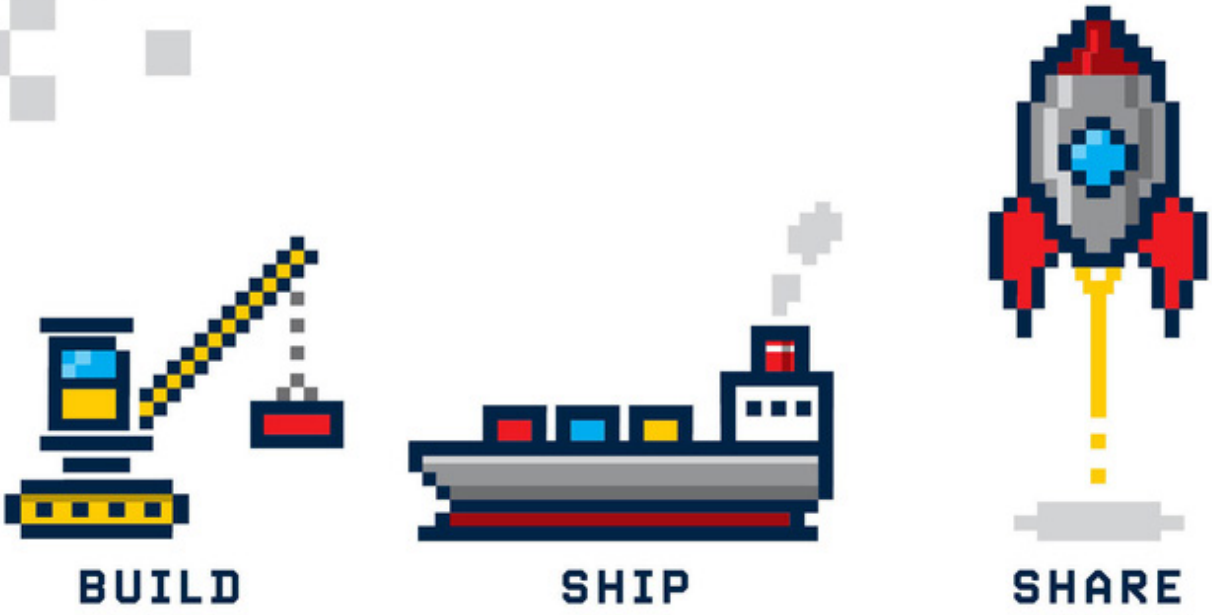

THE AI ENGINEERING BOOTCAMP

DR. GREG LOUGHNANE, PH.D.
WITH
CHRISTOPHER BROUSSEAU *AND*
C.S. ALEXIUK

WILEY

The AI Engineering Bootcamp



The AI Engineering Bootcamp

Build, Ship, Share

Dr. Greg Loughnane with
Christopher Brousseau and
C. S. Alexiuk

WILEY

Copyright © 2027 by John Wiley & Sons, Inc. All rights reserved, including rights for text and data mining and training of artificial intelligence technologies or similar technologies.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permission.

The manufacturer's authorized representative according to the EU General Product Safety Regulation is Wiley-VCH GmbH, Boschstr. 12, 69469 Weinheim, Germany, Product_Safety@wiley.com.

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Limit of Liability/Disclaimer of Warranty: While the publisher and the author have used their best efforts in preparing this work, including a review of the content of the work, neither the publisher nor the author make any representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation any implied warranties of merchantability or fitness for a particular purpose. Certain AI systems have been used in the creation of this work. No warranty may be created or extended by sales representatives, written sales materials or promotional statements for this work. The fact that an organization, website, or product is referred to in this work as a citation and/or potential source of further information does not mean that the publisher and author endorse the information or services the organization, website, or product may provide or recommendations it may make. This work is sold with the understanding that the publisher is not engaged in rendering professional services. The advice and strategies contained herein may not be suitable for your situation. You should consult with a specialist where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002. For product technical support, you can find answers to frequently asked questions or reach us via live chat at https://support.wiley.com.

If you believe you've found a mistake in this book, please bring it to our attention by emailing our reader support team at wileysupport@wiley.com with the subject line "Possible Book Errata Submission."

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data: 2026952391

ISBN: 9781394324057 (Paperback)
ISBN: 9781394324071 (ePDF)
ISBN: 9781394324064 (ePub)

Cover Design: Wiley
Cover Images: Wiley, © merovingian/Getty Images

Dedicated to the people who built, shipped, and shared with
The AI Makerspace community along the way. Keep going.

—Dr. Greg Loughnane

To my Jess, Odie, Maxi, and Emrys, the very best people in my world.

—Christopher Brousseau

Dedicated to Laura, and all those who are building, shipping, and sharing,
wherever they are.

—C. S. Alexiuk

About the Authors

Dr. Greg Loughnane is the Founder & CEO of AI Makerspace, where he trains engineering and data science teams at mid-market and enterprise companies to build and ship production AI applications. Since 2021, he has developed and taught industry-leading AI & Machine Learning engineering education programs. He loves trail running and is based in Columbus, Ohio.

Christopher Brousseau is a Machine Learning Engineer with a background in linguistics and localization. He specializes in linguistically-informed NLP with a focus on international applications. His expertise has contributed to the success of ML and Data product initiatives in both startup environments and Fortune 500 companies.

C. S. Alexiuk is a Senior Product Research Engineer at NVIDIA on the Nemotron team, where he helps bring open models to developers. He previously co-founded AI Makerspace as CTO, teaching more than 1,000 engineers to build with LLMs. He lives in Toronto.

Acknowledgments

The team at Wiley is the reason this book ultimately got finished. Their patience with the AI Makerspace team and their ability to adapt the plan with calm, measured action through all the trials and tribulations of our young company were a masterclass in book publishing. Thanks, Jim.

Cohort 9 of The AI Engineering Bootcamp, on which this book is primarily based, also included one legendary student, Christopher Brousseau. You may know him as the co-author of *LLMs in Production*. Chris "The Wiz" and I asked him to take the course when we met him at the MLOps World Conference in the fall of 2025. We told him he should get certified with us and that we would do business together when he was done. He got certified, and then we did business together. He also helped put the finishing touches on this book and get it over the finish line.

Finally, to the 500 paying students who helped shape the course and to the thousands in the AI Makerspace community on Discord, YouTube, and LinkedIn who were part of our journey—thank you for building 🏗️, shipping 🚢, and sharing 🚀 with us over the years. As 2026 winds down, we're moving in a different direction to serve more companies and train Forward Deployed Engineers (FDEs) within them. The future is bright, albeit with an altogether different vibe. Thank you to all of our early adopters 🙏. We quite literally would not be here without you.

Contents at a Glance

Preface		xxv
Introduction		xxvii
Chapter 1	Understanding AI Engineering	1
Chapter 2	RAG from Scratch: Embeddings, Vectors, and Retrieval	31
Chapter 3	The Agent Loop	61
Chapter 4	Agentic RAG	95
Chapter 5	Multi-Agent Applications	129
Chapter 6	Agent Memory	171
Chapter 7	Deep Agents	211
Chapter 8	Evaluation Infrastructure	255
Chapter 9	Advanced Retrieval	289
Chapter 10	Full-Stack Applications	323
Appendix A	Setting Up Your Dev Environment	355
Appendix B	Twelve Projects to Build, Ship, and Share	379
Index		387

Contents

Preface xxv

Introduction xxvii

Chapter 1 Understanding AI Engineering 1

What Is AI Engineering? 1
The AI Engineering Stack 2
LLMs and Their Capabilities 3
- Language Models Are Few-Shot Learners 3
- Emergent Capabilities 4
- Limitations and Challenges 4

From Prototyping to Production 5
- The Prototyping Phase 5
- The Productionizing Phase 6
- Key Skills for Each Phase 6

The Role of Agents in Modern AI Systems 7
Key Skills and Tools 8
- Python and Backend Development 8
- LLM Orchestration Frameworks 9
- One Path to RAG: Vector Databases and Search 9
- Evaluation and Monitoring 9

Your First Vibe Check: Getting Started 10
- Evaluating LLM Applications Qualitatively 10
- The AI Engineer Challenge 10

LLM Prototyping Best Practices 11
- Prompt Engineering Basics 11
- Context Engineering Fundamentals 12
- Iterative Development 13
- Quick Feedback Loops 13

Introduction to Evaluation 14
Why Evals Matter from Day One 14
Starting with Vibe Checks 14
Building Toward Systematic Evaluation 15
Building StudyBuddy v1 15
What You're Building 15
Prerequisites 16
Accounts You'll Need 16
Software You'll Need 16
Understanding the Tools 17
Forking and Cloning the Repository 18
Forking the Repository 18
Cloning Your Fork 18
Verifying Your Setup 19
Opening the Project in Cursor 20
Backend Setup 20
Installing Dependencies 20
Setting Your API Key 21
Understanding the Backend Code 21
Modifying the Prompt 23
Starting the Backend Server 23
Testing the Backend 24
Frontend Setup 24
Installing Frontend Dependencies 25
Starting the Frontend Server 25
How the Frontend and Backend Connect 25
Testing the Application 26
Understanding the Frontend Code 26
Deploying to Vercel 27
Installing the Vercel CLI 27
Deploying Your Application 27
Setting Environment Variables in Vercel 28
Getting Your Public URL 28
Deploying Versions to Vercel 28
Nice Work 29

Chapter 2 RAG from Scratch: Embeddings, Vectors, and Retrieval 31
Context Engineering: Why LLMs Need Help 32
The Context Window Problem 32
RAG as a Solution Pattern 33
The LLM Application Stack 34
Introduction to Embeddings 35
What Are Embeddings? 35
Semantic Similarity vs. Keyword Matching 36
Embedding Models 36
How Embeddings Capture Meaning 37
Vector Databases 37
Architecture and Design Principles 38

Popular Vector Database Solutions 38
When You Need a Vector Database 39
Indexing Strategies 39
Brute Force vs. Approximate Search 39
HNSW: Hierarchical Navigable Small World 40
IVF: Inverted File Index 40
Trade-Offs: Speed vs. Accuracy 40
Similarity Search 41
Distance Metrics 41
Cosine Similarity 41
Other Distance Metrics 43
K-Nearest Neighbors 43
Approximate Nearest Neighbors 43
Choosing the Right Metric 44
Chunking Strategies 44
Why Chunking Matters 44
Fixed-Size Chunking 45
Semantic Chunking 45
Overlap and Context Preservation 47
Chunking Best Practices 48
The RAG Pipeline 48
The Indexing Phase 48
The Query Phase 49
Common Pitfalls 51
The 12-Factor Agent Methodology 51
Configuration and Dependencies 52
State Management Patterns 52
Applying 12-Factor to AI Systems 52
Building StudyBuddy v2 53
What We're Adding 53
Architecture 53
The RAG Pipeline 54
Backend Setup 57
Installing Dependencies 57
Running StudyBuddy v2 58
The Frontend: RAG Status 58
Testing the RAG Pipeline 59
Deploying to Vercel 59
What's Next 60

Chapter 3 The Agent Loop 61
Understanding the Agent Pattern 62
What Makes Something an Agent? 62
Agents vs. Workflows 62
The ReAct Pattern 63
When You Need Agents 64
Core Functions of an Agent 64

Perception: Understanding the Current Situation 64
Planning: Deciding What to Do Next 65
Action: Using Tools and Taking Steps 66
Observation: Seeing Results and Adapting 66
The Feedback Loop 67
Planning and Reasoning 67
ReAct Pattern Implementation 67
Reasoning Models 68
Agent Engineering 69
Introduction to LangChain 70
LangChain's Philosophy and Design 70
Component Architecture 70
Why Abstractions Matter (and the Concerns) 71
The create_agent API 72
When to Use Frameworks vs. Custom Builds 73
Tool Use and Function Calling 73
How Tools Work in Agentic Systems 74
Defining Tools in LangChain 74
Tool Selection Mechanisms 76
Parameter Extraction from Natural Language 76
Error Handling in Tool Execution 77
Context Engineering in Practice: Managing Agent State 78
Managing Agent Context vs. RAG Context 78
What Goes in the Context Window for Agents 79
Avoiding Context Overload 79
Working with Qdrant 80
Why We're Switching from a Custom Vector Database 80
Qdrant Architecture and Design 80
In-Memory Mode for Development 81
Server Mode for Production 81
LangChain Integration 82
Creating Collections and Configuring Retrievers 83
Building StudyBuddy v3 84
Where We Left Off 84
What We're Adding 84
Architecture 85
Implementation Walkthrough 85
Exposing Agent Reasoning 90
The Frontend 91
Testing Agentic Behavior 91
Running Locally 92
Deploying to Vercel 93
What's Next 94

Chapter 4 Agentic RAG 95

Beyond Basic RAG 96
Limitations of Static RAG 96
What Makes RAG Agentic 96

When You Need Agentic RAG 97
Looking Under the Hood 97
What *create_agent* Actually Does 98
When to Drop Down to LangGraph 99
Introduction to LangGraph 100
Core Concepts 100
Building Your First Graph 101
The Agentic RAG Architecture 103
State Machine Design 103
Node Definitions 104
Edge Definitions and Routing 105
Assembling the Graph 105
Query Planning and Analysis 106
Understanding User Intent 106
Query Decomposition 106
Complexity Assessment 107
Dynamic Retrieval Strategies 107
Deciding When to Retrieve 107
Deciding What to Retrieve 108
Deciding How Much to Retrieve 108
Multi-Source Retrieval 108
Result Synthesis and Response Generation 109
Combining Multiple Sources 109
Handling Conflicting Information 109
Confidence Scoring 109
Handling Ambiguity and Uncertainty 110
Detecting Ambiguous Queries 110
When to Ask vs. When to Assume 110
Acknowledging Uncertainty 110
Observability and Monitoring with LangSmith 111
Setting Up LangSmith Traces 111
Debugging and Analyzing Traces 112
Using Open-Source Models 112
Building StudyBuddy v4 114
What We're Building 114
The Starting Point 115
The Knowledge Base 116
Node Implementations 117
Routing Functions 122
Building the Graph 123
The API Layer 124
Running Locally 124
Testing the System 125
Viewing Traces in LangSmith 126
Deploying to Vercel 126
What's Next 127

Chapter 5 Multi-Agent Applications 129
When to Use Multiple Agents 129
The Case for Multi-Agent Systems 130
When *Not* to Build Multi-Agent Systems 130
A Decision Framework 131
Context Optimization 131
The Context Rot Problem 131
How Multiple Agents Manage Context 132
Context Isolation Benefits 132
Token Economics 133
Multi-Agent Architectures 133
The Supervisor Pattern 133
The Swarm Pattern 134
Hierarchical Structures 135
The Pipeline Pattern 136
Agent Roles and Specialization 137
Manager Agents 137
Worker Agents 138
Designing Agent Personalities 138
Avoiding Role Confusion 138
Coordination and Communication 139
Message Passing and Handoffs 139
Shared State with LangGraph 140
Agent Node Functions 140
Routing Logic 143
Building and Running the StateGraph 144
Conflict Resolution 145
Error Handling 145
Detecting and Breaking Stuck Loops 146
Multi-Agent Patterns and Use Cases 147
Report Generation Teams 147
Code Generation Workflows 147
Content Creation Workflows 148
Parallel Research Patterns 148
Visualization and Debugging with LangSmith Studio 148
What Is LangSmith Studio v2? 149
Graph Mode vs. Chat Mode 149
Debugging Multi-Agent Interactions 150
Testing and Iteration 150
Building StudyBuddy v5 150
Where We Left Off 150
What We're Adding 151
Defining the Agent Team 151
Creating the Supervisor 156
Shared State 159
The Search Tool 160

Building the Graph 160
Node Implementations 161
Routing Functions 162
Assembling the Graph 163
The FastAPI Backend 165
Testing Multi-Agent Coordination 166
Using LangSmith Studio 167
A Note on the UI 168
Running Locally 168
Deploying to Vercel 169
What's Next 170

Chapter 6 Agent Memory 171
The Memory Problem in AI Agents 172
Why Persistent Memory Matters 172
Memory vs. RAG: Understanding the Difference 173
Memory System Architecture 173
The Memory Store Pattern 175
Giving Agents Memory Tools 176
Namespace Organization 177
Advanced Retrieval Pattern 178
Types of Memory 179
Semantic Memory: Facts and Knowledge 179
Episodic Memory: Past Experiences 180
Procedural Memory: Learned Behaviors 180
When to Use Each Type 181
Memory Formation Strategies 181
Hot Path: Active Memory Formation 181
Background: Passive Memory Extraction 182
Trade-Offs and Decision Framework 184
Hybrid Strategies 184
Memory Storage and Retrieval 184
Storing Memories with Metadata 184
Semantic Search for Relevant Memories 185
Recency and Relevance Weighting 185
Memory Consolidation over Time 186
Memory Integration Patterns 187
Injection at Start 187
On-Demand Retrieval 187
Proactive Memory 188
Adapting Behavior Based on History 189
Moving to PostgreSQL for Production 189
Why PostgreSQL Everywhere 189
Setting Up Local PostgreSQL 190
Database Schema Design 190
Connecting to PostgreSQL 192
Setting Up Vercel Postgres 192

Optimizing Performance Through Caching 193
The Cost of Redundant Generation 193
Content-Addressed Caching Strategies 193
Database-Backed Performance Wins 194
Caching Generated Content 195
Hashing Inputs for Cache Keys 196
Storing Generated Outputs 196
Measuring Performance Improvements 197
When to Cache 197
Building StudyBuddy v6 198
Where We Left Off 198
What We're Adding 198
Persistence Strategy 199
Database Configuration 199
Custom Memory Store 200
Content-Addressed Flashcard Caching 203
Running Locally 207
Deploying to Vercel 208
Testing Persistence 209
What's Next 210

Chapter 7 Deep Agents 211
What Makes an Agent "Deep"? 211
The Four Key Elements of Deep Agents 213
Planning and Task Decomposition 214
Breaking Down Complex Problems 215
Forward Planning vs. Backward Chaining 216
Context Management in Deep Agents 217
Selective Information Retention 218
Context Compression and Summarization 219
When to Summarize vs. Retain Details 220
Subagent Spawning and Delegation 221
Managing Subagent Lifecycles 222
Parallel vs. Sequential Delegation 224
Hybrid Delegation Patterns 225
Error Handling in Delegation 226
Collecting and Integrating Results 227
Multistep Reasoning 227
Sequential Reasoning Chains 227
When to Stop Going Deeper 228
Self-Reflection and Metacognition 230
Evaluating Progress Toward Goals 231
Strategy Adjustment Based on Results 231
The Reflection Loop 232
Backtracking and Recovery 234
Detecting Dead Ends 234

Rolling Back Decisions 235
Exploring Alternative Paths 236
Building StudyBuddy v7 237
Where We Left Off 237
What We're Adding 237
The Curriculum Planner Agent 237
Task Decomposition for Learning Goals 238
Subagent Delegation for Flashcard Generation 240
Progress Checkpoints and Assessments 241
Context Summarization for Long Study Sessions 243
Managing Learning Context Across Sessions 243
Backtracking When Learning Doesn't Work 246
Memory-Connected Learning Modes 247
Struggle Areas Flow to Tutoring 247
Learning Insights Flow to Practice 248
The Frontend Shows Focus Areas 249
Putting It Together 249
Curriculum Study Integration 250
Running Locally 252
Deploying to Vercel 253
What's Next 254

Chapter 8 Evaluation Infrastructure 255
Why Evaluation Matters 255
The Challenge of Creating Test Data 256
Introduction to Synthetic Data 257
The RAGAS Framework 258
The Four Core Metrics 258
Basic RAGAS Evaluation 260
Interpreting Results 260
RAGAS Testset Generation 261
Generating Test Cases 262
Difficulty-Based Generation 262
Question-Type Diversity 263
Adversarial Test Cases 264
Data Quality and Coverage 264
Metrics-Driven Development 266
The Improvement Loop 266
Setting Quality Thresholds 267
Avoiding Goodhart's Law 267
Loading Datasets into LangSmith 268
Creating and Managing Datasets 268
Organizing Datasets 268
Running Evaluations 269
Comparing Experiments 269
Building StudyBuddy v8 270
What We're Adding 270

Setting Up RAGAS for Our Domain 270
Building Evaluation Datasets 272
Custom Evaluators for Educational Content 275
Running Baseline Evaluation 278
Evaluation API Endpoint 280
Evaluation Dashboard 282
Testing the Evaluation System 285
Running Locally 286
Using the Evaluation Infrastructure 286
Step 1: Generate Synthetic Test Data 287
Step 2: Run the Evaluation 287
Step 3: View the Dashboard 287
Viewing the Results in LangSmith 288
Important Notes 288
What's Next 288

Chapter 9 Advanced Retrieval 289
Beyond Basic Vector Search 289
Hybrid Search 291
Reciprocal Rank Fusion 292
Hybrid Search in Practice 293
Reranking 294
Bi-Encoders vs. Cross-Encoders 294
Using Cohere for Reranking 295
When to Rerank 296
Query Expansion 296
LLM-Based Expansion 297
Risks and Alternatives 297
RAG-Fusion 298
Semantic Chunking 300
Embedding-Based Boundary Detection 300
Structure-Aware Chunking 300
Compression and Filtering 301
Contextual Compression 301
Relevance Filtering as a Lightweight Alternative 301
Hierarchical Retrieval 302
Systematic Comparison of Retrievers 303
Controlling Variables 303
Segment Analysis 304
The Decision Framework 304
Building StudyBuddy v9 305
Where We Left Off 305
What We're Adding 305
Semantic Chunking for Reference Materials 306
Hybrid Search Implementation 307
Cohere Reranking Integration 309
RAG-Fusion for Complex Queries 311

