PACF) plot 59

passive adaptation 394

patches 249

patching 249

Patch Time Series Transformer (PatchTST) 195, 249-253

pattern discovery, via visualization 55

autocorrelation and residual analysis 59-62

visual diagnostics for temporal structures 55-59

pattern, for conceptual foundation and practical framework

independence assumptions, to temporal structure respect 23

model artifact focus, to production system design 24

optimization mindset, to adaptation infrastructure 21

point estimate fixation, to uncertainty aware decision-making 22

static system thinking to dynamic system design 21

technical metric obsession, to business value alignment 23

pitfalls

avoiding 20

Point-in-Time Correctness (PIT) 365

Feature Store solution 365-368

predictability

measuring 63

naive model 64, 66

seasonal naive model 66, 67

signal-to-noise proxy 64

prediction drift 377, 394

Prediction-Interval Coverage Probability (PICP) 346

Predictive Maintenance (PdM) 436

proactive adaptation 395

Probability of Default (PD) 436

Production Gap 352-56

production-ready uncertainty 297

- calibration drift, monitoring for 299
- conditional coverage, diagnosing 301, 302
- interval quality, evaluating 297-299
- rolling coverage dashboard, building 299, 300

prompt engineering for multimodal LLMs 339

Proportional Hazards (PH) 427

Proxy Metrics 377

Prudential Regulation Authority (PRA) 436

Python

- core principles, for production-ready systems 20
- features 16
- right tools, selecting for time series problem 17

PyTorch Geometric Temporal 400

Q

quantitative foundation models 318-323

- Amazon Chronos 328-332
- Google TimesFM 323-328
- Salesforce Moirai 332-335

R

random splits 95-97

window, expanding 97, 98

window, rolling 99, 100

Random Survival Forests (RSF) 428

Ray

used, for distributed forecasting 371-373

reconciliation approaches 204-206

bottom-up approach 206

middle-out approach 207

modern reconciliation techniques 207

optimal combinations 207-213

top-down approach 207

Recurrent Neural Network (RNN) 228, 389, 420

regression task

forecasting as 118-120

regularized or Bayesian VAR 194

representation learning 179

resampling error 54

Reversible Instance Normalization (RevIN) 324

right-censoring 421

right tools, selecting for time series problem

anomaly detection 18

classification 18

forecasting 17
regression 19
rolling window statistics 156, 157
Root Mean Squared Error (RMSE) 12, 105, 354
Root Mean Squared Scaled Error (RMSSE) 413

S

Salesforce Moirai 332-335
scale-independent metrics 105
Mean Absolute Percentage Error (MAPE) 106
Mean Absolute Scaled Error (MASE) 106
scaling strategies 371
distributed forecasting, with ray 373
distributed forecasting, with Ray 371
hierarchical reconciliation, at scale 373
local vs. global modeling paradigms 371
Schedule-Compute-Cache pattern 357
scikit-learn model, for ML forecast
creating 138
manual feature engineering 139, 140
prediction 142, 143
setup and baseline 138, 139
training 140, 141
validation 144, 145
scikit-learn regressors

using 135, 136
seasonal naive model 66, 67
Shale-Boylan-Johnston (SBJ) 413
simple VAR model 190-192
sktime 128, 129
smart models
failure reasons 7-9
split conformal prediction 275
Standard Split-Conformal Prediction 307
stationarity 34
achieving 39-41
statistical stationarity tests 37-39
variance, stabilizing with log and Box-Cox transformations 41
visualizing 35-37
statistical forecasting methods
overview 88-90
remote work 90-93
statistical methods 72, 73
stream processing (real-time) architecture 359
event-driven inference and state management 359
training-serving skew challenge 360
structural break 51
structural pattern recognition 384
supply gap 368

survival analysis 421-425

Symmetric MAPE (sMAPE) 316

Syntetos-Boylan Approximation (SBA) 413

Syntetos-Boylan criteria 413

T

Temporal Convolutional Networks (TCNs) 227-233

temporal cross-validation 96

temporal data leakage 365

temporal dependence 27, 28

structure 29-31

Temporal Fusion Transformer 195

Teunter-Syntetos-Babai (TSB) 413

Text-to-Text Transfer Transformer (T5) 328

Theta method 85-87

TimeGrad 392

time series 4, 5

data components 5

multivariate 5

univariate 5

using, in python context 6

time series applications

advanced models, failure reasons 10

Time Series Classification (TSC) 384

diffusion models, for time series 392, 393

human activity recognition problem 388-392
self-supervised representation 385-387
sktime, versus aeon 387
time series data pipeline
building 48
frequency conversion and resampling decisions 53
missing data assessment 51
quality checks and missing data 50
quality threshold decision-making 52
robust conversion and validation 50
structural break, detecting 51
temporal indexing and frequency management 48, 49
Time-Series Foundation Models (TSFMs) 4, 323
adversarial fragility 344
example 315-318
foundation models, deploying 347, 348
generalization risks 343
long-horizon decay 346, 347
memorization risks 343
need for 315
performance risks 342
probabilistic calibration 345
Time Series Graph-based Measures of Information Transfer (Tigramite) 402

Time-Series Language Models (TSLMs) 315, 336

time-series systems

monitoring 377

observability 377

time-to-event 421

discrete-time hazard forecasting, at scale 432-435

multi-horizon forecasting 427-430

tree-based survival 427-430

time-to-event calibration

evaluating 430-432

Training-Serving Skew 360

transformer models 245

Crossformer 256-259

iTransformer 253-256

PatchTST 249-253

temporal fusion transformer (TFT) 246-249

tree-based model 166

tree model

selecting 166-172

tsfresh 176-178

TSMixer 241-245

Two-Stage Attention (TSA) 256

U

uncertainty

aleatoric uncertainty 274
distributional uncertainty 274
epistemic uncertainty 274
problem 273
uncertainty method
choosing 307-309
feature-driven (heteroskedastic) errors 308
stable (homoskedastic) errors 308
time-driven (volatile) errors 308
unstable errors (distribution drift 309
univariate forecasting 183
univariate forecast with exogenous variables 186
univariate time series 5

V

validation 93, 94
advanced techniques 100
industry-specific validation strategy 110-114
random splits 94-97
validation, advanced techniques
nested cross-validation 101-104
walk-forward validation 100
validity guarantee 287
Value-at-Risk models 19
VARMAX 184-194

Vector Autoregression (VAR) 152, 184, 189

Granger causality 192

Vector Error Correction Models (VECM) 194

Vision Transformers (ViT) 332

volatility clustering 302

identifying 302, 303

W

walk-forward validation 100

X

XGBoost 48, 175