The Adaptive Retriever 313
Running Evaluations 315
The Retrieval Comparison Tool 317
Running Locally 320
Using the New Features 320
Chat Normally and Observe the Difference 320
Compare Retrieval Strategies Directly 321
Run the Full Evaluation Pipeline 322
Inspect the Semantic Chunks 322
What's Next 322

Chapter 10 Full-Stack Applications 323
Industry Use Cases 323
Architecture Overview 325
The Frontend Layer 325
The Backend Layer 326
The Agent Layer 326
Bringing It All Together 326
Backend Architecture 327
Frontend Components 329
User Experience Considerations 331
State Management 332
Agent Integration 333
Database Design for Agent Apps 335
Building StudyBuddy v10 336
Learning Program Architecture 337
Document Upload and Indexing 339
Flashcards and Spaced Repetition 343
The Full-Stack Interface 345
Streaming Responses 348
User-Scoped Data Isolation 350
Running the Application 351
Deploying to Vercel 352
What You've Built 354

Appendix A Setting Up Your Dev Environment 355
What We're Building 355
Setting Up Your Code Editor 356
Option A: Cursor (Recommended) 356
Option B: VS Code (Also Great) 356
Essential Extensions 357
Getting to Know Your Terminal 357
What's a Terminal? 358
Getting to Your Terminal 358
Understanding the Command Line 358
Git and GitHub: Version Control 359
Installing Git 359

Configuring Your Git Identity 359
Setting Up GitHub Authentication 360
Creating a Personal Access Token 360
Setting Up SSH Access 361
Basic Git Workflow 366
Essential .gitignore 367
Setting Up Python with uv 367
Installing uv 368
Creating Your First Virtual Environment 368
Creating a Virtual Environment with Python 3.12 368
Activating the Virtual Environment 368
Deactivating When You're Done 369
Installing Essential Packages 369
Configuring Your Editor to Use the Virtual Environment 369
Getting API Keys for LLM Providers 370
OpenAI API Key 370
Anthropic API Key 370
Hugging Face Access Token 371
Storing API Keys Securely 371
Storing Local Environment Variables 371
Storing System Environment Variables 372
Setting Up Docker for Containerization 372
Installing Docker Desktop 373
Verifying Docker Installation 374
Docker Basics 374
Jupyter Notebooks 375
Installing and Running Jupyter 375
JupyterLab (Optional but Recommended) 376
Project Structure—Organizing Your Work 376
Your First Integration Test 377

Appendix B Twelve Projects to Build, Ship, and Share 379
MeetingMinder: Your Intelligent Meeting Summarizer 379
CodeCoach: Your Pair Programming Partner 380
MealMentor: Your Kitchen Companion 381
LitReview Lab: Your Research Literature Assistant 381
CareerNavigator: Your Professional Growth Guide 382
ComplianceGuide: Your Regulatory Intelligence Assistant 382
TravelGuide: Your Trip Planning Assistant 383
CodeStudio: Your API Integration Assistant 383
FinanceHelper: Your Money Management Partner 384
ExperimentDesigner: Your ML Research Planning Assistant 384
GraphBuilder: Your Knowledge Extraction Assistant 385
IdeaBridge: Your Cross-Domain Innovation Assistant 385
Pick Your Path 386

Index 387

Preface

Following ChatGPT's release in November 2022, foundational AI engineering patterns began to emerge. Prompt engineering, retrieval-augmented generation (RAG), and LLM operations (LLMOps) were commonly referenced in public discourse. The question "Should I use RAG or fine-tuning?" pointed us toward the future—toward a bigger picture we couldn't yet see.

I remember arguing subtle distinctions between LLMOps (e.g., operating LLMs in production) and LLM engineering (e.g., pretraining, post-training, and optimizing LLMs before production) with my co-author, Chris "The Wiz" Alexiuk. We even built two courses, one for each discipline. But two courses with nearly the same name confused people. It wasn't yet clear how all the pieces would fit together. Over the course of 2023, however, things started to solidify for the industry. By midsummer, it was clear to me and others that AI engineering had gone mainstream. In January 2024, we went all in on AI engineering to build The AI Engineering Bootcamp. We combined everything we'd done in both of our previous courses.

From January 2024 to August 2026, we ran 10 cohorts of approximately 10 weeks each for 538 students, including 272 (50.56%) who could call themselves Certified AI Engineers through AI Makerspace. During that time, we saw many individual students achieve their goals and dreams in the industry. A young man I met when he was finishing college in Colombia made it to the UK and eventually to Mistral AI. Another student was under 18 when he took our course from Canada; two years later, he had achieved his dream of working at Vercel and was on his way to Silicon Valley. Startups were incubated. Mid-market and enterprise teams were sent to us and went on to have a massive impact at their companies. Retirees got back in the game because they were so curious. Data scientists made the transition to AI engineering. Some who struggled

for years finally cracked the code for breaking into the industry as part of the community we built.

People built 📐, shipped 🚢, and shared 🚀. Like legends, as we liked to say.

Throughout our tenure teaching the course, the curriculum continued to evolve (see the changelog). This book reflects the curriculum from Cohort 9, which we ran from January to March 2026. It was written in partnership with the team at Wiley during that cohort.

Following Cohort 10—v1.0, as we like to call it—the world is once again undergoing its next evolution. It's reminiscent of the end of 2023, as AI engineering was emerging. What's next, in the distance? AI Forward Deployed Engineering (FDE). That's our next chapter. Once you get what you need from this book, we'll see you there.

—Dr. Greg
Owner & CEO
AI Makerspace
August 14, 2026

Introduction

You've played with ChatGPT. Maybe you've built a quick prototype that calls the OpenAI API and does something cool. You've seen the demos, read the tweets, and felt the hype. But there's a massive gap between "I got an LLM to do a thing" and "I built, deployed, and shipped a production AI application that real people actually use." That gap? That's what this book is about.

Over the course of 10 chapters, you're going to learn AI engineering from the ground up. Not just the theory, but the actual patterns, architectures, and production-ready techniques that working AI engineers use every single day. Each chapter teaches you a core set of concepts—retrieval-augmented generation, agent loops, multi-agent systems, memory, evaluation, and deployment—and then you put those concepts to work by building the next version of StudyBuddy, an AI-powered learning assistant that combines conversational tutoring with spaced repetition flashcard practice.

StudyBuddy starts as a simple chatbot in Chapter 1. By Chapter 10, it's a full-stack application. Every version builds on the last, so you're never just reading about ideas in the abstract. You're writing real code, solving real problems, and shipping real features.

The tools you'll work with are the same ones used in many production AI systems today: OpenAI's GPT models and embeddings, LangChain and LangGraph for orchestration, Qdrant for vector search, LangSmith for observability and evaluation, FastAPI for your backend, and Vercel for deployment. By the time you're done, you'll understand how these tools work; you'll know when to use them and when not to; and you'll know how to swap them out when something better comes along.

This is not a theoretical overview. This is a build-it-yourself, ship-it-yourself, own-it-yourself kind of book.

What You Need to Know

This book is for developers who want to become AI engineers. Maybe you're a software developer looking to add AI to your toolkit. Maybe you've come through a coding bootcamp and are ready to specialize. Maybe you've been tinkering with LLMs on your own and you want to go from weekend projects to production systems. Whatever your path, if you're motivated and ready to build, you're in the right place at the right time.

You don't need a PhD in machine learning. You don't need to understand transformer architectures at the matrix multiplication level. But you do need a solid foundation in a few key areas before we jump in:

- **Python proficiency:** You should be comfortable with classes, decorators, `async`/`await`, virtual environments, and package management. We're writing production Python, not beginner scripts.
- **Basic web development:** HTML, CSS, and vanilla JavaScript. You should understand how REST APIs work, what HTTP methods do, and how client-server architecture fits together.
- **Command-line comfort:** You'll live in the terminal. Git, `uv`, `npm`, and environment variables should feel familiar, not frightening.
- **API experience:** You should know how to make HTTP requests, handle JSON responses, and work with API keys. If you've integrated a third-party API before, you're good.
- **An OpenAI API account:** You'll need API access with roughly $50 in credits to work through the exercises. We'll also use the free tiers of other services where possible.
- **Python 3.12+ and Node.js 18+:** Make sure both are installed before Chapter 1. Appendix A has detailed setup instructions.

If you checked most of those boxes, you're ready. If a couple feel shaky, don't sweat it, because you'll pick things up as we go. The important thing is that you're willing to write code, break things, and learn by doing.

Book Roadmap

The book is organized into three parts, each building on the last. Here's what you're going to learn and build across all 10 chapters.

Part I: Foundations (Chapters 1–4)

Part I is where we lay the groundwork. You'll go from zero to a working agentic RAG application, picking up the core patterns and tools that everything else in the book depends on.

Chapter 1: Understanding AI Engineering

We kick things off with the fundamentals of working with large language models. You'll learn LLM prototyping best practices, how adding context to your prompts can dramatically improve output quality, and how to evaluate responses using structured "vibe checks"—a lightweight but surprisingly effective approach to early-stage evaluation. In the StudyBuddy section, you build v1, a simple tutoring chatbot powered by OpenAI's API and deployed to Vercel. It's basic, but it's live. You'll have something running on the Internet by the end of this chapter.

Chapter 2: RAG from Scratch: Embeddings, Vectors, and Retrieval

This is where retrieval-augmented generation enters the picture. You'll learn about embeddings, similarity search, and how to build a RAG pipeline from scratch in Python. No frameworks—just fundamentals. You'll understand why RAG matters, how vector databases work under the hood, and what the LLM application stack looks like. StudyBuddy v2 adds document upload and vector-powered retrieval so the tutor can ground its explanations in your actual study materials.

Chapter 3: The Agent Loop

Now we introduce agents. You'll learn what an "agent" really is (hint: it's simpler than you think), understand the ReAct loop that drives agent decision-making, and get hands-on with LangChain's core constructs for building production agents. StudyBuddy v3 becomes an agentic tutor that decides for itself when to search your materials and when to answer from its own knowledge, powered by LangChain and backed by Qdrant for vector storage.

Chapter 4: Agentic RAG

Here, we go deeper. You'll learn LangGraph for fine-grained orchestration, giving you control over every step of the agent's workflow. You'll add query analysis, dynamic retrieval strategies, reflection, and confidence scoring to your

agent toolkit. You'll also set up LangSmith for observability, because you can't improve what you can't see. StudyBuddy v4 gets a sophisticated RAG pipeline with planning and self-assessment, plus full tracing through LangSmith.

Part II: Advanced Agents (Chapters 5–7)

Part II is where things get serious. You'll move from single-agent systems to multi-agent architectures, add persistent memory, and build agents that can plan and execute over long time horizons.

Chapter 5: Multi-Agent Applications

You'll learn when and why to split work across multiple agents, explore common multi-agent patterns like the supervisor pattern, and understand how to manage context across agent teams. This is also where you'll start using LangSmith Studio to visualize and debug complex agent interactions. StudyBuddy v5 is a big leap: You build a multi-agent system with a Tutor Agent, Card Generator, Quality Checker, and Scheduler, unifying both tutoring and flashcard practice modes under one roof, complete with SM-2 spaced repetition scheduling.

Chapter 6: Agent Memory

Memory is what turns a stateless tool into something that feels intelligent. You'll learn the difference between short-term and long-term memory, explore strategies for writing memories during runtime versus in the background, and understand how memory systems work. StudyBuddy v6 remembers your learning history across sessions: what you've mastered, where you struggle, and how you prefer to learn.

Chapter 7: Deep Agents

This chapter tackles agents that operate over longer time horizons. You'll learn the four key elements of deep agents: planning and task decomposition, context management, subagent spawning, and long-term memory. These are the patterns behind systems like deep research tools and autonomous coding assistants. StudyBuddy v7 becomes a learning planner that can decompose multi-week study schedules, identify prerequisite chains, and spawn subagents to research topics on your behalf.

Part III: Evaluation and Optimization (Chapters 8–10)

Part III is about making your systems measurably better. You'll learn to generate test data, evaluate RAG and agent performance quantitatively, and systematically optimize your retrieval pipeline.

Chapter 8: Evaluation Infrastructure

You can't improve what you can't measure. You'll learn to generate high-quality synthetic test data using RAGAS, build evaluation pipelines in LangSmith, and adopt a metrics-driven development workflow. You'll understand the evaluation metrics that matter for both RAG accuracy and agent behavior. StudyBuddy v8 gets a complete evaluation suite: synthetic test datasets, automated scoring for tutoring quality and flashcard relevance, and baseline performance metrics you can track over time.

Chapter 9: Advanced Retrieval

Now you'll optimize retrieval with techniques like hybrid search (combining dense vectors with BM25), reranking with Cohere, semantic chunking, and RAG-Fusion for comprehensive coverage. More importantly, you'll learn a systematic approach for comparing retrieval strategies so you can make data-driven decisions about what works best for your use case. StudyBuddy v9 upgrades its retrieval pipeline and benchmarks every change against the evaluation infrastructure you built in Chapter 8.

Chapter 10: Full-Stack Applications

Time to give StudyBuddy a real interface. You'll learn end-to-end application architecture for AI-powered systems, including how to connect an agent backend to a modern frontend. StudyBuddy v10 becomes a complete web application with a learning dashboard, flashcard review interface, drag-and-drop material upload, study calendar, and analytics—and it's all mobile-friendly.

That's the journey. Ten chapters, ten versions, one complete application.

This Book's Code

This is a book about building, shipping, and sharing code, so of course there's a ton of code. Good news: you don't have to type everything by hand because it's all available in our GitHub repos:

- **Jupyter notebooks:** Every chapter's code has been added to a Jupyter notebook that you can run. You can fork or download those files here: `https://github.com/AI-Maker-Space/the-ai-engineering-bootcamp-book`
- **StudyBuddy:** Every chapter also features an incremental build of the StudyBuddy app. All the app's code lives here: `https://github.com/AI-Maker-Space/studybuddy`

Who This Book Is For

This book is for anyone looking to go from zero to AI engineering. It will help you understand the core design patterns at play, including prompting, RAG, agents, and evals.

It is great for people who are trying to transition into AI engineering from another discipline or for anyone interested in understanding AI systems that are becoming easier to build every day.

If you are already an advanced practitioner, you may enjoy the return to basics, to a simpler time. However, you may also not enjoy the lack of nuance around emerging techniques. Such is the nature of drawing a line in the sand when defining the scope of the content. Take RAG, for instance. We implement this primarily with an embeddings-based approach in the book. Today, it may be better to use direct agentic search or an LLM-wiki technique for prototyping and save an embeddings-based approach for production environments operating at scale. If this kind of trade-off detail is important to you, please seek out our latest course offerings for individuals and companies at `www.aimakerspace.io`.

CHAPTER 1

Understanding AI Engineering

Here's the move: before you stack agents, before you summon LangGraph, before you declare yourself a full-stack AI engineer, you've got to get your mental models dialed in. This chapter is the onboarding ramp for the entire discipline. Once you internalize this stuff, everything else—retrieval, context engineering, agents, evals—clicks like Lego.

Think of this chapter as the operating system upgrade your brain needs before you launch the apps.

What Is AI Engineering?

AI engineering is the discipline of building applications powered by AI models, especially large language models (LLMs). That sounds simple, but there's a lot packed into that definition. Unlike machine learning (ML) engineers, who focus on training models from scratch or fine-tuning existing ones, AI engineers primarily work with pretrained models accessed through APIs. The models themselves are largely a given; your job is to figure out how to leverage their capabilities to solve real problems.

This distinction matters more than you might think. Traditional ML engineering requires deep expertise in model architectures, training infrastructure, and the mathematics of optimization. AI engineering requires a different skill set. You need to understand how to prompt models effectively, how to orchestrate

complex workflows involving multiple model calls, how to ground model outputs in real data through techniques such as retrieval-augmented generation (RAG), and how to evaluate whether your system is actually working.

AI engineering also differs from traditional software engineering, though it builds heavily on those foundations. In traditional software, you write deterministic code. Given the same input, you get the same output, every single time. LLM-powered applications are fundamentally probabilistic. The same prompt can produce different outputs on different runs. The model might hallucinate facts, get confused by edge cases, or suddenly start performing worse for reasons that aren't immediately obvious. This probabilistic nature requires new patterns for testing, evaluation, and reliability.

The AI engineer role emerged around 2023 as organizations realized they needed people who could bridge these worlds. Companies had access to increasingly powerful models through APIs such as OpenAI's GPT-4 and Anthropic's Claude, but turning those APIs into reliable, production-ready applications required skills that didn't quite fit the traditional ML or software engineering molds. AI engineers fill that gap. They're the people who take a powerful but unpredictable capability and turn it into something you can actually ship to users.

If you have a background in software development, you're already halfway there. The patterns you know—API design, error handling, testing, deployment—all still apply. You just need to adapt them for a world where your core logic is probabilistic rather than deterministic. That's what this book is going to teach you.

The AI Engineering Stack

Every discipline has its *stack*, the layers of technology that work together to enable what you're building. The AI engineering stack has four main layers, and understanding how they fit together will help you make sense of everything else in this book.

At the bottom, you've got *foundation models*. These are the LLMs themselves, including GPT-5, Claude, Gemini, Llama, and others. They're trained on enormous datasets by organizations with massive compute resources, and they provide the core intelligence that powers everything above them. As an AI engineer, you typically don't train these models yourself. You access them through APIs and focus on leveraging their capabilities effectively.

The next layer up is the *API and inference layer* (sometimes called the *Model-as-a-Service*, or *MaaS*, layer). This is where you actually interact with the models. Services such as OpenAI's API, Anthropic's API, Google's Vertex AI, and AWS Bedrock handle the infrastructure of running models at scale and expose them through clean REST APIs. You send in a prompt; you get back a completion. This layer also includes local inference options like Ollama for running open-source models on your own hardware.

Above that sits the *orchestration layer*. This is where things get interesting for AI engineers. Raw API calls are powerful, but real applications typically need more: you need to chain multiple calls together, integrate external data sources, manage conversation history, handle tool use, and implement complex control flows. Frameworks such as LangChain, LangGraph, and LlamaIndex live at this layer. They provide abstractions that make it easier to build sophisticated LLM-powered workflows without reinventing the wheel every time.

Finally, at the top, you've got the *application layer*, which consists of the actual products and features you're building. This might be a chatbot, a document analysis tool, a code assistant, a research agent, or any number of other applications. This layer is where everything comes together: your frontend, your backend APIs, your database, and all the LLM orchestration underneath. It's also where you think about user experience, reliability, and all the other concerns that make software actually useful.

Value can be created at any layer of this stack, but most AI engineers spend their time in the orchestration and application layers. The models themselves are commoditizing, with multiple providers offering competitive capabilities. The differentiation comes from how you use those models: how you prompt them, what data you ground them in, how you orchestrate complex workflows, and how you build great user experiences on top.

LLMs and Their Capabilities

To build effectively with large language models, you need to understand what they can actually do. The capabilities of modern LLMs are genuinely remarkable, but they're also genuinely limited in ways that matter for application development.

Language Models Are Few-Shot Learners

The breakthrough paper that kicked off the current era was OpenAI's 2020 paper on GPT-3, titled *Language Models Are Few-Shot Learners*. The key insight was that very large language models could perform new tasks with just a few examples provided in the prompt, without any fine-tuning or retraining. This capability, called *in-context learning*, changed everything.

Before GPT-3, if you wanted a model to do something new—say, classify customer support tickets into categories—you'd need to collect a training dataset, fine-tune a model, and go through a whole ML development cycle. With in-context learning, you could just show the model a few examples in the prompt, and it would figure out the pattern. This dramatically lowered the barrier to building AI-powered applications.

In-context learning is why prompt engineering matters. The examples, instructions, and context you provide in your prompt are literally teaching the model how to perform your specific task. This is a fundamentally different paradigm from traditional programming, where the computer does exactly what you tell it. With LLMs, you're more like a manager giving instructions to a smart but sometimes unpredictable employee.

Emergent Capabilities

As models have gotten larger and been trained on more data, they've developed capabilities that weren't explicitly programmed or even expected. These are called *emergent capabilities* because they emerge from scale in ways that aren't fully understood.

Chain-of-thought reasoning is a perfect example. If you ask an LLM to solve a multistep problem directly, it often fails. But if you ask it to think through the problem step by step, explaining its reasoning as it goes, accuracy improves dramatically. Nobody programmed this capability! It emerged from training on text that includes humans reasoning through problems.

Other emergent capabilities include code generation, mathematical reasoning, following complex instructions, and what researchers call *theory of mind*—the ability to reason about what other people might be thinking or believing. These capabilities have gotten dramatically better with each generation of models, and there's no clear ceiling in sight.

For AI engineers, emergent capabilities are both exciting and challenging. They're exciting because they expand what's possible. Tasks that seemed impossible two years ago are now achievable with the right prompting. They're challenging because these capabilities can be unpredictable. A technique that works brilliantly on one model might fail completely on another or might stop working when the model gets updated.

Limitations and Challenges

Now for the reality check. LLMs are powerful, but they have significant limitations that you'll need to work around constantly:

- *Hallucinations* are the most famous limitation. LLMs will confidently state things that are completely false. They'll invent citations, make up statistics, and fabricate quotes from real people. This isn't a bug that can be easily fixed; it's a fundamental consequence of how these models work. They're trained to produce plausible-sounding text, not to be factually accurate. Managing hallucinations through techniques such as RAG is a core skill for AI engineers.
- *Context window* limitations constrain how much information you can provide to the model in a single prompt. The *context window* is the total amount

of text that a language model can read and process at one time. But models don't read text character by character. Instead, they process chunks called *tokens*, where a token is roughly three to four characters of English text (about three-quarters of a word). Your entire conversation—including your instructions, any documents you've included, and the model's previous replies—has to fit within the model's context window, measured in tokens. While context windows have grown dramatically—from 4K tokens in early GPT-3.5 to 128K or more tokens in almost all modern models—they're still finite. Long conversations, large documents, or complex applications can easily exceed these limits. You'll learn techniques for managing context effectively throughout this book.

- *Consistency and reliability* remain ongoing challenges. The same prompt can produce different outputs on different runs. Model behavior can change when providers update their models. What worked in testing might fail in production for reasons that are hard to diagnose. Building reliable systems on top of probabilistic models requires new patterns and practices.
- *Latency and cost* matter for real applications. LLM API calls are slow compared to traditional database queries, often taking several seconds for complex requests. They're also expensive, especially for high-volume applications. Optimizing for both latency and cost is a practical concern you'll face on any serious project.

None of these limitations make LLMs unusable—far from it. But they do mean you need to design your applications with these constraints in mind. The best AI engineers are the ones who understand both the capabilities and the limitations and design accordingly.

From Prototyping to Production

One of the unique aspects of AI engineering is how dramatically different the prototyping and production phases feel. In traditional software development, there's usually a fairly smooth continuum from prototype to production. You're writing the same kind of code, just with more tests and better error handling. With LLM applications, the jump from "cool demo" to "reliable production system" is much bigger.

The Prototyping Phase

Prototyping with LLMs is remarkably fast. You can go from idea to working demo in hours or even minutes. Fire up a Jupyter notebook, write a few prompts, call an API, and suddenly you've got something that looks like it works. This

speed is one of the advantages of working with LLMs. The barrier between "I wonder if this would work" and "let me try it" has nearly vanished.

During prototyping, your focus is on exploration and validation. Does this task even work with an LLM? What prompting approaches give the best results? What are the edge cases? You're trying to answer fundamental questions about feasibility before investing in production infrastructure.

The tools of prototyping reflect this exploratory mindset. Jupyter notebooks are perfect for quick iteration. Playground interfaces such as those provided by OpenAI and Anthropic let you experiment with prompts without writing any code. Simple scripts that make API calls and print results are often all you need. The goal is learning, not building.

The Productionizing Phase

Production is a different beast entirely. That prompt that worked great in your notebook? It fails on 15% of real-world inputs. The API call that was fast enough in testing? It times out under load. The output format that looked consistent? It varies in ways that break your downstream processing.

Productionizing LLM applications requires systematic attention to reliability, observability, and evaluation. You need robust error handling for API failures, rate limiting, and malformed responses. You need logging and monitoring to understand how your system is behaving in the wild. You need evaluation frameworks to catch regressions before your users do.

The architecture of production systems is also more complex. Simple request-response patterns give way to asynchronous processing, caching layers, fallback strategies, and sophisticated orchestration. A prototype might make one API call; a production system might make dozens, orchestrated across multiple services with complex retry logic.

Key Skills for Each Phase

Prototyping and production require somewhat different skills, though they obviously overlap.

For prototyping, you need strong prompt engineering skills, especially the ability to quickly iterate on prompts and figure out what works. You need enough coding skill to write quick scripts and process outputs. You need creativity and intuition about what LLMs can do. And you need the judgment to know when you've validated an idea enough to move forward or when it's time to try a different approach.

For production, you need solid software engineering fundamentals: API design, error handling, testing, deployment, monitoring. You need to understand distributed systems concepts because production LLM applications often involve multiple services working together. You need evaluation expertise to build

systems that can catch problems before users do. And you need the patience to chase down the long tail of edge cases that didn't show up in your prototype.

The best AI engineers develop strength in both areas. They can move quickly during prototyping without accumulating technical debt that will slow them down later. They can build production systems that are robust without being over-engineered. This book will help you develop both sets of skills.

The Role of Agents in Modern AI Systems

If there's one concept that represents the frontier of AI engineering, it's agents. *Agents* are LLM-powered systems that can reason about problems, make decisions, use tools, and take actions to accomplish goals. They represent a fundamental shift from LLMs as text generators to LLMs as autonomous problem solvers.

At their simplest, agents combine an LLM with the ability to call external tools. Instead of just generating text, the model can decide to search the web, query a database, execute code, or interact with external APIs. The model reasons about what action to take, executes that action, observes the result, and then reasons about what to do next. This loop of reasoning and action (see Figure 1.1) continues until the task is complete.

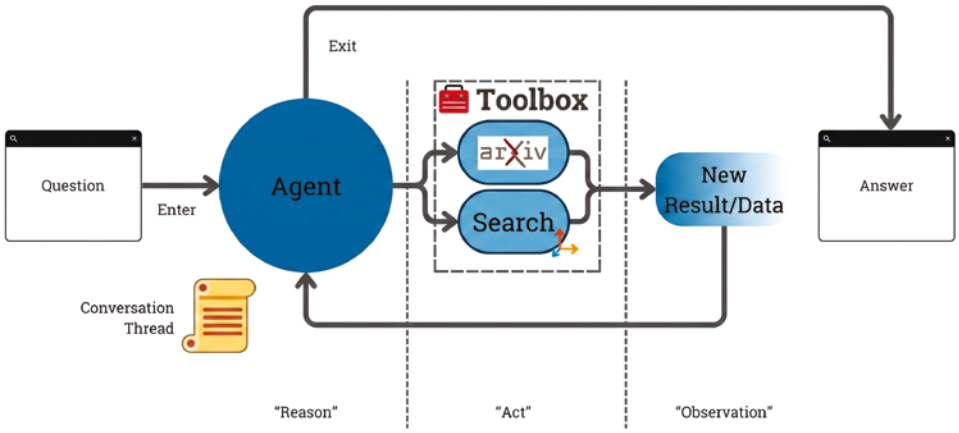

Figure 1.1: The agent loop: the LLM reasons about the task, selects a tool action, observes the result, and repeats—a process known as the ReAct pattern.

The ReAct pattern (reasoning and acting) formalizes this approach. In ReAct, the model alternates between thinking steps—where it reasons about the situation and plans its next action—and action steps, where it actually executes tools. This combination of deliberate reasoning with concrete actions turns out to be remarkably powerful for complex tasks.

More sophisticated agents add additional capabilities. Memory systems let agents learn from past interactions and maintain context across sessions. Multi-agent architectures let two or more specialized agents collaborate on complex tasks. Planning systems let agents break down large goals into manageable subtasks. These advanced patterns unlock applications that would be impossible with simple prompt-and-response workflows.

Consider a few examples of what agents make possible. A research agent can take a question, search multiple sources, synthesize information, identify gaps in its knowledge, search again, and eventually produce a comprehensive answer, all without human intervention. A coding agent can understand a feature request, explore a codebase, write code, run tests, fix bugs, and iterate until the feature works. A customer service agent can understand a complex request, look up account information, check policies, take actions in backend systems, and resolve issues end to end.

Agents also come with new challenges. They can get stuck in loops, make cascading errors, or take unexpected actions. They're harder to evaluate than simple LLM applications because their behavior depends on sequences of decisions rather than single outputs. They require more sophisticated orchestration frameworks and more careful design.

This book dedicates significant attention to agents because they represent where the field is heading. By the end, you'll understand how to build, evaluate, and deploy agents that can handle complex, multistep tasks reliably.

Key Skills and Tools

AI engineering sits at the intersection of several technical disciplines. You don't need to be an expert in all of them from day one, but understanding what skills and tools matter will help you prioritize your learning.

Python and Backend Development

Python dominates AI engineering. Most LLM libraries, orchestration frameworks, and tooling are Python-first. The data science ecosystem—pandas, NumPy, and friends—makes Python natural for working with the kinds of data processing that LLM applications often require. If you're coming from another language, investing in Python fluency will pay dividends quickly.

Beyond Python specifically, solid backend development skills matter. You'll be building APIs, managing state, handling concurrent requests, and integrating with databases and external services. Frameworks such as FastAPI make it straightforward to build production-grade backends for LLM applications. Understanding async programming is particularly valuable because LLM API calls are inherently I/O-bound, so you'll want to execute them concurrently whenever possible.

LLM Orchestration Frameworks

Raw API calls can take you surprisingly far, but complex applications benefit from orchestration frameworks. LangChain provides abstractions for common patterns: prompt templates, output parsers, chains of operations, and integrations with various tools and data sources. LangGraph builds on LangChain to enable stateful, multistep agent workflows with explicit control over the execution graph.

These frameworks aren't without controversy. Some developers feel they add unnecessary abstraction and prefer working with raw APIs. Others find them invaluable for managing complexity. The truth is somewhere in between: frameworks are most valuable when you're building complex, multistep workflows and less necessary for simpler applications. This book will teach you both approaches so you can make informed decisions about when to use what.

One Path to RAG: Vector Databases and Search

RAG is one of the most important patterns in AI engineering. Instead of relying solely on what the model learned during training, you retrieve relevant information at query time and include it in the prompt. This grounds the model's responses in your specific data and dramatically reduces hallucinations.

RAG requires infrastructure for semantic search. Vector databases such as Qdrant, Pinecone, Weaviate, and Chroma store embeddings (essentially, numerical representations of text) and enable fast similarity search. When a user asks a question, you embed their query, find the most similar documents in your vector store, and include those documents in the prompt. Understanding embeddings, chunking strategies, and retrieval optimization is a valuable skill.

Evaluation and Monitoring

You can't improve what you can't measure. Evaluation is arguably the most important skill in AI engineering because it's what separates systems that seem to work from systems that actually work reliably.

LangSmith, Anthropic's developer console, and similar tools provide observability into your LLM applications. You can trace individual requests through complex workflows, see exactly what prompts were sent and what responses came back, and track performance over time. When something goes wrong in production, these tools are invaluable for diagnosis.

Beyond observability, you'll learn to build evaluation datasets, implement automated evaluation metrics, and develop systematic approaches to testing LLM applications. The challenge is that traditional software testing—asserting specific outputs for specific inputs—doesn't translate well to probabilistic systems. You need new frameworks, and we'll develop them throughout this book.

Your First Vibe Check: Getting Started

Let's get practical. Before diving into frameworks and architectures, you need to develop intuition for how LLMs behave. The best way to do that is through what we call *vibe checks*: quick, qualitative evaluations of whether your system is doing what you want.

A vibe check is exactly what it sounds like: you run some examples through your system and see if the outputs feel right. No formal metrics, no automated evaluation—just human judgment about whether things are working. It's informal, it's fast, and it's surprisingly powerful for early-stage development.

The term comes from the broader culture: you vibe check a situation, a person, a party. Does this feel right? Is the energy good? Applied to LLM applications, you're asking: Does this output feel like what I wanted? Is the quality there? Are there obvious problems?

Vibe checks aren't a replacement for rigorous evaluation. You absolutely need systematic, automated evaluation as you move toward production. But vibe checks are the right starting point because they're fast, they build intuition, and they catch obvious problems before you invest in more sophisticated approaches.

Evaluating LLM Applications Qualitatively

Good vibe checks require a bit of structure to be effective. You want to test a range of inputs that represent what real users will send. You want to include edge cases and potential failure modes. And you want to pay attention to dimensions of quality that matter for your specific application.

For a chatbot, you might evaluate tone, helpfulness, accuracy, and whether responses stay on topic. For a summarization system, you'd look at coverage, conciseness, and whether key information is preserved. For a code generation tool, you'd care about correctness, style, and whether the code actually runs. The dimensions depend on the application.

As you vibe check, take notes. What's working well? What's failing? Are there patterns in the failures? These observations will guide your prompt engineering and eventually inform your systematic evaluation framework.

The AI Engineer Challenge

Here's what makes AI engineering genuinely challenging: the feedback loops are long and the signals are noisy. In traditional software, if your code has a bug, it either crashes or produces obviously wrong output. You fix the bug, run

the test again, and move on. With LLMs, the system might produce reasonable-seeming output that's subtly wrong. It might work on your test cases and fail on real-world inputs. It might even work perfectly today and fail tomorrow because the model got updated.

This uncertainty is why evaluation matters so much and why you need to develop strong intuition through practice. The more you work with LLMs, the better you get at anticipating problems, recognizing patterns, and knowing when something seems off. Vibe checks are the beginning of building that intuition.

LLM Prototyping Best Practices

With the fundamentals covered, let's talk about how to actually build stuff. Prototyping well is a skill, and there are patterns that will make you more effective.

Prompt Engineering Basics

Prompt engineering is the art of communicating effectively with language models. It sounds simple—just write clear instructions—but there's real depth here. Small changes in wording can produce dramatically different results. The best prompts aren't always the most obvious ones:

- **Start strong.** The *instructions* you pass to the model initially—also widely called the *system prompt*—are where you define the model's persona, constraints, and behavior. These instructions point the model in the direction you want it to go.
- **Be clear.** Be specific about what you want. Instead of "summarize this text," try "summarize this text in 2–3 sentences, focusing on the main argument and key supporting evidence." Instead of "write some code to find the longest string," try "write a Python function that takes a list of strings and returns the longest one, handling the case where the list is empty." The more specific you are, the better your results.
- **Use examples when appropriate.** Few-shot prompting—showing the model examples of input-output pairs before your actual input—is remarkably effective for many tasks. The examples teach the model what you want better than instructions alone. For complex formats or subtle distinctions, examples are often essential.
- **Instruct the model to think step by step for complex reasoning tasks.** The simple phrase "Let's think through this step by step" can dramatically improve accuracy on math problems, logic puzzles, and multistep

analysis. This technique, called *chain-of-thought prompting*, gives the model space to reason rather than forcing it to produce an immediate answer.

- **Pay attention to formatting.** The structure of your prompt matters: where you put instructions, how you delimit sections, what format you request for outputs. Clear structure helps the model parse your intent. Consistent formatting across examples helps the model learn the pattern.

Context Engineering Fundamentals

If prompt engineering is about crafting effective instructions for LLMs, context engineering is about controlling what information the model has access to. The two are closely related but conceptually distinct.

When you send a prompt to an LLM, you're sending not only a question or instruction but also context that shapes how the model responds (see Figure 1.2). That context might include system instructions that define the model's persona and constraints, examples that demonstrate the desired behavior, retrieved documents that provide relevant information, conversation history that maintains continuity, and any other information that helps the model understand what you need.

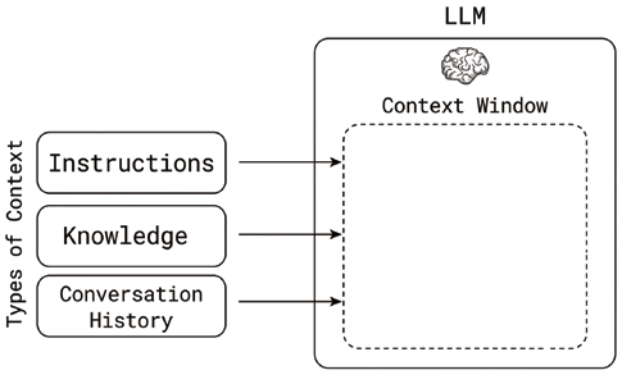

Figure 1.2: The context window includes system instructions, documents and other relevant knowledge, and conversation history.

Context engineering is about making deliberate decisions about what goes into that context window. What information does the model need? In what order should it appear? How do you handle cases where there's more relevant information than can fit in the context window? These questions get increasingly important as you build more sophisticated applications.

Think of context as the model's working memory. Everything the model knows for a given request is either in its training data or in the context you

provide. Since you can't control the training data, context is your primary lever for controlling behavior. Getting good at context engineering is a core skill for any AI engineer.

Iterative Development

LLM development is fundamentally iterative. You try something, see how it works, adjust, and try again. This is different from traditional software, where you can often think through a solution completely before implementing it. With LLMs, you have to experiment:

- **Start simple.** Begin with the most straightforward prompt you can think of and see what happens. Often it works better than you'd expect, and you've saved yourself the effort of over-engineering. If it doesn't work, you have a baseline to improve from.
- **Change one thing at a time.** When a prompt isn't working, it's tempting to rewrite the whole thing. Resist that urge. Change one aspect, see if it helps, and build understanding incrementally. If you change five things and it gets better, you don't know which change helped.
- **Keep track of what you've tried.** It's remarkably easy to go in circles, trying the same approaches repeatedly without realizing it. Keep notes, save prompt versions, and document what worked and what didn't. Your future self will thank you.

Quick Feedback Loops

Speed of iteration is everything during prototyping. The faster you can try something and see results, the more you'll learn and the better your eventual solution will be.

- **Use playgrounds and notebooks for exploration.** The web interfaces provided by OpenAI, Anthropic, and others let you try prompts instantly without any setup. Jupyter notebooks let you write quick code, see results, and iterate without the overhead of full application development.
- **Build small test sets early.** Even during prototyping, having a handful of representative inputs that you can quickly run through your prompt is valuable. It keeps you honest about whether changes are actually improvements or just trading one problem for another.
- **Automate the boring parts.** If you find yourself copying and pasting the same text repeatedly, write a quick script. If you're manually comparing outputs, build a simple comparison tool. Any friction in your iteration loop compounds over time.

Introduction to Evaluation

We'll end where AI engineering actually begins: with evaluation. If there's one thing that separates successful LLM applications from failed ones, it's the quality of their evaluation practices. Getting this right from the start will save you enormous pain later.

Why Evals Matter from Day One

Here's a scenario that plays out constantly: a team builds an LLM feature, tests it manually, ships it, and then gradually realizes it's not working as well as they thought. By that point, they've got users depending on the feature, no systematic way to measure quality, and no baseline to compare improvements against. Fixing the problem is much harder than it would have been if they'd built evaluation in from the start.

Evaluation serves multiple purposes. It catches *regressions*, changes that accidentally make things worse. It enables confident iteration because you can try new approaches knowing you'll detect problems. It provides evidence for stakeholders, since "trust me, it works" is not as compelling as "here are the metrics." And it builds institutional knowledge about what quality means for your specific application.

The investment pays dividends throughout the project lifecycle. Teams with good evaluation practices move faster because they can make changes confidently. They ship better products because they catch problems before users do. And they make better decisions because they have data rather than intuition.

Starting with Vibe Checks

Vibe checks, which we introduced earlier, are the right starting point for evaluation. They're fast to implement since you're just running examples and looking at outputs. They require no infrastructure; a notebook is all you need. And they force you to think about what quality means for your application, which is valuable even if you later automate that judgment.

As your project matures, you'll want to formalize vibe checks into something more systematic. The examples you've been testing manually become a formal test set. The quality dimensions you've been evaluating informally become explicit criteria. The intuitions you've developed become documentation that others can learn from.

But don't skip the vibe check phase. The understanding you build by manually evaluating outputs is essential for designing good automated evaluations later. If you don't know what good looks like, you can't teach a system to measure it.

Building Toward Systematic Evaluation

Systematic evaluation involves test datasets, automated metrics, and regular evaluation runs. It's more work to set up, but it scales in ways that manual evaluation can't.

Test datasets are curated collections of inputs with known good outputs or at least clear quality criteria. They should be representative of real usage, include edge cases and potential failure modes, and be large enough to give statistical confidence. Building good test datasets is an ongoing investment because you'll add to them as you discover new failure modes.

Automated metrics quantify aspects of quality. Some are simple, like "did the output match the expected format?" Others use LLMs to evaluate LLM outputs, which sounds circular but works surprisingly well in practice. The challenge is choosing metrics that actually correlate with what you care about. A metric that's easy to compute but doesn't reflect real quality is worse than useless because it gives false confidence.

Evaluation infrastructure ties everything together. You need ways to run evaluations regularly, track metrics over time, compare different versions of your system, and investigate specific failures. Tools like LangSmith provide much of this infrastructure out of the box.

The path from vibe checks to systematic evaluation isn't a one-time journey; it's an ongoing process. As your application evolves, your evaluation needs to evolve with it. New features require new tests. New failure modes require new metrics. The best teams treat evaluation as a core part of their development process, not an afterthought.

Building StudyBuddy v1

Welcome to the hands-on part of your AI engineering journey. This section walks you through building the first version of the StudyBuddy tutoring assistant, so it's your first opportunity to build and deploy an LLM-powered application from scratch. By the end of this guide, you'll have a working app live on the Internet. Not a toy. Not a demo. A real, deployed application that anyone can use.

What You're Building

We offer the AI Engineer Challenge, which is designed to get you from zero to deployed as quickly as possible. You're going to:

1. Fork and clone a starter repository.
2. Set up a Python backend powered by FastAPI and the OpenAI API.

3. Use "vibe coding" techniques in Cursor to generate a Next.js frontend.
4. Deploy the whole thing to Vercel.

The goal here isn't perfection—it's momentum. You're building the muscle memory of shipping AI applications. Every single concept we cover in this book builds on this foundation.

Prerequisites

Before you dive in, there are a few things you need to make sure you have ready. The next few sections take you through what you need to set up. Note that you'll already have most of what follows if you've already gone through Appendix A.

Accounts You'll Need

You need the following accounts:

- **GitHub account:** If you don't have one yet, head to `github.com` and sign up. GitHub is where your code lives, where you'll fork the starter project, and where Vercel pulls your code from for deployment. It's free, and you'll use it constantly as an AI engineer.
- **OpenAI API account:** You need API access to OpenAI's models. Refer to Appendix A to learn how to get and store an OpenAI API key. You'll also want to add some credits to your account. For the exercises in this book, $50 worth of credits will last you a good while.
- **Vercel account:** Vercel is where you'll deploy your application. The easiest path is to sign up using your GitHub account, which will make the deployment process seamless since Vercel can automatically access your repositories.

Software You'll Need

Install the following software if you haven't already:

- **Git:** Git is your version control system. It tracks changes to your code and lets you collaborate with others. Refer to Appendix A for the installation instructions.
- **A code editor (Cursor recommended):** While you can use any editor, this guide assumes you're using Cursor. Cursor is a fork of VS Code with AI capabilities baked in. It's specifically designed for the kind of AI-assisted development we're doing here. You can also use VS Code with the Claude Code or GitHub Copilot extension installed. Refer to Appendix A for installation and setup instructions for both editors.

- **Terminal access:** On Mac or Linux, you've got Terminal built in. On Windows, you can use Command Prompt, PowerShell, or Windows Terminal. If you're on Windows and want a more Unix-like experience, consider installing Windows Subsystem for Linux (WSL).
- **Node.js and npm:** Node.js is a JavaScript runtime, and npm is its package manager. You'll need these for the Next.js frontend and for deploying to Vercel. Download from `nodejs.org`, where the LTS (long-term support) version is your best bet.

Understanding the Tools

Before we jump into the steps, let's take a moment to understand what each tool does. This context will make everything click better as you work through the setup.

- `uv`: The modern Python package manager. `uv` is a relatively new Python package manager that's blazingly fast. Think of it as a replacement for `pip` that also handles Python version management. Here's what makes it special:
  - It automatically downloads and manages Python versions for you.
  - It creates isolated virtual environments so your projects don't conflict.
  - It's significantly faster than `pip` for installing packages.
  - It uses a `pyproject.toml` file to track dependencies.

  When you run `uv sync`, it reads the project's dependencies and installs everything you need in one shot. No more manually creating virtual environments or worrying about Python versions.
- **FastAPI:** A modern Python web framework for building APIs. It's fast (it's right there in the name), easy to use, and generates automatic documentation for your endpoints. The starter project includes a FastAPI backend that handles communication between your frontend and the OpenAI API.
- **Next.js:** A React framework that makes building web applications easier. It handles routing, server-side rendering, and optimization out of the box. We use Next.js for the frontend because:
  - It has a great developer experience with hot reloading.
  - It works seamlessly with Vercel for deployment.
  - It uses React, which is the most popular frontend library.
- **Cursor:** This editor looks and feels like VS Code, but it has AI capabilities built directly into the editor. You can chat with an AI assistant about your code, ask it to generate new code, or have it help you debug issues.

The "vibe coding" approach we use here leverages Cursor's ability to generate entire frontend applications based on natural language descriptions.

- **Vercel:** A platform for deploying web applications. It's particularly good at deploying JavaScript frontends, but it also supports Python backends through serverless functions. When you push code to GitHub, Vercel can automatically deploy your changes—a technique called *continuous deployment*.

Forking and Cloning the Repository

All right, let's start building. First, you need to get the starter code onto your machine.

Forking the Repository

In GitHub, you can create a copy of a repository under your own GitHub account by *forking* that repo. Why fork instead of just clone? Because you'll want to push your changes back to GitHub, and you can't push to someone else's repository. Your fork is your own copy that you control.

For this project, you begin by forking the AI Engineer Challenge repository:

1. Open your browser and navigate to the AI Engineer Challenge repository on GitHub: `https://github.com/AI-Maker-Space/the-ai-engineering-bootcamp-book`
2. Click the Fork button. The Create a New Fork page appears.
3. Edit the repo name, if you feel like it. Note: You'll be using this forked repo as the basis of the LLM app you build in this book, so changing the repo name to one that matches the name of your project is recommended. If you're not sure about your project yet, use something like **my-llm-app**.
4. Click Create Fork.

When the forking is complete, you end up back at your GitHub account with the forked repo now listed among your own repos.

Note that the following code examples reference the StudyBuddy app that we build in this book, so if you want to follow along, you can access the repo directly at `https://github.com/AI-Maker-Space/studybuddy`

Cloning Your Fork

That forked repo is yours to mess around with as needed. To get started, you need to make a local copy on your machine, which you do by *cloning* the forked repo.

How you clone depends on whether you're using HTTPS or SSH for authentication (refer to Appendix A):

- **HTTPS:** In your forked repo on GitHub, click Code, make sure the HTTPS tab is displayed, then click the Copy button to the right of the URL. The URL should look similar to this (where `USERNAME` is your GitHub username and `REPO-NAME` is the name you gave to the fork):

  ```
  https://github.com/USERNAME/REPO-NAME.git
  ```

- **SSH:** In your forked repo on GitHub, click Code, click the SSH tab, then click the Copy button to the right of the address, which should be similar to this:

  ```
  git@github.com:USERNAME/REPO-NAME.git
  ```

Open your terminal and navigate to where you want the project to live. When you clone the repo, `git` automatically creates a subdirectory with the repo name, so you're navigating here to what will be the clone's parent folder. For example, if all your projects will live in subdirectories of the `~/projects` folder, then you'd navigate to that folder as follows:

```
cd ~/projects
```

Now type **git clone**, type a space, paste the copied URL or address, and then press Enter or Return. If you're using HTTPS, you might be prompted for your GitHub username and a personal access token, unless you stored your credentials as we told you in Appendix A.

Git connects to GitHub, then clones the forked repository into a new subdirectory with the same name as the forked repo.

Verifying Your Setup

Let's make sure everything is connected properly. Assuming your forked repo uses the default name `The-AI-Engineer-Challenge`, run the following commands:

```
cd The-AI-Engineer-Challenge
git remote -v
```

You should see output showing your fork as the origin remote:

```
origin git@github.com:USERNAME/The-AI-Engineer-Challenge.git (fetch)
origin git@github.com:USERNAME/The-AI-Engineer-Challenge.git (push)
```

This is where your code will be pushed when you make changes.

Opening the Project in Cursor

Now you can open the project in Cursor:

```
cursor .
```

If the `cursor` command isn't recognized, you may need to install the Cursor command-line tools. Open Cursor manually, go to the Command Palette (Cmd+Shift+P on Mac, Ctrl+Shift+P on Windows/Linux), and search for "Shell Command: Install 'cursor' command."

Backend Setup

The backend is where the AI magic happens. It's a FastAPI application that connects to OpenAI's API and processes requests from your frontend.

Installing Dependencies

The dependencies are specified in the `pyproject.toml` file. Your first task is to open the file and update the `dependencies` section to the following:

```
dependencies = [
    "fastapi>=0.115.0",
    "jupyter>=1.1.1",
    "uvicorn[standard]>=0.32.0",
    "openai>=1.54.0",
    "python-dotenv>=1.0.0",
]
```

It's particularly important to add the `python-dotenv` library, because you need it to read the environment variables where things like your OpenAI API key are stored.

Note, however, that Vercel doesn't use the `pyproject.toml` file for deployment. Instead, Vercel gets the project dependencies from a different file named `requirements.txt`, which must reside in the backend directory. Create a new text file named `requirements.txt` in the project's `api` directory, then add the following dependencies:

```
fastapi>=0.115.0
uvicorn[standard]>=0.32.0
openai>=1.54.0
python-dotenv>=1.0.0
```

With that done, from the project root directory, run:

```
uv sync
```

This command does several things:

- It checks if you have the required Python version (3.12).
- It downloads Python 3.12 if you don't have it.
- It creates a virtual environment in the `.venv` folder.
- It installs all the project dependencies.

You'll see output showing the packages being installed. This might take a minute the first time.

Setting Your API Key

The backend needs your OpenAI API key to communicate with OpenAI's models. Set it as an environment variable:

On Mac/Linux:

```
export OPENAI_API_KEY=sk-your-actual-api-key-here
```

On Windows (Command Prompt):

```
set OPENAI_API_KEY=sk-your-actual-api-key-here
```

On Windows (PowerShell):

```
$env:OPENAI_API_KEY="sk-your-actual-api-key-here"
```

Important: This sets the environment variable for your current terminal session only. When you close the terminal, it's gone. For a more permanent solution, you can add the `export` command to your shell's configuration file, as we describe in Appendix A.

Understanding the Backend Code

The backend code resides in the file `v1-basic-chatbot/api/index.py`:

```
from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from openai import OpenAI
from pathlib import Path
import os
from dotenv import load_dotenv

# Load .env from parent directory (v1-basic-chatbot/)
env_path = Path(__file__).parent.parent / ".env"
load_dotenv(env_path)
```

```
# Validate API key at startup
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
if not OPENAI_API_KEY:
    raise ValueError("OPENAI_API_KEY environment variable is not set")

app = FastAPI()

# CORS so the frontend can talk to backend
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"]
)

client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

class ChatRequest(BaseModel):
    message: str

@app.post("/api/chat")
def chat(request: ChatRequest):
    try:
        user_message = request.message
        response = client.responses.create(
            model="gpt-4o-mini",
            instructions=" You are StudyBuddy, a helpful AI tutoring
assistant...",
            input=user_message
        )
        return {"reply": response.output_text}

    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error calling
OpenAI API: {str(e)}")
```

Here's a quick run-through of what this code does:

- It sets up a FastAPI server with CORS enabled for frontend communication.
- It loads environment variables (including `OPENAI_API_KEY`) from the `.env` file.
- It initializes an OpenAI client using the API key.
- It defines a `ChatRequest` model that expects a message string.
- It exposes a `POST` endpoint at `/api/chat` that:
  - Accepts a chat message in the request body
  - Validates that the OpenAI API key is configured

- Sends the message to OpenAI's API using the `gpt-4o-mini` model
- Returns the AI's response as JSON
- Handles errors and returns appropriate HTTP status codes

Modifying the Prompt

In `v1-basic-chatbot/api/index.py`, the `create()` request includes an `instructions` string, which is your prompt to the model. The default prompt from the forked repo is `You are a supportive mental coach`. You need to change that to a prompt that fits with the LLM app you want to create.

We'll be creating a state-of-the-art tutoring assistant as our example app, so here are our starting instructions (the system prompt):

```
You are StudyBuddy, a helpful AI tutoring assistant. Your job is to help
students learn by:

- Explaining concepts clearly and at the right level for the student
- Breaking down complex ideas into simpler pieces
- Providing examples to illustrate your explanations
- Encouraging questions and curiosity
- Being patient and supportive

When a student asks you something:
1. First, understand what they're trying to learn
2. Explain the concept in clear, simple language
3. Use concrete examples when possible
4. Check if they understood by asking if they need clarification

Keep your explanations concise but thorough. If a concept is
complicated, break it into smaller parts. Always be encouraging and make
learning feel approachable, not intimidating.
```

Not sure what to build yet? Be sure to check out Appendix B, "Twelve Projects to Build, Ship, and Share," for inspiration and initial prompts.

Starting the Backend Server

Now fire up the backend server:

```
cd v1-basic-chatbot
uv run uvicorn api.index:app --reload --port 8000
```

Let's break down what the second command does:

- `uv run` executes a command within the virtual environment that `uv` created.
- `uvicorn` is an Asynchronous Server Gateway Interface (ASGI) server that runs your FastAPI application.

- `api.index:app` tells `uvicorn` to look in the `api` folder for a file called `index.py`, then within that file look for a variable named `app`. That's the actual FastAPI application object.
- `--reload` enables auto-reload, so the server restarts when you change code.
- `--port 8000` tells the server to listen for requests on port 8000.

You should see output indicating the server is running at `http://127.0.0.1:8000`.

Leave this terminal window running. The server needs to stay active while you work on the frontend.

Testing the Backend

Before you start vibe-coding the frontend in the next section, it's a good idea to run a test to make sure the backend is working properly. The backend's `api/chat` endpoint is a `POST` endpoint, so you can't test it directly in a web browser (which only sends `GET` requests). At the terminal, run the following `curl` command instead:

```
curl -X POST http://localhost:8000/api/chat \
  -H "Content-Type: application/json" \
  -d '{"message": "Hello"}'
```

If all is well, you'll get back a response that looks something like this:

```
{"reply":"Hi there! I'm StudyBuddy. What would you like help with today?
Tell
me your subject and topic, your grade level, and whether you want an
explanation, practice problems, or a quick review. If you're not sure,
I can
suggest a few options and we can dive in. What area should we start
with?"}
```

We'll be talking a lot about prompting in this book, so don't sweat your prompt too much at this early stage. It's good enough just to point the model in the general direction you want it to go.

Frontend Setup

The frontend will provide the chat interface that users interact with. We'll build it using Next.js and React.

Installing Frontend Dependencies

Keep the backend running, open a new terminal window, and navigate to the frontend directory:

```
cd v1-basic-chatbot/frontend
npm install
```

This installs all the Node.js dependencies specified in `package.json`, including Next.js, React, and react-markdown for rendering formatted AI responses.

Starting the Frontend Server

Start the Next.js development server:

```
npm run dev
```

You should see output indicating the server is running at `http://localhost:3000`.

How the Frontend and Backend Connect

During local development, you'll have two servers running:

- **Backend (FastAPI):** `http://localhost:8000`
- **Frontend (Next.js):** `http://localhost:3000`

The Next.js configuration in `next.config.ts` includes a proxy that forwards `/api/*` requests from the frontend to the backend:

```
async rewrites() {
  return [
    {
      source: '/api/:path*',
      destination: 'http://localhost:8000/api/:path*',
    },
  ]
}
```

This means when the frontend makes a request to `/api/chat`, it automatically gets forwarded to `http://localhost:8000/api/chat`. This proxy only applies during local development. In production on Vercel, the routing is handled differently.

Testing the Application

Open `http://localhost:3000` in your browser. You should see the StudyBuddy chat interface. Try asking it something like:

- "Explain the causes of World War I."
- "What's the difference between mitosis and meiosis?"
- "Help me understand how compound interest works."

Understanding the Frontend Code

The frontend uses React components organized in the `src/` directory. Here's the project structure:

```
frontend/
├── src/
│   ├── app/
│   │   ├── globals.css   # Styles and animations
│   │   ├── layout.tsx    # Root layout with HTML structure
│   │   └── page.tsx      # Main chat page with state management
│   └── components/
│       ├── LoadingDots.tsx   # Loading animation
│       ├── Message.tsx       # Individual message bubble
│       ├── MessageInput.tsx  # Input textarea + send button
│       └── MessageList.tsx   # Scrollable message container
├── next.config.ts        # Next.js configuration (API proxy)
├── package.json          # Dependencies
└── tsconfig.json         # TypeScript configuration
```

Here's a rundown of the key components:

- **`page.tsx`:** The main chat page:
  - It manages the messages state (array of user and assistant messages).
  - It handles sending messages to the API.
  - It shows loading states and error messages.
- **`Message.tsx`:** Renders individual message bubbles:
  - User messages appear on the right with a blue background.
  - Assistant messages appear on the left with a gray background.
  - The component uses react-markdown to render formatted responses.
- **`MessageInput.tsx`:** The input area:
  - The component includes an auto-resizing textarea.

- It includes a send button.
- It supports keyboard shortcuts (Enter to send, Shift+Enter for a new line).

Deploying to Vercel

Once you've got a working app on your local machine, it's time to share it with the world.

Installing the Vercel CLI

Open a new terminal (keep your frontend running) and install the Vercel command-line tool:

```
npm install -g vercel
```

The `-g` flag installs it globally, making the `vercel` command available from anywhere.

Deploying Your Application

Navigate to your project root and run:

```
vercel
```

When Vercel asks if you want to deploy your app, type **Y** and press Enter or Return. The first time you run this, you'll be guided through setup:

1. You'll be prompted to log in to Vercel. Use the GitHub option for the smoothest experience.
2. Vercel will ask if you want to link to an existing project or create a new one. Choose to create a new one.
3. Configure your project settings. You'll be asked about:
   - Which directory contains your code (usually the current directory).
   - Build settings (Vercel usually auto-detects these).
   - Whether you want to link to your GitHub repo. Vercel will link to the original repo from which your project was forked, so say No here. After the deployment, run `vercel git connect` and then be sure to choose your own repo in the list.

Vercel builds and deploys your application.

Setting Environment Variables in Vercel

Your deployed app needs the OpenAI API key. Send a web browser to `vercel.com` to display your Vercel dashboard, then follow these steps:

1. Click your project to open it.
2. Click Settings.
3. Click Environment Variables and make sure the Create new tab is displayed.
4. In the Key text box, type **OPENAI_API_KEY**.
5. In the Value text box, paste your API key.
6. Click Save.
7. When Vercel displays a message that a redeployment is needed for the changes to take effect, click Redeploy to open the Redeploy dialog, then click Redeploy.

Getting Your Public URL

Once deployment completes, Vercel gives you a URL. It'll look something like `your-project-name.something.vercel.app`. This is your live application.

Test it in an incognito browser window to make sure it works for people who aren't logged into your accounts.

Deploying Versions to Vercel

In order to deploy each of the fourteen versions of StudyBuddy (one for each chapter of the book), you need to tell Vercel what you want to deploy. To do that, you need to create two files in the repo root.

First, add `vercel.json` to point to the `v1-basic-chatbot` directory:

```
{
  "version": 2,
  "builds": [
    {
      "src": " v1-basic-chatbot/api/index.py",
      "use": "@vercel/python"
    },
    {
      "src": "v1-basic-chatbot/frontend/package.json",
      "use": "@vercel/next"
    }
  ],
  "routes": [
    { "src": "/api/(.*)", "dest": "/v1-basic-chatbot/api/index.py" },
    { "src": "/(.*)", "dest": "/v1-basic-chatbot/frontend/$1" }
```

```
  ],
  "git": {
    "deploymentEnabled": {
      "main": true
    }
  },
  "ignoreCommand": "bash scripts/should-deploy.sh"
}
```

Second, create a `scripts` directory and add `should-deploy.sh` with the following script to trigger on v1 changes:

```
#!/bin/bash
# Deploy if changes are in v1-basic-chatbot or vercel config
# Exit 1 = proceed with build, Exit 0 = skip build
git diff HEAD^ HEAD --name-only | grep -qE "^v1-basic-chatbot/|^vercel\.
json" && exit 1 || exit 0
```

With these files in place, you can deploy by simply pushing to GitHub. Vercel will automatically deploy when changes are detected in the `v1-basic-chatbot` directory (or in `vercel.json`).

Nice Work

You built and deployed an LLM-powered application. That's a real accomplishment. This is the first of many applications you'll build. The skills you just used—setting up development environments, working with APIs, using AI-assisted development, deploying to production—are the foundations of AI engineering.

Let's keep building.

CHAPTER

2

RAG from Scratch: Embeddings, Vectors, and Retrieval

Here's the situation. You built a chatbot in Chapter 1. It can have conversations; it can answer questions; it can be helpful. But there's a fundamental problem: it only knows what was in its training data. Ask it about your company's internal documentation? It has no idea. Ask it to help a student learn from their specific textbook? Can't do it. The model's knowledge is frozen at the moment it was trained.

This is the context problem, and it's the first major challenge every AI engineer faces. You can't just keep retraining models with new information because that's slow, expensive, and totally impractical for most applications. You need a way to give your LLM access to specific, relevant information at query time.

That's where RAG comes in. Retrieval-augmented generation. The name's a mouthful, but the concept is straightforward: when a user asks a question, you retrieve relevant information from your documents and include it in the prompt you send to the model. The model then generates a response based on both its training and the specific context you provided.

RAG is everywhere in production AI systems. Customer support bots that answer questions about your product documentation? RAG. Research assistants that help analysts explore proprietary data? RAG. Legal tools that search through case law? RAG. Internal knowledge bases that let employees find company information? All RAG.

In this chapter, you're going to learn how RAG works from first principles. We're not just going to use a library and call it a day. We're going to build the

entire pipeline from scratch so you understand exactly what's happening under the hood: how documents get turned into vectors, how similarity search actually works, how chunking affects quality—all of it.

Then we'll take StudyBuddy from v1—that basic chatbot you deployed in Chapter 1—and transform it into v2, a RAG-powered tutoring system that can actually teach from uploaded study materials. Upload a textbook chapter, ask a question about it, and get an explanation grounded in that specific content.

By the end of this chapter, you'll understand embeddings, vector databases, similarity search, chunking strategies, and the complete RAG pipeline. You'll know when RAG makes sense and when it doesn't. You'll have built a working system that demonstrates these concepts in action.

Context Engineering: Why LLMs Need Help

Before we dive into the technical details of how RAG works, let's talk about why it exists. The problem RAG solves is fundamental to how LLMs work, and understanding that problem is crucial to building effective systems.

The Context Window Problem

Every LLM has a context window (see Figure 2.1), which is the maximum amount of text it can process at once. For GPT-5, that's 128K tokens. For Claude Opus, it's 200K tokens. These sound like big numbers, and they are, but they fill up fast.

As you learned in Chapter 1, tokens are the units LLMs use to process text. They encode whole words and subwords such as prefixes, suffixes, and grammatical markers. In practice, token counts run about a third higher than word counts, so 128K tokens is about 96,000 words, which is maybe 300 pages of text. That's one decent-sized book. What if you're building a research assistant that needs to search across thousands of documents? What if you're building a customer support bot that needs to access your entire documentation library? You can't fit all of that into the context window.

Even if you technically could fit everything, you shouldn't. More context means slower processing, higher costs, and worse performance. Models perform best when they're working with focused, relevant information. Give them too much context and they start to lose track of what matters. This is called the *needle-in-the-haystack problem*: the model has a harder time finding the relevant information when it's buried in a mountain of irrelevant text.

So, you need a way to be selective. You need to retrieve only the most relevant information for each query, include just that in the context, and let the model work with a manageable amount of focused information. That's what RAG does.

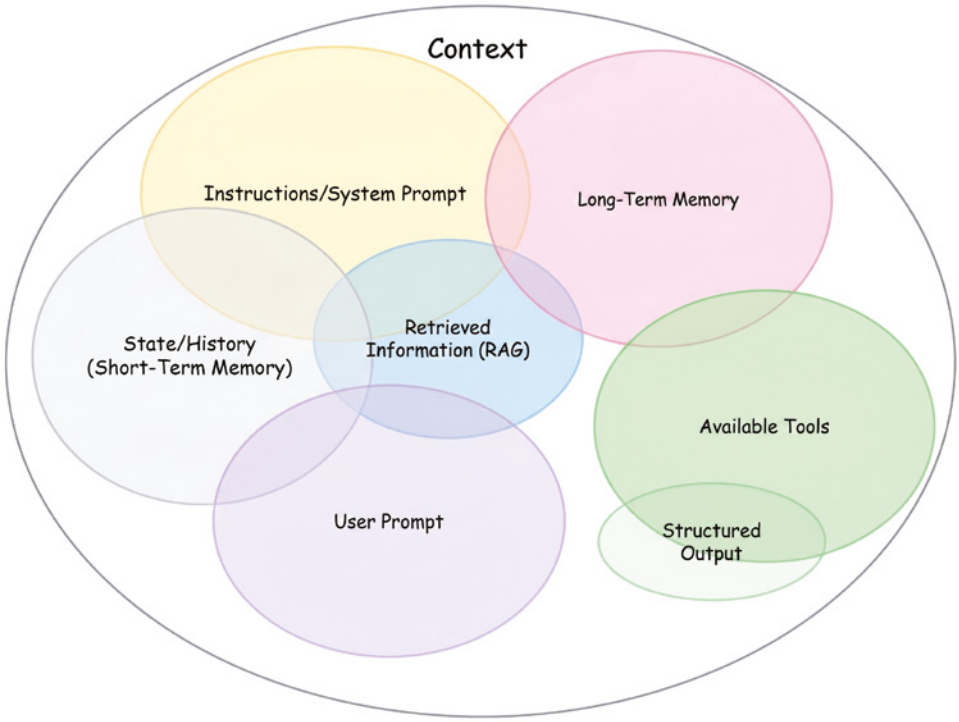

Figure 2.1: The LLM context window is a shared space competed for by every input the model receives.

RAG as a Solution Pattern

RAG breaks the problem into two phases: indexing and querying. During indexing, you process your documents and store them in a way that makes retrieval fast and accurate. During querying, you take the user's question, find the most relevant pieces of information, and include those in the prompt you send to the model.

The indexing phase happens once, usually when you first set up your system or when you add new documents. You load your documents, split them into manageable chunks, convert each chunk into a vector representation, and store those vectors in a database optimized for similarity search.

The querying phase happens every time a user asks a question. You convert their question into a vector, search your database for the chunks with the most similar vectors, retrieve those chunks, and include them in the prompt. The model then generates a response based on both its training and the specific context you provided.

This chapter focuses on vector-based semantic search as the retrieval mechanism, which is one of the most common and powerful RAG implementations. But it's worth noting that retrieval can also use keyword search, database queries, or any combination of these. The pattern is what matters: retrieve the relevant information, add it to the context, then let the model generate a grounded response.

This pattern is powerful because it separates knowledge storage from knowledge retrieval. You can have millions of documents indexed, but you only pull in the few dozen most relevant chunks for any given query. The model stays focused, costs stay manageable, and responses stay grounded in your specific data.

The LLM Application Stack

RAG fits into the larger AI engineering stack we discussed in Chapter 1, but it's worth seeing how the layers map when you zoom in on a data-intensive application. The *LLM application stack* for a RAG system shown in Figure 2.2 unpacks the orchestration and application layers from Chapter 1 into more granular pieces:

- **Data layer:** Your source documents, databases, and APIs—everything you want your LLM to know about.
- **Processing layer:** Chunking, embedding, and indexing your data so it's searchable.
- **Retrieval layer:** Vector databases and search algorithms that find relevant information.
- **LLM layer:** The actual language model (GPT-5, Claude, Llama, etc.).
- **Application layer:** Your frontend interface, backend APIs, business logic, prompts, and agents.

The data, processing, and retrieval layers collectively correspond to what Chapter 1 called the orchestration layer, which is all the work that happens between raw documents and the model call. The LLM and application layers map directly onto Chapter 1's LLM and application layers. It's one architecture, but now you have two views of it.

In this chapter, we're focusing on the middle three layers: processing, retrieval, and LLM. We're building a complete RAG system that takes raw documents, makes them searchable, and connects them to an LLM that can answer questions about them. Later chapters will add more sophistication to this: agents that can decide when to retrieve, multistep reasoning, memory systems, and more. But RAG is the foundation. Get this right, and everything else gets easier.

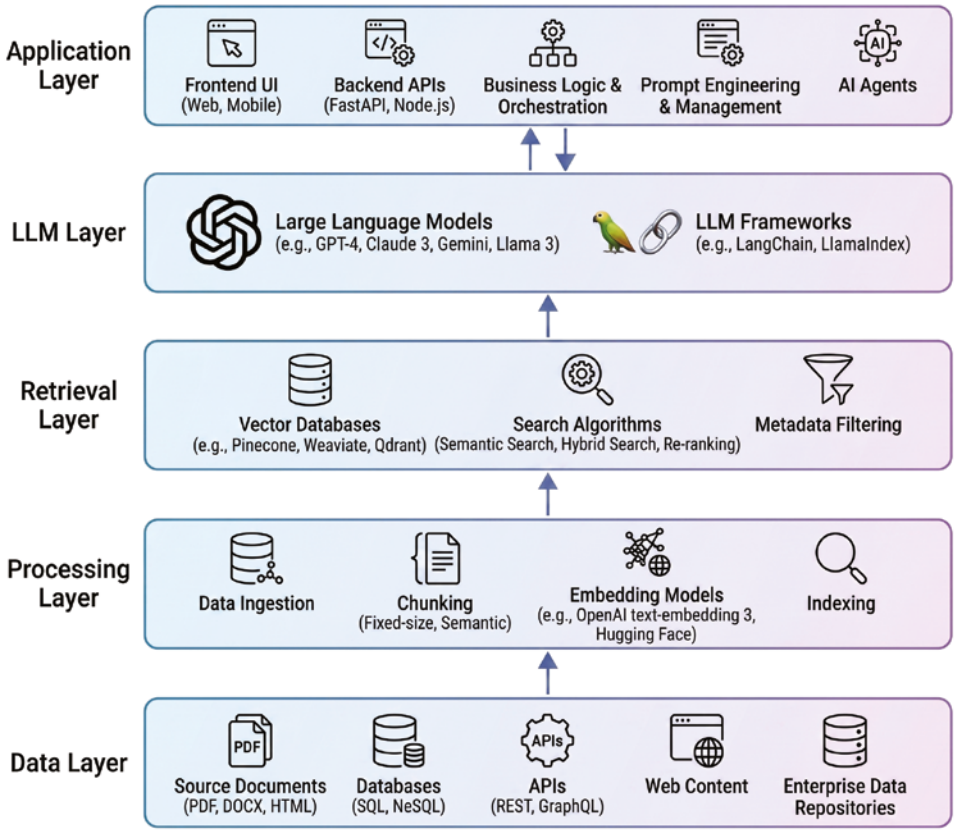

Figure 2.2: The LLM application stack.

Introduction to Embeddings

Embeddings—also called *embedding vectors*—are what makes RAG work. They're how we turn text into vectors: lists of numbers that machines can compare mathematically. If you've never worked with embeddings before, the concept can feel abstract at first, but stick with it because this is one of those fundamental ideas that unlocks a ton of what's possible in AI engineering.

What Are Embeddings?

An embedding is a list of numbers—called a *vector*—that represents the meaning of a piece of text. When you embed the sentence "The cat sat on the mat," you get back something like

```
[0.23, -0.45, 0.67, -0.12, ...]
```

going on for hundreds or thousands of dimensions. Each number in that list captures some aspect of the text's meaning.

The key insight is that similar meanings produce similar vectors. The embedding for "The cat sat on the mat" is going to be close in vector space to the embedding for "A feline rested on a rug." They use different words, but the semantic meaning is similar, so the vectors are similar. Meanwhile, "The stock market crashed today" will have a vector that's far away from both of those.

This is what makes embeddings so powerful for search. Instead of matching keywords, you're matching meaning. A user can ask "How do I reset my password?" and you can retrieve a document titled "Account Recovery Procedures" even though they share no words in common. The embeddings capture that both are about the same concept.

Semantic Similarity vs. Keyword Matching

Traditional search engines rely on keyword matching. If a user searches for "automobile insurance," the search engine looks for documents containing those exact words or close variants. This works okay when users know the right keywords to use, but it breaks down in a lot of situations.

What if the user searches for "car coverage" but your documents say "vehicle protection policy"? Keyword matching misses that connection. What if they ask a question in natural language like "What should I do if I get into an accident while traveling abroad?" and your documentation has a section titled "International Collision Procedures"? Again, keyword matching struggles.

Semantic search solves this by comparing meanings instead of words. The embedding model has learned that "car" and "vehicle" and "automobile" are related concepts. It knows that "coverage" and "insurance" and "protection policy" are in the same semantic space. When you search using embeddings, you find documents based on conceptual similarity, not just word overlap.

This doesn't mean keyword matching is obsolete. Sometimes exact word matches matter—think: legal documents, technical specifications, code search. The best systems often combine both approaches, which we'll explore in Chapter 9. But for most RAG applications, semantic search is the primary mechanism.

Embedding Models

Embedding models are specialized neural networks trained to produce good vector representations of text. The model you choose affects the quality of your retrieval, so it's worth understanding the landscape.

OpenAI's `text-embedding-3-small` and `text-embedding-3-large` are solid choices. They're easy to use, well documented, and produce 1,536-dimensional vectors for `small` and 3,072-dimensional vectors for `large`. The large model is more accurate but costs more and is slower. For most applications, small is fine.

Cohere's embedding models are another good option, especially if you're doing multilingual work. They support dozens of languages and are optimized for retrieval tasks.

There's also a growing ecosystem of open-source embedding models you can run yourself. Models such as sentence-transformers/all-MiniLM-L6-v2 and BAAI/bge-large-en-v1.5 perform well and can be self-hosted if you have privacy requirements or want to avoid API costs. We'll explore self-hosted models in Chapter 13.

When evaluating embedding models, the key metrics are retrieval accuracy and speed. The MTEB (Massive Text Embedding Benchmark) leaderboard ranks models across various tasks. For production systems, you also care about cost, latency, and whether the model supports your languages. Begin with a hosted model like OpenAI's text-embedding-3-small because it's hard to beat for getting started.

How Embeddings Capture Meaning

Understanding exactly how embedding models learn to capture meaning would require a deep dive into neural network architectures and training procedures, but the high-level picture is worth knowing.

These models are trained on massive amounts of text using a technique called *contrastive learning*. The basic idea: the model learns to make similar sentences have similar vectors and dissimilar sentences have different vectors. During training, it sees pairs of sentences and learns to adjust its internal parameters so that semantically similar pairs end up close together in vector space.

The model isn't explicitly taught that "car" and "automobile" are synonyms. Instead, it learns this implicitly by seeing these words used in similar contexts across millions of documents. Words that appear in similar contexts get similar embeddings. Sentences that express similar ideas get similar embeddings. This is how semantic meaning emerges from the training process.

The dimensionality of the embedding—whether it's 384, 768, 1,536, or some other number—determines how much nuance the model can capture. More dimensions mean more capacity to represent subtle differences in meaning, but they also mean more computation and storage. It's a trade-off, and the optimal choice depends on your use case.

Vector Databases

Once you've got embeddings, you need somewhere to store them and a way to search through them efficiently. That's where vector databases come in.

Architecture and Design Principles

A *vector database* is a specialized database optimized for storing high-dimensional vectors and performing similarity searches. The core operation—given a query vector, find the vectors most similar to it—needs to be fast even when you have millions or billions of vectors stored.

Traditional databases aren't built for this. You could store vectors in PostgreSQL and compute similarity scores for every vector on every query, but that's slow. Vector databases use specialized indexing structures that make similarity search much faster, usually at the cost of some accuracy. This trade-off between speed and accuracy is fundamental to how these systems work.

Under the hood, vector databases organize vectors in ways that let them quickly narrow down the search space. Instead of comparing your query to every single vector, they use index structures to identify candidate regions and only compare against vectors in those regions. Think of it like using a library's card catalog instead of scanning every book on every shelf.

Popular Vector Database Solutions

The vector database landscape has exploded in the last few years. Here are the main players you should know about:

- **Chroma** is an open-source embedding database that's easy to get started with. It runs in memory for development, scales to persistent storage for production, and has a simple API. Great for prototyping and small- to medium-sized projects.
- **FAISS**, developed by Meta, is a library for efficient similarity search. It's not a database, really. It's more of a toolkit for building similarity search into your own systems. If you need maximum control or you're building at massive scale, FAISS is worth learning. But it's lower level than the other options.
- **Pinecone** is a fully managed vector database designed for production workloads. You don't manage any infrastructure; you just use their API. It's fast, it scales automatically, and it handles all the operations complexity. The trade-off is cost and vendor lock-in.
- **Qdrant** is an open-source vector database with strong performance characteristics and a clean API. It's what we'll use starting in Chapter 3. You can run it yourself or use their managed cloud service. The developer experience is excellent, and it handles filtering and metadata well.
- **Weaviate** is another open-source option with a focus on semantic search and hybrid search capabilities. It supports multiple vector indexes, has good integration with various ML frameworks, and can combine vector search with traditional filtering.

There are also vector search extensions for traditional databases, such as pgvector for PostgreSQL. These let you add vector search to your existing database infrastructure, which is convenient when you already have data in those systems.

When You Need a Vector Database

Not every application needs a full vector database. If you're working with a small dataset—say, a few hundred documents—you can get away with simpler approaches. Store your vectors in memory, compute similarity on the fly, and call it a day. The performance will be fine, and you avoid the operational complexity of running a database.

You start needing a proper vector database when your dataset grows to thousands or millions of documents, when you need fast query times at scale, when you need to filter results based on metadata, or when you need to support concurrent queries from multiple users. Production RAG systems almost always use a vector database.

For this chapter, we're going to build a custom vector database from scratch as a learning exercise. You'll understand exactly what's happening when you call a similarity search. Then in Chapter 3, we'll switch to Qdrant for a more robust implementation.

Indexing Strategies

When you have millions of vectors, comparing your query to every single one is too slow. Vector databases use indexing strategies to make search faster. Understanding these strategies helps you choose the right configuration for your use case and debug performance issues.

Brute Force vs. Approximate Search

The simplest approach to similarity search is brute force: compute the similarity between your query vector and every vector in your database, then return the top-k results. This is exact—you're guaranteed to find the true nearest neighbors—but it's also slow. With a million vectors, you're doing a million distance calculations on every query.

For small datasets, brute force is actually fine. If you've got a thousand vectors and each distance calculation is cheap, you can do a thousand comparisons in milliseconds. But once you get to tens or hundreds of thousands of vectors, you need something faster.

That's where approximate nearest neighbor search comes in. Instead of checking every vector, you use an index structure to quickly narrow down candidates, then only compute exact distances for those candidates. You trade a small amount

of accuracy for a massive speedup. In practice, the accuracy loss is usually negligible—for example, you might miss the 10th closest result and return the 11th instead—but for most applications that doesn't matter.

HNSW: Hierarchical Navigable Small World

HNSW is one of the most popular indexing algorithms for approximate nearest neighbor search. It builds a multilayer graph structure where each vector is a node and edges connect similar vectors.

Think of it like a highway system. The top layers are like interstate highways that let you quickly travel long distances across the vector space. As you get closer to your destination, you drop down to smaller roads—the lower layers—that give you more local connections. When searching, you start at the top layer, navigate to the general region where your query belongs, then descend through the layers, getting more precise at each level.

The genius of HNSW is how it balances build time, search time, and accuracy. Building the index takes longer than simpler approaches, but once it's built, searches are extremely fast—usually logarithmic in the number of vectors—and accuracy is excellent. For most RAG applications, HNSW is the default choice.

IVF: Inverted File Index

IVF takes a different approach. It clusters your vectors into groups during indexing, then assigns each vector to its nearest cluster. When you search, you first identify the most relevant clusters, then only search within those clusters.

Imagine organizing a library by topic. Books about physics go in one section, books about history in another, books about cooking in a third. When someone asks for a physics book, you don't search the entire library; you go straight to the physics section and search there. That's IVF.

IVF is particularly good when your data has natural clusters—for example, a document collection that's clearly divided by topic, language, or product category—or when you need to handle massive datasets with billions of vectors. The trade-off is that accuracy depends heavily on how many clusters you search. Search too few, and you might miss relevant results that fell into nearby clusters. Search too many, and you lose the speed benefit.

Trade-Offs: Speed vs. Accuracy

Every indexing strategy forces you to choose between speed and accuracy. Brute force is perfectly accurate but slow. Approximate methods are fast but might occasionally miss a result or return something slightly suboptimal.

In practice, this trade-off is usually worth it. Getting 95% accuracy at 100 times the speed is almost always better than perfect accuracy at a crawl. Most

RAG applications retrieve multiple documents anyway, so you're getting the top 5 or top 10 results, not just the single best match. If your approximate index returns the second-best result as the top match, that's probably fine.

The key is understanding these trade-offs and tuning your system appropriately. Your vector database will expose parameters that control the speed-accuracy balance. More thorough indexes mean slower writes and more memory, but faster and more accurate reads. Adjust based on your use case. For example, a real-time customer support bot needs low latency; a document ingestion pipeline that runs overnight can afford slower write throughput in exchange for improved accuracy; a system running on constrained infrastructure may need to optimize for memory footprint. Know your bottleneck and tune accordingly.

Similarity Search

Similarity search is the core operation in RAG. You have a query vector and a database full of document vectors. You want to find the most similar documents. Simple concept, but the details matter.

Distance Metrics

There are three main ways to measure how similar two vectors are: cosine similarity, Euclidean distance, and dot product.

Cosine Similarity

Cosine similarity measures the angle between two vectors, not their lengths. Think of it as an "are these pointing in the same direction?" test, with a bit of trigonometry doing the heavy lifting. It returns a value between −1 and 1, which can be interpreted as follows:

- **Cosine similarity = 1:** The vectors point in exactly the same direction. They are perfectly aligned. One vector may be longer than the other, but direction-wise they're identical. In practice, this means maximum similarity. In text embeddings, you interpret this value as "these two things mean the same thing."
- **Cosine similarity = 0:** The vectors are at right angles (90° apart). They are *orthogonal*. There is no directional relationship between them. In practical terms, this means no similarity. For embeddings, you interpret this value as "these two things are unrelated."
- **Cosine similarity = −1:** The vectors point in exactly opposite directions. They are perfectly anti-aligned. This is maximum dissimilarity. In many embedding spaces, this often means "opposites" or strong contradiction, though not all models use negative space that cleanly.

Here's a useful mental model: cosine similarity ignores magnitude and only cares about direction. If you imagine each vector as an arrow from the origin, cosine similarity tells you whether the arrows overlap (1), are unrelated (0), or fight each other (−1).

One practical note: in many modern embedding models (especially NLP ones), values cluster between about 0 and 1, and strongly negative values are relatively rare. So −1 is more of a theoretical extreme than a common real-world outcome.

For two vectors A and B, their cosine similarity is given by the following formula:

$$\frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i}^{n} B_i^2}}$$

Where `(A · B)` is the dot product of the vectors and A_i and B_i are the *i*th elements of A and B, respectively.

This is the most common metric for text embeddings. It handles different vector magnitudes gracefully: a longer document and a shorter document can still be highly similar if they're about the same topic.

We included cosine similarity in our `SimpleVectorDB` class, earlier, but here's a regular Python function that you can use to try it out:

```
import numpy as np

def cosine_similarity(vec1: np.ndarray, vec2: np.ndarray) -> float:
    """Calculate cosine similarity between two vectors"""
    dot_product = np.dot(vec1, vec2)
    norm1 = np.linalg.norm(vec1)
    norm2 = np.linalg.norm(vec2)
    return dot_product / (norm1 * norm2)

# Example usage
vec_a = np.array([1.0, 2.0, 3.0])
vec_b = np.array([2.0, 4.0, 6.0])
similarity = cosine_similarity(vec_a, vec_b)
print(f"Cosine similarity: {similarity:.4f}")
```

Output:

```
Cosine similarity: 1.0000
```

These two vectors are perfectly aligned, which makes sense because `vec_b` is just a scalar multiple of `vec_a`.

Other Distance Metrics

Euclidean distance measures the straight-line distance between two points in space. The smaller the distance, the more similar the vectors. Unlike cosine similarity, Euclidean distance is sensitive to vector magnitude, which can skew results when comparing the embeddings of documents that have different lengths. Euclidean distance has its uses, particularly for non-text vectors such as image features, but for text embeddings, it's less reliable than cosine similarity.

The *dot product* measures both the angle and magnitude. Higher values mean the vectors are more similar. If your embedding model normalizes its output vectors, the dot product is mathematically equivalent to cosine similarity, but faster to compute. Always check your model's documentation; if it normalizes, you can use dot product as a performance optimization.

The bottom line: for text embeddings, stick with cosine similarity unless you have a specific reason to use something else. For RAG applications, cosine similarity is the gold standard.

K-Nearest Neighbors

When you search for similar vectors, you typically don't just want the single most similar one; you want the top *k* results. This is called *k-nearest neighbors* search, or *kNN*. The *k* is a parameter you choose based on how many documents you want to retrieve.

For RAG applications, *k* is usually somewhere between 3 and 20. Too few results, and you might miss important context. Too many, and you waste context window space on less relevant information. The right value depends on your chunk size, how specific your queries are, and how diverse your document collection is.

In practice, you might retrieve more chunks than you actually use. Retrieve the top 20, then apply some filtering or reranking to narrow down to the top 5 that actually go into the prompt. We'll explore reranking in detail in Chapter 9.

Approximate Nearest Neighbors

Approximate nearest neighbors, or *ANN*, is what your vector database is actually doing under the hood. Instead of computing exact distances to every vector, it uses indexing structures like HNSW or IVF to quickly find candidates that are probably close to your query.

The approximation introduces some error. Sometimes you'll get the 11th closest vector instead of the 10th. For most applications, this doesn't matter. The gain in speed is worth the tiny loss in accuracy. What matters is that your top results are all relevant, not that they're in perfectly precise order.

Most vector databases let you tune the accuracy-speed trade-off through configuration parameters. More thorough searches take longer but return better results. Fast searches might occasionally miss something. Tune based on your latency requirements and how much accuracy matters for your use case.

Choosing the Right Metric

For text embeddings, cosine similarity is almost always the right choice. It captures semantic similarity well and isn't sensitive to vector magnitude, which usually doesn't carry meaningful information in embeddings.

If your embedding model produces normalized vectors—and most do—then dot product is equivalent to cosine similarity but faster to compute. Check your model's documentation to see if vectors are normalized. If they are, use dot product. If not, use cosine similarity.

Euclidean distance can work for embeddings but is less common. It makes sense for other types of vectors where magnitude matters—image features, audio spectrograms, things like that. For text, stick with cosine or dot product.

Chunking Strategies

Before you can embed documents, you need to split them into chunks. Chunking is one of those decisions that seems simple but has a huge impact on your RAG system's performance. Get it right, and your retrieval is accurate and efficient. Get it wrong, and you'll spend weeks debugging why your system returns irrelevant results.

Why Chunking Matters

You can't embed an entire book as a single vector. Well, technically you could, but you shouldn't. Embeddings work best on coherent pieces of text that express a focused idea. A whole book contains too many diverse topics, so the embedding would be a muddy average that doesn't represent any particular part well.

Even a shorter document usually needs to be split up. A 10-page technical document covers multiple subtopics. If you embed it all as one chunk, a query about a specific detail will retrieve the entire 10 pages, most of which aren't relevant. You want granular retrieval that gets just the paragraph or section that answers the question.

But if you chunk too small, you lose context. A single sentence often doesn't make sense without the sentences around it. You need enough text in each chunk to capture a complete idea while keeping chunks small enough for focused retrieval.

Fixed-Size Chunking

The simplest approach is *fixed-size chunking*: split your documents into chunks of *N* tokens or characters. For example, take every 500 words and make a chunk. Move forward 400 words and take the next 500 words, creating 100 words of overlap. Repeat until the document is done. Here's a Python implementation:

```
def fixed_size_chunking(text: str, chunk_size: int = 500,
                        overlap: int = 50) -> List[str]:
    """Split text into fixed-size chunks with overlap"""
    words = text.split()
    chunks = []

    for i in range(0, len(words), chunk_size - overlap):
        chunk = words[i:i + chunk_size]
        chunks.append(' '.join(chunk))

    return chunks
```

Fixed-size chunking is easy to implement and works reasonably well as a starting point. It's also predictable because you know exactly how many chunks you'll get and how big each one will be. The downside is that it doesn't respect natural boundaries in the text. You might split a chunk right in the middle of a paragraph or even mid-sentence.

Typical chunk sizes range from 256 to 1,024 tokens. Smaller chunks give you more granular retrieval but less context per chunk. Larger chunks provide more context but less precision. A good starting point is 512 tokens with 50–100 tokens of overlap.

Semantic Chunking

Semantic chunking tries to split documents at natural boundaries—paragraphs, sections, sentences—rather than at arbitrary character counts. The idea is to keep semantically coherent units together.

The simplest version is paragraph-based chunking. Split on double newlines, treating each paragraph as a chunk. This works surprisingly well for many documents because paragraphs usually contain complete thoughts. If a paragraph is too long, you can further split it at sentence boundaries. Here's a practical implementation that chunks on paragraph boundaries:

```
def semantic_chunking(text: str, max_chunk_size: int = 1000) ->
List[str]:
    """Split text into semantic chunks at paragraph boundaries"""
    # Split on double newlines (paragraph breaks)
    paragraphs = text.split('\n\n')
```

```
    chunks = []
    current_chunk = []
    current_size = 0

    for paragraph in paragraphs:
        paragraph = paragraph.strip()
        if not paragraph:
            continue

        para_size = len(paragraph.split())

        # If adding this paragraph exceeds max size, start new chunk
        if current_size + para_size > max_chunk_size and current_chunk:
            chunks.append('\n\n'.join(current_chunk))
            current_chunk = [paragraph]
            current_size = para_size
        else:
            current_chunk.append(paragraph)
            current_size += para_size

    # Add the last chunk
    if current_chunk:
        chunks.append('\n\n'.join(current_chunk))

    return chunks

# Example usage
document = """First paragraph about topic A.

Second paragraph also about topic A.

Third paragraph about topic B.

Fourth paragraph about topic A.

Fifth paragraph also about topic A.

Sixth paragraph about topic B."""

chunks = semantic_chunking(document, max_chunk_size=20)
for i, chunk in enumerate(chunks):
    print(f"Chunk {i+1}: {len(chunk.split())} words")
```

This approach keeps paragraphs together, which preserves semantic coherence. You can also adapt it to split on section headers, sentences, or custom boundaries depending on your document structure.

For documents with clear structure (like technical documentation with headers), you might want to chunk by section:

```
def chunk_by_sections(text: str, header_pattern: str = r'^#{1,3} ') ->
List[str]:
    """Chunk markdown documents by section headers"""
    import re

    lines = text.split('\n')
    chunks = []
    current_chunk = []

    for line in lines:
        # Check if line is a header
        if re.match(header_pattern, line):
            # Save previous chunk if it exists
            if current_chunk:
                chunks.append('\n'.join(current_chunk))
            current_chunk = [line]
        else:
            current_chunk.append(line)

    # Add the last chunk
    if current_chunk:
        chunks.append('\n'.join(current_chunk))

    return chunks
```

More sophisticated approaches use embeddings themselves to find natural split points. You compute embeddings for each sentence, then look for points where the embedding changes significantly, which is likely a topic boundary. This is more complex to implement but can produce excellent results on unstructured text.

Overlap and Context Preservation

Overlap between chunks helps preserve context across boundaries. If you split exactly at the end of one chunk and start the next chunk immediately after, you risk losing context when a concept spans the boundary.

By overlapping chunks—say, the last 100 tokens of one chunk are also the first 100 tokens of the next—you ensure that concepts near boundaries appear fully in at least one chunk. The trade-off is storage and indexing cost because you're storing some text multiple times. But for most applications, the improved retrieval quality is worth it.

How much overlap? Typically, 10–20% of your chunk size is a solid starting point because it's enough to preserve context across boundaries without storing large amounts of duplicate text. If you're using 500-word chunks, 50–100 words of overlap keeps ideas that span a chunk boundary from disappearing into the gap. Go below 10%, and you still risk losing context. Go above 20%, and you're mostly duplicating storage for diminishing returns.

Chunking Best Practices

Start simple with fixed-size chunks and iterate from there. Use 512 tokens with 50–100 tokens of overlap as your baseline. Test your retrieval quality with real queries. If you're consistently getting chunks that cut off mid-thought, consider semantic chunking.

Keep metadata with your chunks. Store not just the chunk text but also the source document, the page number, the section heading, the timestamp—whatever context helps. This metadata is useful for filtering and for explaining to users where information came from.

Consider your use case when choosing chunk size. Customer support chatbots might want smaller chunks for precise answers. Research assistants might want larger chunks for more comprehensive context. Legal document analysis might want section-based chunking to respect document structure.

Test different chunking strategies against your evaluation dataset. This is what Chapter 8 is all about. You can't optimize what you don't measure. Build a test set of queries and expected results, then systematically evaluate how different chunking approaches perform.

The RAG Pipeline

Now that you understand the individual pieces, let's see how they fit together into a complete RAG pipeline. There are two phases: indexing and querying. Indexing happens once per document. Querying happens on every user request.

The Indexing Phase

Indexing transforms your documents into a searchable form. As shown in Figure 2.3, the pipeline looks like this: load your documents, chunk them into smaller pieces, generate embeddings for each chunk, and store those embeddings in your vector database along with the original text and any metadata.

Loading documents can be as simple as reading text files or as complex as parsing PDFs, extracting tables, and preserving formatting. For now, assume you've got plain text. Tools like LangChain provide document loaders for common formats, which we'll use in Chapter 3.

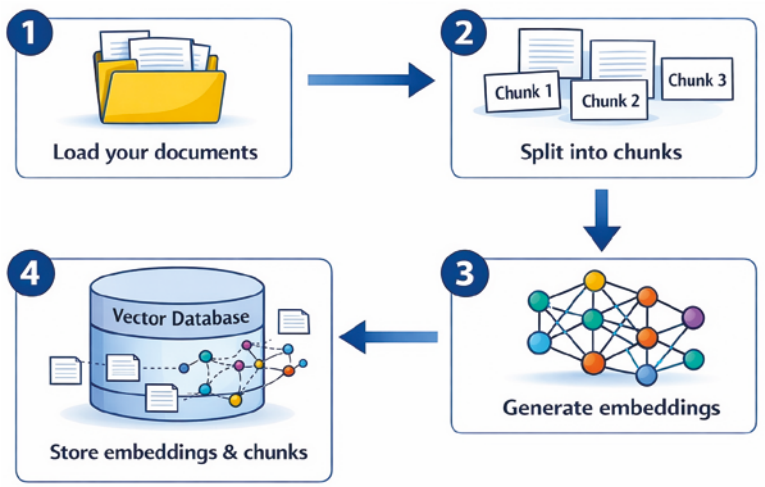

Figure 2.3: The indexing phase of a RAG system.

Chunking splits each document into pieces using one of the strategies we just discussed. Each chunk becomes a separate entry in your database. Keep track of which chunk came from which document and where in the document it appeared, because that metadata is crucial.

Embedding converts each chunk of text into a vector. You call your embedding model—OpenAI, Cohere, whatever you chose—and get back a list of numbers for each chunk. This is usually the most expensive part of indexing in terms of API costs and time.

Storing puts everything into your vector database. Each entry contains the embedding vector, the original chunk text, and metadata like source document, chunk index, timestamps, or custom fields you want to filter on later. The database builds its indexes so similarity search will be fast.

The Query Phase

The query phase handles user requests in real time, as shown in Figure 2.4. A user asks a question. You embed their question using the same model you used for indexing, search your vector database for the most similar chunks, retrieve those chunks, format them into a prompt with the user's question, send that prompt to your LLM, and return the model's response.

Embedding the query is straightforward: take the user's question as a string, call your embedding model, and get back a vector. This needs to be fast because it happens on every query. Embedding APIs typically return results in 100–300 ms.

Searching the vector database finds the k most similar chunks. Your database does the heavy lifting here, using its indexes to quickly narrow down candidates. You specify *k*—how many results you want—and optionally some filters on metadata. The database returns the top *k* chunks along with their similarity scores.

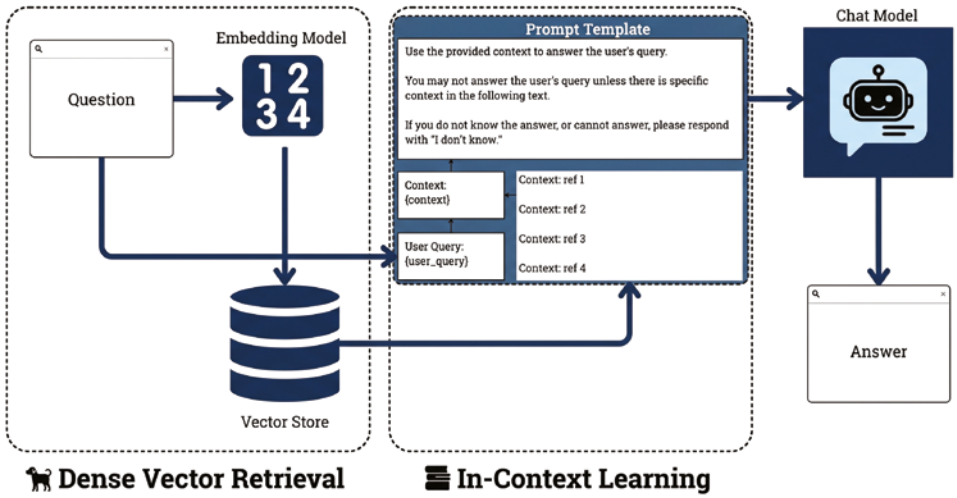

Figure 2.4: The query phase of the RAG pipeline.

Retrieving means pulling out the actual text of those chunks along with any metadata. You now have the relevant context for answering the user's question.

Generating the response puts it all together. You build a prompt that includes the user's question and the retrieved context. Something like: "Given the following context, answer the question. Context: [chunk1] [chunk2] [chunk3]. Question: [user question]." You send this to your LLM and stream back the response.

Common Pitfalls

RAG seems simple, but there are failure modes you need to watch for. One common issue is retrieving chunks that are relevant to the query but don't actually contain the information needed to answer it. The chunk matches semantically but doesn't have the details. This usually means your chunking is off: chunks are too small or split poorly.

Another issue is retrieved chunks being out of order or missing context. The user asks about a process that spans multiple paragraphs, but your system retrieves paragraph 3 and paragraph 7 without the connecting steps. Overlap helps, but sometimes you need smarter retrieval strategies that fetch surrounding context or use document structure.

A third pitfall is over-retrieving. You grab 20 chunks because you're worried about missing something, but now your prompt is cluttered with tangentially relevant information and the model gets confused. More context isn't always better. Quality over quantity.

Finally, watch out for embedding model mismatches. If you indexed with one model but query with a different model, your vector spaces don't align and retrieval falls apart. Always use the same embedding model for indexing and querying.

The 12-Factor Agent Methodology

Before we jump into building StudyBuddy v2, let's talk about production readiness. The 12-Factor App methodology, originally written for web applications, has been adapted for AI systems. These principles guide how you architect, configure, and deploy agents so they're maintainable and scalable.

The original 12 factors—codebase, dependencies, config, backing services, build/release/run, processes, port binding, concurrency, disposability, dev/prod parity, logs, and admin processes—all apply to AI systems. But there are additional considerations specific to agents.

Configuration and Dependencies

Configuration should live in environment variables, not hard-coded in your source. API keys, model names, database connection strings, all of that goes in environment variables or a secrets manager. This lets you change configuration without changing code and keeps secrets out of version control.

Dependencies must be explicitly declared and isolated. Use virtual environments for Python, lock files for reproducible builds, and specify exact versions of libraries. Don't rely on system-wide packages. This is foundational—if your dependencies aren't managed, deployments break in weird ways.

State Management Patterns

Agents need to manage state, including conversation history, retrieved documents, and intermediate reasoning steps. The 12-factor methodology says processes should be stateless and share nothing. For agents, this means storing state in backing services like databases or caching layers, not in memory on the application server.

If your agent process crashes and restarts, it should be able to pick up where it left off by loading state from persistent storage. Conversely, you should be able to run multiple agent processes concurrently without them stomping on each other's state.

For RAG applications, this means your vector database is a backing service. Your embeddings, your chunks, your indexes—they all live in the database, not in process memory. This lets you scale horizontally by adding more application servers while they all share the same vector database.

Applying 12-Factor to AI Systems

The build, release, and run phases should be strictly separated. Building means generating your application artifacts—container images, packaged code, whatever. Releasing combines your build with configuration for a specific environment. Running executes that release. You never change code in production. You build a new release and deploy it.

Disposability means your agent processes should start fast and shut down gracefully. This matters for scaling and deployment. If spinning up a new process takes five minutes, you can't respond quickly to load spikes. If shutting down drops in-flight requests, users have a bad experience. Fast startup and graceful shutdown enable modern deployment patterns like rolling updates and autoscaling.

Logs should be treated as event streams. Don't write log files. Instead, emit logs to `stdout` and let your infrastructure collect them. This is crucial for debugging agent behavior in production. You want to see what the agent retrieved,

what prompt it generated, and what the model returned, all in a centralized logging system.

As we build more sophisticated systems in later chapters, these principles become increasingly important. For now, just be aware of them. Structure your code with 12-factor principles in mind, and it'll be much easier to deploy to production later.

Building StudyBuddy v2

Alright. Time to build. You've got the concepts. Now let's implement them.

In Chapter 1, you built StudyBuddy v1, a basic tutoring chatbot that could explain concepts but had no access to specific study materials. It was helpful but limited. If a student asked about a topic not in the model's training data, v1 couldn't help. If they uploaded their textbook and asked questions about it, v1 had no way to reference that content.

StudyBuddy v2 fixes that. We're adding RAG so students can get explanations grounded in specific study materials. Ask about mitosis, and you'll get an answer that references the exact explanations from the indexed content.

Note that the following code examples reference the StudyBuddy app that we build in this book, so if you want to follow along, you can access the repo directly at `https://github.com/AI-Maker-Space/studybuddy`.

What We're Adding

StudyBuddy v2 adds RAG retrieval and source attribution. When a student asks a question, StudyBuddy retrieves relevant sections from indexed study materials and uses them to generate grounded explanations. The response includes references to the documents the information came from.

Under the hood, we're building a RAG pipeline from scratch. No frameworks yet because we want you to see exactly how everything works. We'll create our own simple vector database in memory, implement chunking, call the OpenAI embeddings API, perform similarity search, and generate augmented prompts.

This is intentionally a learning exercise. In Chapter 3, we'll rebuild this using proper tools like LangChain and Qdrant. But for now, writing it yourself helps you understand what those tools are actually doing.

Architecture

Like v1, StudyBuddy v2 uses a hybrid architecture: a Next.js frontend with a Python/FastAPI backend. The backend handles the RAG pipeline (chunking, embeddings, retrieval), while the frontend provides a React-based chat interface.

```
v2-rag-from-scratch/
├── api/
│   ├── index.py              # FastAPI + RAG pipeline
│   └── requirements.txt      # Python deps (includes numpy)
├── frontend/
│   ├── src/
│   │   ├── app/
│   │   │   ├── layout.tsx    # Root layout
│   │   │   ├── page.tsx      # Chat page with RAG status
│   │   │   └── globals.css   # Tailwind styles
│   │   └── components/
│   │       ├── Message.tsx       # Message with markdown
│   │       ├── MessageList.tsx   # Scrollable container
│   │       ├── MessageInput.tsx  # Input textarea
│   │       └── LoadingDots.tsx   # Loading animation
│   ├── next.config.ts        # API proxy config
│   └── package.json          # Node.js dependencies
├── .env                      # API keys
├── pyproject.toml
└── README.md
```

The key difference from v1 is the backend. While v1 just forwarded messages to OpenAI, v2's backend includes a complete RAG pipeline: chunking, embedding, storage, retrieval, and augmented generation.

The RAG Pipeline

The backend has two main flows: indexing (processing documents) and querying (answering questions). When the server starts, it indexes the built-in study materials, which are three short passages covering mitosis, photosynthesis, and the water cycle. This gives you a concrete knowledge base to query against. When a question comes in, it retrieves relevant chunks and augments the prompt.

Let's walk through the code. First, the vector database:

```
import numpy as np
from typing import List, Tuple

class SimpleVectorDB:
    def __init__(self):
        self.vectors = []
        self.texts = []
        self.metadata = []

    def add(self, vector: List[float], text: str, meta: dict):
        self.vectors.append(np.array(vector))
        self.texts.append(text)
        self.metadata.append(meta)
```

```
    def search(self, query_vector: List[float], k: int = 5):
        query = np.array(query_vector)
        similarities = []

        for vec in self.vectors:
            # Cosine similarity
            similarity = np.dot(query, vec) / (np.linalg.norm(query) *
np.linalg.norm(vec))
            similarities.append(similarity)

        top_k = np.argsort(similarities)[-k:][::-1]

        results = []
        for idx in top_k:
            results.append({'text': self.texts[idx], 'metadata': self
.metadata[idx],
                            'score': float(similarities[idx])})

        return results
```

This is about as simple as a vector database can get. We store vectors, texts, and metadata in parallel lists. When searching, we compute cosine similarity between the query and every stored vector, then return the top *k* results.

Cosine similarity is computed as the dot product of normalized vectors. We use NumPy for the math because it's fast. The `argsort` call finds the indices that would sort the similarity array. We take the last *k* to get the highest scores, then reverse them to get descending order.

Now the chunking function:

```
def chunk_text(text: str, chunk_size: int = 500,
               overlap: int = 50) -> List[str]:
    words = text.split()
    chunks = []

    for i in range(0, len(words), chunk_size - overlap):
        chunk = words[i:i + chunk_size]
        chunks.append(' '.join(chunk))

    return chunks
```

Fixed-size chunking by word count. We split the text into words, then take chunks of `chunk_size` words with `overlap` words of overlap. The stride is `chunk_size - overlap`, so each chunk starts where the previous chunk was `overlap` words from finishing.

Embedding and indexing:

```
from openai import OpenAI

client = OpenAI()
```

```
vector_db = SimpleVectorDB()

def embed_text(text: str) -> List[float]:
    response = client.embeddings.create(
        model='text-embedding-3-small',
        input=text
    )
    return response.data[0].embedding

def index_document(text: str, doc_name: str):
    chunks = chunk_text(text)

    for i, chunk in enumerate(chunks):
        embedding = embed_text(chunk)
        vector_db.add(embedding, chunk, {'doc_name': doc_name, 'chunk_
id': i})
```

The `embed_text` function calls OpenAI's API to generate embeddings. We're using text-embedding-3-small, which returns 1,536-dimensional vectors. The `index_document` function chunks the document, embeds each chunk, and stores it in our vector database along with metadata about which document it came from and which chunk number it is.

Query and retrieval:

```
def retrieve_context(query: str, k: int = 3) -> str:
    query_embedding = embed_text(query)
    results = vector_db.search(query_embedding, k=k)

    context_parts = []
    for result in results:
        context_parts.append(f"[From {result['metadata']['doc_name']}]:
\n{result['text']}")

    return '\n\n'.join(context_parts)

def answer_question(question: str) -> str:
    context = retrieve_context(question)

    prompt = f'''You are StudyBuddy, a helpful tutoring assistant.

Given the following context from study materials, answer the student's
question. Be clear and thorough, and cite which document you're
referencing when relevant.

Context:
{context}

Question: {question}'''

    response = client.chat.completions.create(
        model='gpt-5-nano',
```

```
        messages=[{
            'role': 'user',
            'content': prompt
        }]
    )

    return response.choices[0].message.content
```

The `retrieve_context` function embeds the query and searches the vector database. It formats the results with document attribution so we know where each chunk came from. The `answer_question` function retrieves context, builds a prompt that includes both the context and the question, and sends it to the LLM.

Backend Setup

The backend is a FastAPI application that connects to OpenAI's API and processes requests from your frontend.

Installing Dependencies

The dependencies are specified in the `pyproject.toml` file. Your first task is to open the file and update the `dependencies` section to the following:

```
dependencies = [
    "fastapi>=0.115.0",
    "uvicorn[standard]>=0.32.0",
    "openai>=1.54.0",
    "python-dotenv>=1.0.0",
    "numpy>=1.26.0",
]
```

It's particularly important to add the `python-dotenv` library because you need it to read the environment variables where things like your OpenAI API key are stored.

Vercel gets the project dependencies from a different file named `requirements.txt`, which must reside in the backend directory. Create a new text file named `requirements.txt` in the project's `api` directory, then add the following dependencies:

```
fastapi>=0.115.0
uvicorn[standard]>=0.32.0
openai>=1.54.0
python-dotenv>=1.0.0
numpy>=1.26.0
```

With that done, from the project root directory, run:

```
uv sync
```

Running StudyBuddy v2

StudyBuddy v2 uses the same two-terminal setup as v1. The backend indexes study materials on startup, and the frontend polls for indexing status.

In your backend server terminal:

```
cd /v2-rag-from-scratch
uv run uvicorn api.index:app --reload --port 8000
```

In your frontend server terminal:

```
cd /v2-rag-from-scratch/frontend
npm install
```

This installs all the Node.js dependencies specified in `package.json`, including Next.js, React, and react-markdown for rendering formatted AI responses.

Finally, on your frontend terminal, start the Next.js development server:

```
npm run dev
```

Visit `http://localhost:3000`. You'll see "Indexing. . ." in the footer while the backend processes the study materials, then "RAG enabled" once it's ready.

The backend includes built-in study materials about mitosis, photosynthesis, and the water cycle. When you ask a question, the system embeds it, finds relevant chunks, retrieves them, and generates an answer grounded in that specific text. The response will reference the source material.

The Frontend: RAG Status

The v2 frontend adds one key feature over v1: RAG status polling. The page uses `useEffect` to poll the `/api/status` endpoint every 2 seconds and give users feedback about whether the RAG pipeline is ready:

```
const [ragStatus, setRagStatus] = useState<{
    indexing_complete: boolean
    chunks_in_db: number
} | null>(null)

useEffect(() => {
    const checkStatus = async () => {
      try {
        const res = await fetch('/api/status')
        if (res.ok) {
            const data = await res.json()
            setRagStatus(data)
        }
      } catch {
      // Retry on failure - backend might not be ready yet
      }
```

```
    }

    checkStatus()
    const interval = setInterval(checkStatus, 2000)
    return () => clearInterval(interval)
}, [])
```

The footer displays the status:

```
<div className="text-center text-xs text-gray-400 py-2">
    StudyBuddy v2 · {ragStatus?.indexing_complete
        ? <span className="text-green-600">RAG enabled</span>
        : <span className="text-amber-500">Indexing...</span>}
      {ragStatus && ragStatus.chunks_in_db > 0 && (
      <span> · {ragStatus.chunks_in_db} chunks</span>
    )}
</div>
```

The rest of the frontend is identical to v1: React components for messages, markdown rendering, and a chat input.

Testing the RAG Pipeline

Try these questions to see RAG in action:

- "What happens during prophase?"
- "Explain how photosynthesis works."
- "What are the stages of the water cycle?"

The responses should reference the indexed study materials. This is RAG working end to end: embedding the query, retrieving relevant chunks, augmenting the prompt with context, and generating a grounded response.

Deploying to Vercel

To deploy v2 to Vercel, you need to update two files in the repo root. First, update `vercel.json` to point to the `v2-rag-from-scratch` directory:

```
{
  "version": 2,
  "builds": [
    {
      "src": "v2-rag-from-scratch/api/index.py",
      "use": "@vercel/python"
    },
    {
      "src": "v2-rag-from-scratch/frontend/package.json",
      "use": "@vercel/next"
    }
```

```
  ],
  "routes": [
    { "src": "/api/(.*)", "dest": "/v2-rag-from-scratch/api/index.py" },
    { "src": "/(.*)", "dest": "/v2-rag-from-scratch/frontend/$1" }
  ],
  "git": {
    "deploymentEnabled": {
      "main": true
    }
  },
  "ignoreCommand": "bash scripts/should-deploy.sh"
}
```

Second, update `scripts/should-deploy.sh` to trigger on v2 changes:

```
#!/bin/bash
# Deploy if changes are in v2-rag-from-scratch or vercel config
# Exit 1 = proceed with build, Exit 0 = skip build
git diff HEAD^ HEAD --name-only | grep -qE "^v2-rag-from-scratch/
|^vercel\.json" && exit 1 || exit 0
```

If you haven't done so already, set your OpenAI API key environment variable in Vercel:

```
vercel env add OPENAI_API_KEY
```

Deploy:

```
vercel --prod
```

Or simply push to GitHub. Vercel will automatically deploy when changes are detected in the `v2-rag-from-scratch` directory.

What's Next

StudyBuddy v2 works, but it's basic. We built everything from scratch to understand the fundamentals. In Chapter 3, we'll rebuild this using proper tools. LangChain will handle document loading and chunking. Qdrant will replace our simple in-memory vector database. We'll add more sophisticated retrieval strategies and better error handling.

But the core pattern stays the same: indexing documents, embedding queries, retrieving context, and generating responses. You now understand how that pattern works at a fundamental level. Everything else is optimization and scale.

You've now seen how RAG works at a fundamental level. RAG is foundational. You'll use this pattern in almost every AI application you build. Next up: making StudyBuddy truly agentic.

Let's keep building.

CHAPTER

3

The Agent Loop

Okay, here's where things get real. In Chapters 1 and 2, you built StudyBuddy from scratch. In Chapter 1, you stood up a basic LLM-powered chatbot and deployed it to Vercel. In Chapter 2, you wrote a vector database, implemented similarity search, chunked documents, and wired up the complete RAG pipeline yourself. That was a learning exercise, and an important one. You now understand exactly what's happening under the hood when you do retrieval-augmented generation.

But that version of StudyBuddy was limited. It could answer questions about documents, sure. But it couldn't *do* anything. It couldn't decide when to search versus when to answer from memory. It couldn't use multiple tools. It couldn't break down complex questions into steps.

In this chapter, we're rebuilding StudyBuddy the right way, with LangChain, with proper abstractions, and most importantly, with the *agent loop*. This is where StudyBuddy stops being a simple Q&A system and starts being an actual agent that can reason, plan, and take action.

This is the version of StudyBuddy we'll build on for the rest of the book. Every chapter from here forward adds new capabilities to what we're building today. This is the foundation for what follows.

Understanding the Agent Pattern

The word "agent" gets thrown around a lot in AI, and honestly, it's become a bit of a buzzword. Every chatbot calls itself an agent now. But there's a real, useful distinction between systems that just respond to inputs and systems that actively reason through problems. Understanding that distinction is critical for building effective AI systems.

What Makes Something an Agent?

An agent is an autonomous system that can leverage reasoning to make dynamic decisions in an application flow, all in the service of achieving a goal. It perceives its environment, reasons about what to do, takes actions that change the environment, and learns from the results. The keyword is autonomous. You give an agent a goal, and it figures out how to achieve it.

A simple chatbot isn't an agent. It takes your message, generates a response, and sends it back. There's no reasoning loop. There's no decision-making about what to do next. It's a single-shot input-output system. An agent, by contrast, can plan multiple steps, use tools, reflect on results, and adjust its strategy.

Think about booking a flight. A non-agentic system would require you to specify every detail: "Search for flights from SFO to JFK on March 15, returning March 22, for two passengers." You have to know exactly what you want. An agentic system could handle: "I need to visit New York for a week next month, keeping costs reasonable." The agent would figure out good dates, search for options, compare prices, maybe even check your calendar to avoid conflicts, and present you with a plan. It's making decisions at every step.

The distinguishing features of an agent are autonomy in decision-making, the ability to use tools to accomplish tasks, multi-step reasoning toward a goal, and reflection on results to improve future actions. If your system has all of these, it's an agent. If it's missing any of them, it's probably just a workflow or a chatbot pretending to be more than it is.

Agents vs. Workflows

Here's a distinction that matters: workflows execute predefined sequences. Agents make decisions. Both are useful, but they solve different problems.

A workflow for processing customer support tickets might look like this: receive ticket, categorize by topic, route to appropriate team, send an acknowledgment email, escalate if not resolved in 24 hours. Each step is predetermined. The system doesn't decide what to do next. Instead, it follows the script. This is powerful for repetitive, well-defined processes.

An agent for customer support would look at each ticket and decide: Is this a simple question I can answer immediately? Does this require looking up account information? Should I route this to a human because it's too complex or emotionally charged? Do I need to ask clarifying questions before I can help? The agent adapts its strategy based on the specific situation.

Workflows are deterministic and predictable. Agents are adaptive and context-sensitive. Workflows are easier to build, test, and debug. Agents are more flexible and can handle edge cases that workflows miss. In practice, the best systems often combine both. Use workflows for the parts of your process that are well-defined and stable. Use agents for the parts that require judgment and adaptation.

The ReAct Pattern

ReAct stands for reasoning and acting, and it's one of the foundational patterns for building agents. The idea is simple but powerful: before taking an action, the agent generates a thought explaining why it's taking that action. After observing the result, it generates another thought about what it learned. This interleaving of thought and action is what makes the agent's behavior interpretable and improvable.

Here's what a ReAct trace looks like in practice. Imagine asking an agent, "What did Sherlock Holmes deduce from the woman's appearance when she arrived at Baker Street in 'The Speckled Band'?" The agent's thought process might go like this: "Thought: This is a question about a specific detail from a specific story. I should search the study materials rather than rely on my general knowledge of the plot. Action: Call search_materials with query='Speckled Band woman Baker Street deductions'." It executes that search and gets results back. Then it thinks, "Observation: Found a relevant passage describing Holmes's observations about Helen Stoner's appearance and demeanor. Thought: I have the specific text needed to answer accurately. Final Answer: [summarizes Holmes's deductions with attribution to the story]."

The thoughts serve multiple purposes. They make the agent's decision-making transparent so you can see why it chose a particular action. They help the model stay on track by explicitly articulating the goal and progress toward it. They provide hooks for improving the system because you can see exactly where reasoning breaks down. And they enable debugging in a way that black-box systems don't.

ReAct isn't the only agentic pattern out there. There's Plan-and-Execute, where the agent first creates a complete plan and then executes each step. There's Tree of Thoughts, where the agent explores multiple reasoning paths in parallel. There's Reflexion, where the agent critiques its own outputs and iterates. But ReAct is the most widely used because it hits a sweet spot of effectiveness, interpretability, and ease of implementation. Most production agents are some variation of ReAct.

When You Need Agents

Not every problem needs an agent. Simple RAG works great for a lot of use cases. If your task is well-defined, if the steps are predictable, and if you don't need the system to make judgment calls, stick with simpler patterns. Agents add complexity, and complexity has costs in development time, debugging difficulty, and runtime performance.

You need an agent when:

- The task requires multiple steps that depend on intermediate results.
- The system needs to decide which tools to use based on context.
- There are multiple valid approaches and the agent needs to choose.
- Errors might occur and the system needs to recover gracefully.
- The solution path isn't knowable in advance.

Customer support is a classic agent use case. You don't know what the customer needs until you understand their question. You might need to look up their account, search documentation, check order status, or escalate to a human. The path depends entirely on what you learn at each step. Research assistants are another good fit. The user asks a broad question, the agent breaks it down, searches for information, synthesizes findings, identifies gaps, searches again, and eventually produces a comprehensive answer. Code review tools that can navigate codebases, run tests, and suggest fixes based on what they find. These all benefit from the agent pattern.

Simple question answering from a knowledge base? Probably doesn't need an agent. Translating text? Not an agent task. Classifying documents? Use a workflow. The rule of thumb: if you can write out the exact sequence of steps ahead of time, you don't need an agent. If the sequence depends on what you learn along the way, you do.

Core Functions of an Agent

Every agent, regardless of implementation details, has the same basic functions: perception, planning, action, and observation (see Figure 3.1). These form a loop that repeats until the agent achieves its goal or determines it can't. Understanding each function helps you design better agents and debug them when they fail.

Perception: Understanding the Current Situation

Perception is how the agent understands what's happening right now. For a conversational agent, perception means reading the user's message and understanding what they're asking for. For a research agent, it means understanding the

research question and what information is already available. For a code review agent, it means reading the pull request, the code changes, and any comments.

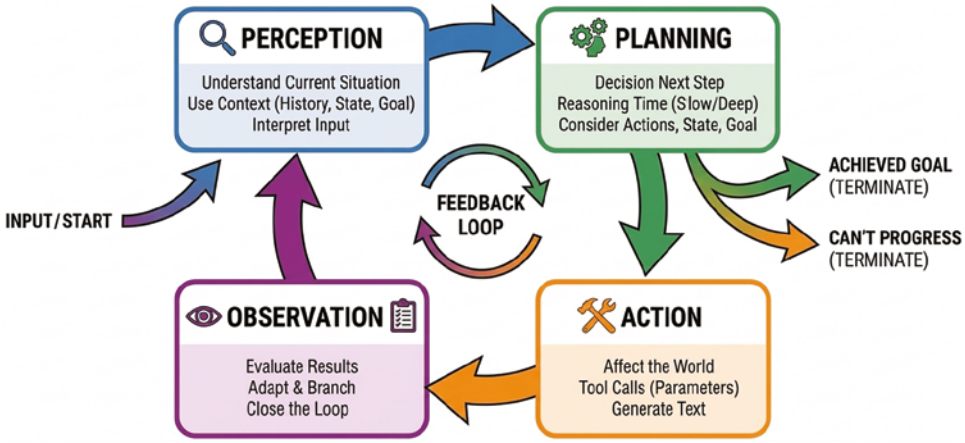

Figure 3.1: The four core functions of an agent form a continuous loop.

Good perception requires context. The agent needs to know not just the immediate input but also the history of the conversation, the current state of any tools it's using, and the goal it's working toward. This is why agent systems maintain state. They're keeping track of what's happened so far and using that to interpret new information.

In practice, perception is often the LLM reading a carefully constructed prompt that includes the user's current message, relevant conversation history, available tools and their descriptions, and instructions about what the agent should be doing. The LLM's output is its interpretation of the situation and its decision about what to do next. The quality of that output depends heavily on the quality of the input.

Planning: Deciding What to Do Next

Once the agent perceives the situation, it needs to decide what to do. This is where reasoning happens. The agent considers its available actions, the current state, and the goal it's trying to achieve. It decides which action will make the most progress.

Planning can be shallow or deep. Shallow planning is reactive: look at the current situation and pick the next action. This is what ReAct does. The agent asks: Given where I am now, what's the best next step? Deep planning involves thinking multiple steps ahead: if I do A, then B becomes possible, which opens up C, which gets me to my goal. We'll explore deep agents in Chapter 7, but for now, shallow planning is plenty powerful.

The planning step is where the agent's thought becomes visible. The agent explicitly articulates its thought process: "The user asked about mitosis. I should search the study materials for information about mitosis before answering." This makes the decision transparent and provides a hook for improvement. If the agent is making bad decisions, you can see exactly why and adjust the prompt or the available actions.

Action: Using Tools and Taking Steps

Actions are how agents affect the world. For a customer support agent, actions might include looking up an order, sending an email, or creating a support ticket. For a research agent, actions might include searching a database, reading a document, or generating a summary. For a code agent, actions might include running tests, searching documentation, or modifying files.

Actions in AI agents are typically implemented as tool calls. The agent decides it needs to use a tool, specifies the tool name and parameters, and executes it. The tool returns results, which the agent observes and incorporates into its next decision. This tool-use pattern is fundamental to how modern agents work, and we'll dive deep into it shortly.

Not all actions involve external tools. Sometimes the action is generating text: writing an explanation, composing an email, creating a summary. The distinction is whether the action produces observable effects beyond just generating tokens. If it does, it's usually implemented as a tool call. If it doesn't, it's just the final output.

Observation: Seeing Results and Adapting

After taking an action, the agent needs to observe the results. What did that tool return? Did it find the information I was looking for? Did it succeed or fail? Is this result helpful for answering the user's question? Observation closes the loop and feeds into the next round of perception and planning.

Good agents don't just blindly execute actions. They evaluate the results and adapt. If a search returns no results, the agent might reformulate the query and search again. If a tool fails with an error, the agent might try a different tool or ask the user for clarification. If the results contradict each other, the agent might search for more information to resolve the conflict.

This is where multistep reasoning becomes powerful. The agent isn't locked into a predetermined sequence. It can branch based on what it observes. Search found nothing? Try a different approach. Tool returned unexpected results? Investigate further. Answer looks wrong? Double-check with another source. This adaptability is what makes agents more robust than workflows.

The Feedback Loop

Perception, planning, action, and observation form a loop. The agent perceives the current state, plans what to do, takes an action, observes the result, and uses that observation to inform its next perception. This continues until the agent decides it has enough information to answer the user's question or determines it can't make further progress.

The termination condition is important. Agents need to know when to stop. Sometimes this is obvious: the user asked a question and you have a complete answer. Sometimes it's less clear: you've searched multiple sources and found conflicting information. The agent needs to recognize when it's done or when it's stuck and should ask for help.

In practice, most agent frameworks implement maximum iteration limits to prevent infinite loops. If the agent hasn't reached a conclusion after 10 or 15 iterations, it's probably stuck and should terminate. This is a safety mechanism, but ideally your agent design should lead to natural termination points well before hitting that limit.

Planning and Reasoning

The heart of an agent is its ability to reason. This is what separates agents from simpler systems. Reasoning is how the agent decides what to do next, evaluates whether its actions worked, and determines when it's done. Let's break down how reasoning works in practice.

ReAct Pattern Implementation

The ReAct pattern structures the agent's reasoning as an alternating sequence of thoughts and actions. The agent thinks about what to do, takes an action, observes the result, thinks about what that result means, and repeats. This explicit interleaving makes the reasoning process visible and debuggable.

Here's how it works in detail. The agent receives a task or question. It generates a thought: "I need to find information about *x*." Based on that thought, it decides on an action: "Search the knowledge base for *x*." It executes that action and receives an observation: "Found three documents mentioning *x*." It processes that observation with another thought: "These documents contain relevant information; I should read them." It takes another action: "Retrieve the full content of these documents." And so on.

The key is that thoughts and actions are explicitly separated and labeled. The agent is doing things, sure, but it's also articulating *why* it's doing them. Here's what a complete ReAct trace looks like:

```
Question: What is the capital of France?
Thought: This is a straightforward factual question about geography.
I know the answer from my training data without needing to use tools.
Final Answer: The capital of France is Paris.
```

For a question that requires tool use, the trace is more complex:

```
Question: What's the current weather in Tokyo?
Thought: I need current weather data for Tokyo. This requires
real-time information that I don't have in my training data.
I should use the weather_api tool.
Action: weather_api
Action Input: {"location": "Tokyo", "units": "celsius"}
Observation: {"temperature": 18, "conditions": "partly cloudy",
"humidity": 65, "wind_speed": 12}
Thought: The API returned complete weather information for Tokyo.
I now have everything needed to answer the user's question.
Final Answer: The current weather in Tokyo is 18°C and partly cloudy,
with 65% humidity and winds at 12 km/h.
```

This serves multiple purposes. It helps the language model stay on track because explicitly stating the goal and progress reinforces what it's trying to accomplish. It makes debugging easier because you can see exactly where the reasoning went wrong. And it enables programmatic intervention because you can parse the thought and adjust the agent's behavior.

Reasoning Models

The ReAct pattern asks an agent to reason explicitly before acting. Modern *reasoning models*, such as OpenAI's GPT-5.x and Anthropic's Claude with extended thinking, take this further. Rather than producing explicit thought-action-observation traces in the output, they perform extended internal reasoning before generating a response. You don't see the thinking process, but the model has worked through the problem more thoroughly than a standard completion would allow.

For most agent applications, standard frontier models handle ReAct-style reasoning well without any special prompting. Reasoning models become worth considering when your tasks require complex multistep logic, mathematical reasoning, or careful planning where working through the problem is half the challenge. The trade-off is latency and cost, since reasoning models take longer and cost more. Use them when the quality improvement justifies it, not by default.

Agent Engineering

Here's a shift that's happening right now in how we think about building agents. For years, we've talked about *prompt engineering*, the art of crafting the perfect instructions to get an LLM to do what you want. That's still important, but it's not enough. When you're building production agents, prompt engineering is maybe 20% of the work. The other 80%? That's agent engineering.

> **Building reliable agents requires a new approach, one that combines product, engineering, and data science thinking. We call this discipline *agent engineering* - the iterative process of refining non-deterministic LLM systems into reliable experiences.**
>
> —***LangChain blog, https://www.blog.langchain.com/series-b/***

Agent engineering is about designing the scaffolding around your agent. It's the systems and patterns that make agents reliable, observable, and controllable. You're not just writing a clever prompt and hoping for the best. You're building infrastructure that handles the messy reality of production: retries when tools fail, guardrails that prevent bad behavior, monitoring that shows you what's actually happening, and control flow that keeps the agent on track even when things go sideways.

Think about it this way. Your prompt tells the agent what to do. Agent engineering is everything that makes sure it actually does it reliably, that you can see what it's doing, that it fails gracefully when something breaks, and that it stays within acceptable boundaries. It's the difference between a demo that works on your laptop and a system that runs in production serving real users.

The core concept here is the *agent harness*. The harness is the infrastructure layer that wraps around your agent's reasoning loop. It's not part of the agent's intelligence; it's the system that manages the agent's execution. The harness handles retries when tools time out, implements rate limiting to prevent runaway costs, enforces maximum iteration limits so agents don't loop forever, logs every step for debugging and monitoring, validates tool outputs before they reach the agent, and implements guardrails that prevent dangerous actions.

Here's what this looks like in practice. Your agent wants to call a tool. Before that happens, the harness checks: Is this within our rate limits? Has this agent already made too many calls? Is this tool allowed in this context? The tool executes. The harness catches failures and implements retry logic with backoff. The tool returns results. The harness validates the output format, checks for sensitive data that needs redaction, and logs everything. Only then does the result go back to the agent. The agent never sees any of this infrastructure. It just works in a world where tools are reliable, costs are controlled, and dangerous actions are prevented.

This is why frameworks matter. They're not just about convenience. Good frameworks provide the harness infrastructure so you don't have to build it yourself. They've thought through the failure modes. They've implemented the retry logic. They've built the observability tools. You focus on the agent's reasoning and tools. The framework provides the harness.

Introduction to LangChain

LangChain is the most popular framework for building LLM applications, and for good reason. It provides abstractions for common patterns, integrations with many tools and services, and a large community. But frameworks are always a trade-off. They give you speed and convenience at the cost of some flexibility and the risk of framework lock-in. Let's talk about LangChain honestly: what it's good for, what the concerns are, and how to use it effectively.

LangChain's Philosophy and Design

LangChain's core philosophy is composability. The framework provides modular components that you can mix and match to build applications. Need to load documents? There's a document loader. Need to split them into chunks? There's a text splitter. Need to embed and store them? There are integrations with every major vector database. Need to create an agent? There's an agent builder. You compose these pieces together to build your application.

This composability is powerful because it means you're not writing everything from scratch. Common patterns are already implemented. Integrations with external services are handled. Error cases are thought through. You can go from idea to working prototype much faster than if you were building everything yourself.

The framework is organized into layers. At the bottom, you have components: individual pieces such as embeddings, vector stores, and tools. In the middle, you have chains: sequences of components that accomplish a specific task. At the top, you have agents: systems that can dynamically decide which chains or tools to use. Each layer builds on the one below it, and you can work at whatever level of abstraction makes sense for your use case.

Component Architecture

The component architecture is what makes LangChain flexible. Each component has a standard interface, which means you can swap implementations without changing your application logic. Using OpenAI embeddings today but want to switch to Cohere tomorrow? Change one line. Using Pinecone for your vector

database but want to try Qdrant? Swap out the vector store component. The rest of your code stays the same.

Here's what that looks like in practice:

```
from langchain_openai import OpenAIEmbeddings
from langchain_cohere import CohereEmbeddings
from langchain_community.vectorstores import Pinecone, Qdrant

# Using OpenAI embeddings with Pinecone
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
vector_store = Pinecone.from_documents(docs, embeddings,
index_name="my-index")

# Switch to Cohere embeddings with Qdrant - only 2 lines change
embeddings = CohereEmbeddings(model="embed-english-v3.0")
vector_store = Qdrant.from_documents(docs, embeddings,
location=":memory:")

# The rest of your code using vector_store.similarity_search() stays
the same
```

This abstraction has costs. Sometimes the standard interface doesn't expose all the features of a particular implementation. Sometimes the abstraction leaks and you need to understand the underlying service anyway. Sometimes the framework makes assumptions that don't fit your use case and you have to work around them. But for most applications, the benefits outweigh the costs. You're trading some control for a lot of convenience.

NOTE **You can find all of this chapter's non-StudyBuddy code in the Jupyter notebook** `chapter03.ipynb`**, available in this book's GitHub repo.**

Why Abstractions Matter (and the Concerns)

Let's be real: abstractions hide complexity, and hidden complexity can cause problems. When you use LangChain, you're trusting the framework to handle details you're not seeing. That's fine when things work, but when they don't, you're debugging code you didn't write and might not fully understand. This is the classic framework problem.

The framework lock-in concern is also legitimate. Once you build your application on LangChain, switching to something else is hard. Your code is shaped by LangChain's abstractions. If the framework makes decisions you disagree with or doesn't support something you need, you're stuck. This is the trade-off with any framework, not just LangChain.

That said, LangChain is open source and widely used. The code is readable. The community is large. If you need to drop down to lower levels or fork the framework, you can. The abstractions are usually pretty thin, so you're not that far from the underlying services. And for most applications, the productivity gain from using the framework far exceeds the risk of lock-in.

The key is using frameworks intentionally. Understand what they're doing for you. Read the source when you need to. Don't treat them as magic black boxes. And be willing to drop down to lower levels or build custom solutions when the framework doesn't fit. Use LangChain where it helps; build custom solutions where it doesn't.

The create_agent API

LangChain 1.0 introduced a new, simplified API for creating agents. The old approach used functions such as `create_react_agent` that are now deprecated. The new approach is cleaner and more flexible. Here's what it looks like:

```
from langchain_openai import ChatOpenAI
from langchain.agents import create_agent
from langchain.tools import tool

# Define a simple tool
@tool
def calculator(expression: str) -> str:
    """Evaluate a mathematical expression. Input should be a valid
Python expression."""
    try:
        result = eval(expression)
        return f"The result is {result}"
    except Exception as e:
        return f"Error: {str(e)}"

# Create the agent
tools = [calculator]
agent = create_agent(
    model="gpt-4o-mini",
    tools=tools,
    system_prompt="You are a helpful assistant that can do math
calculations."
)

# Invoke the agent
response = agent.invoke({
    "messages": [{"role": "user", "content": "What is 15% of 240?"}]
})

# Extract and display the answer
```

```
for msg in response["messages"]:
    if msg.type == "ai" and msg.content and not getattr(msg, 'tool_
calls', []):
        print(f"Answer: {msg.content}")
```

The `create_agent` function takes a language model, a list of tools, and optionally a system prompt. It returns an agent that can reason about which tools to use and execute them. Under the hood, it's setting up the ReAct loop, formatting the prompts, and handling the tool execution. You get a working agent with just a few lines of code.

This is a high-level abstraction. You're not managing the prompt structure, the parsing of tool calls, or the execution loop. LangChain handles all of that. This is great for getting started quickly, but it also means you have less control over the details. For most applications, that's fine. You want the agent to work, not to fiddle with prompt engineering. But if you need fine-grained control, you'll use LangGraph instead, which we'll cover in Chapter 4.

When to Use Frameworks vs. Custom Builds

Here's the decision framework: use LangChain when you're building something that fits its patterns and you want to move fast. Building a RAG application? LangChain has everything you need. Creating an agent that uses tools? `create_agent` gets you there in minutes. Integrating with standard services? LangChain's integrations handle the details.

Build custom solutions when you need behavior that doesn't fit the framework's abstractions. When performance is critical and you need to optimize every detail. When you're doing research and need to experiment with novel patterns. When you have requirements that the framework doesn't support. Or when you're building something so specific that the framework's generality is getting in the way.

In practice, most applications use a mix. Use LangChain for the standard parts, such as document loading, embedding, vector stores, basic agent setup. Build custom solutions for the parts that are unique to your use case, such as specialized tools, custom reasoning patterns, and domain-specific logic. This hybrid approach gives you the benefits of both: speed from the framework and flexibility where you need it.

Tool Use and Function Calling

Tools are how agents interact with the world beyond generating text. They're the bridge between the LLM's reasoning and actual effects. Understanding how tool use works is fundamental to building capable agents.

How Tools Work in Agentic Systems

A tool is a function that the agent can call to accomplish a specific task. The agent doesn't execute the function directly. Instead, it generates a structured request to call the function, the framework executes it, and the result comes back to the agent. The agent then continues reasoning with that new information.

This indirection is important. The LLM can't actually search databases or call APIs or read files. It can only generate text. Tools let you expose these capabilities to the agent by defining interfaces the agent can invoke. The agent generates text that describes what function to call and with what parameters. Your code parses that, executes the function, and returns the results.

Modern LLMs have built-in support for function calling. You describe available functions in a schema, include that in the prompt, and the model generates structured requests to call those functions. This is much more reliable than trying to parse free-form text. The model knows it needs to generate valid function calls, not just talk about calling functions.

Defining Tools in LangChain

LangChain makes tool definition straightforward. You create a function, decorate it with tool metadata, and pass it to the agent. The framework handles schema generation, prompt formatting, and execution. Here's what that looks like in practice:

```
from langchain.tools import tool
import requests
@tool
def get_weather(city: str, units: str = "celsius") -> str:
    """
    Get current weather information for a city.

    Args:
        city: Name of the city (e.g. 'Tokyo', 'New York')
        units: Temperature units, either 'celsius' or 'fahrenheit'
    """
    # This is a simplified example - real implementation would
call an API
    api_url = f"https://api.weather.com/current?city={city}&units={units}"
    response = requests.get(api_url)

    if response.status_code == 200:
        data = response.json()
        return f"Temperature: {data['temp']}°, Conditions:
{data['conditions']}"
    else:
        return f"Error: Could not fetch weather for {city}"
```

You define a tool as a Python function with type hints and a docstring. The docstring is crucial because it gets passed to the LLM as the description of what the tool does. The LLM uses this description to decide when to call the tool. If your description is vague or inaccurate, the agent will use the tool incorrectly or not at all.

The function signature defines the parameters. Type hints tell the framework what types to expect, which get encoded in the schema. The framework validates parameters before execution, so you don't have to worry about the agent passing invalid data. If validation fails, the error gets returned to the agent, which can try again with correct parameters.

Here's a more complex tool that demonstrates multiple parameters and error handling:

```
@tool
def search_database(query: str, max_results: int = 5, filter_type: str =
None) -> str:
    """
    Search a database for relevant records.

    Args:
        query: The search term or phrase to look for
        max_results: Maximum number of results to return (1-20)
        filter_type: Optional filter - 'recent', 'popular', or
'archived'
    """
    # Validate inputs
    if max_results < 1 or max_results > 20:
        return "Error: max_results must be between 1 and 20"

    if filter_type and filter_type not in ['recent', 'popular',
'archived']:
        return "Error: filter_type must be 'recent', 'popular', or
'archived'"

    # Perform search (simplified)
    results = perform_db_search(query, max_results, filter_type)

    if not results:
        return f"No results found for '{query}'"

    # Format results
    formatted = f"Found {len(results)} results for '{query}':\n"
    for i, result in enumerate(results, 1):
        formatted += f"\n{i}. {result['title']} - {result['summary']}"

    return formatted
```

Tool Selection Mechanisms

When an agent has multiple tools available, it needs to decide which one to use. This decision happens in the reasoning step. The agent looks at the current task, considers the available tools and their descriptions, and chooses the one that seems most appropriate.

Tool selection quality depends heavily on tool descriptions. The LLM can't read your code, so it only knows what you tell it in the description. If you have a tool called `search_documents` and another called `search_database`, the agent needs clear descriptions to know which one to use. "Search documents" and "Search database" aren't enough. You want "Search the indexed study materials for information about a topic" versus "Search the structured database for user records and metadata."

Here's an example showing good tool descriptions:

```
@tool
def search_vector_db(query: str) -> str:
    """
    Search semantic document store using vector similarity.
    Use this for finding conceptually related information in
unstructured documents.
    Best for: research questions, topic exploration, finding relevant
passages.
    """
    # Implementation
@tool
def query_sql_db(sql_query: str) -> str:
    """
    Execute SQL query on structured database.
    Use this for precise lookups of structured data: user info,
transactions, etc.
    Best for: finding specific records, aggregating data,
checking status.
    """
    # Implementation
```

Tool selection also depends on the agent's reasoning. A well-prompted agent will think through which tool makes sense before calling it. The thought might say: "The user wants to know about photosynthesis, which should be in the study materials. I'll use search_vector_db." This explicit reasoning improves tool selection because the agent is forced to justify its choice.

Parameter Extraction from Natural Language

Once the agent decides to use a tool, it needs to extract the right parameters from the user's question. This is where function calling really shines. For example,

if the agent decides to search, it generates a structured search call with specific arguments.

If the user asks, "What happens during mitosis?" the agent might generate a tool call such as

```
search_documents(query="mitosis")
```

If they ask, "Find me information about cell division, specifically the mitosis phase," the agent might extract this:

```
search_documents(query="mitosis cell division phases")
```

The agent is interpreting the user's natural language and translating it into structured function parameters.

This works because the LLM has been trained on function-calling patterns and knows how to map natural language to structured formats. The schema you provide guides this extraction. If your tool parameter is called `query` and is described as "the search term to look for," the agent knows to extract relevant keywords from the user's message and pass them as that parameter.

Error Handling in Tool Execution

Tools can fail. The database might be unavailable. The search might return no results. The API might rate-limit you. Your agent needs to handle these failures gracefully.

LangChain's tool system returns errors to the agent as observations. If a tool fails, the error message becomes part of the reasoning loop. The agent sees something like "Error: Database connection timeout" and can decide what to do next. Maybe try again. Maybe try a different tool. Maybe tell the user there's a problem.

Good tool implementations return informative errors. Don't just return "Error" or an exception message. Return something the agent can work with:

```
@tool
def search_api(query: str) -> str:
    """Search an external API for information."""
    try:
        response = requests.get(f"https://api.example.com/search?q=
{query}")

        if response.status_code == 429:
            return "Rate limit exceeded. Please wait a moment and try
again."

        if response.status_code == 404:
            return f"No results found for '{query}'. Try a different
search term."
```

```
        if response.status_code != 200:
            return f"API error (status {response.status_code}). Try
again later."

        results = response.json()
        if not results:
            return f"Search for '{query}' returned no results. Try
broader terms."

        return format_results(results)

    except requests.ConnectionError:
        return "Cannot connect to search API. Check your network
connection."
    except Exception as e:
        return f"Unexpected error during search: {str(e)}"
```

This gives the agent context for making a good next decision. You can also implement retry logic in your tools. If a tool fails with a transient error, retry a few times before returning the error to the agent. If a search returns no results, you might automatically try a relaxed version of the query. The goal is to handle routine failures at the tool level so the agent only sees errors when there's genuinely a problem it needs to reason about.

Context Engineering in Practice: Managing Agent State

Managing context is different for agents than for simple RAG systems. Agents maintain state across multiple steps, use tools that produce results, and generate thoughts. All this needs to fit into the context window. Understanding how to manage agent context is crucial for building systems that actually work.

Managing Agent Context vs. RAG Context

In basic RAG, context management is straightforward. You retrieve relevant documents, stuff them into the prompt, and generate a response. The context window contains the user's question, the retrieved documents, and any system instructions. Simple.

Agents are more complex. The context window needs to include the user's original question, the conversation history, the agent's thoughts, tool call results, available tool descriptions, and instructions for how to use tools. All of this competes for limited space. As the agent takes more actions, the context grows. Eventually you hit the limit and need to decide what to keep and what to discard.

The key difference is that agent context is dynamic and cumulative. Each reasoning step and tool call adds to the context. You can't just replace context

like you do when retrieving new documents in RAG. You need the history to maintain continuity. But you also can't keep everything forever. This is the context management problem for agents.

What Goes in the Context Window for Agents

A typical agent context contains several distinct pieces. At the top, you have system instructions that describe what the agent is and how it should behave. These are usually static across all requests.

Next comes the tool definitions. Each available tool needs a name, description, and parameter schema. The agent uses these to decide which tools to call. This section can get large if you have many tools, which is why agent frameworks typically limit you to a reasonable number of tools at once.

Then you have the conversation history and thoughts. This includes the user's messages, the agent's thoughts, actions taken, and results observed. This is the dynamic part that grows with each iteration. It's also the most important part because it contains the state of the agent's reasoning.

Finally, you might have retrieved context from previous tool calls. If the agent searched a database and got results, those results need to be in context so the agent can use them. This is similar to RAG context, but it's the product of the agent's own actions rather than predetermined retrieval.

When context grows too large, you have several options. Summarization condenses history into key points. Selective retention keeps recent interactions in detail while older parts exist in summary form. External memory systems store information in databases rather than the context window. We'll explore this in Chapter 6. The key is designing your tools to return concise results and keeping thoughts focused on decisions rather than verbose explanations.

Avoiding Context Overload

Context overload happens when you try to fit too much into the window and the agent gets confused. More information isn't always better. The agent performs best when it has focused, relevant context. A thousand tokens of highly relevant information beats ten thousand tokens of mixed relevance.

Design your tools to return concise results. If a search returns ten documents, don't dump all ten into context. Return snippets or summaries. Let the agent ask for more detail if needed. This keeps context manageable and forces the agent to be selective about what it retrieves.

Keep thoughts concise. You want enough detail to understand the agent's thinking, but you don't need verbose explanations. "I need to search for mitosis" is plenty. You don't need "I am going to search the study materials database for information about mitosis because the user asked about it and that's where relevant biology content would be stored." Shorter is better.

Monitor context usage and set limits. Most frameworks let you track how much of the context window you're using. If you're consistently hitting 90% utilization, you probably need to trim something. Leave some headroom for the agent's response generation. If the context is completely full, there's no space left for the agent to think and generate output.

Working with Qdrant

In Chapter 2, we built a simple in-memory vector database to understand how RAG works. That was great for learning, but production systems need something more robust. Qdrant (pronounced *quadrant*) is an open-source vector database that's fast, feature-rich, and has excellent integration with LangChain. We're switching to it for StudyBuddy v3.

Why We're Switching from a Custom Vector Database

Our custom vector database from Chapter 2 worked fine for the learning exercise, but it has limitations. It's in-memory only, so data disappears when the process stops. It uses brute-force search, which doesn't scale. It has no filtering capabilities. It doesn't handle concurrent access. These might not matter for a tutorial, but they absolutely matter for real applications.

Qdrant solves all of these problems. It's a proper database with available persistent storage. It uses HNSW indexing for fast approximate nearest neighbor search. It supports metadata filtering, so you can narrow searches to specific documents or time ranges. It handles concurrent queries efficiently. And it integrates seamlessly with LangChain, so you don't have to write much code to use it.

Qdrant Architecture and Design

Qdrant is built for vector search at scale. It stores vectors in collections, which are like tables in a traditional database. Each collection has a dimension size that matches your embedding model. Within a collection, you have points, which are individual vectors along with their payloads. The payload is where you store metadata and the original text.

The indexing uses HNSW, which we discussed in Chapter 2. This gives you fast searches even with millions of vectors. Qdrant also supports filtering on payload fields, which is incredibly useful. You can search for "mitosis" but only in documents tagged as "biology" or only in documents uploaded after a certain date. This combination of vector similarity and metadata filtering is what makes Qdrant powerful for real applications.

In-Memory Mode for Development

Qdrant has two modes: in-memory and persistent. In-memory mode runs entirely in RAM with no disk storage. It's perfect for development because it's fast, requires zero configuration, and cleans up automatically when your process stops. You don't need to install a database server or manage files. Just import the library and start using it.

Here's how to set up Qdrant in memory:

```
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams

# Create in-memory client
client = QdrantClient(":memory:")

# Create a collection
client.create_collection(
    collection_name="documents",
    vectors_config=VectorParams(
        size=1536,  # Must match your embedding model dimension
        distance=Distance.COSINE  # Cosine similarity for text
embeddings
    )
)

# Add some vectors
from qdrant_client.models import PointStruct
points = [
    PointStruct(
        id=1,
        vector=[0.1, 0.2, ...],  # 1536-dimensional vector
        payload={"text": "Document content", "source": "file.txt"}
    )
]
client.upsert(collection_name="documents", points=points)
```

The trade-off is that your data disappears when the process ends. For development, that's usually fine. You're reindexing test data anyway. And if you need to preserve data between runs during development, you can export your collection to a snapshot and reload it later. But for production, you'll use persistent mode.

Server Mode for Production

Production deployments run Qdrant as a server. You can run it in Docker, deploy it to a VM, or use Qdrant Cloud, its managed service. The server mode uses persistent storage, so your vectors survive restarts. It supports replication for

high availability. It can handle concurrent connections from multiple clients. All the things you need for production.

The good news is that your code is the same in both modes. You just change the connection configuration. For in-memory, you create a local client. For server mode, you provide a URL and an API key:

```
# In-memory mode (development)
client = QdrantClient(":memory:")

# Local server mode (testing)
client = QdrantClient(host="localhost", port=6333)

# Cloud mode (production)
client = QdrantClient(
    url="https://your-cluster.qdrant.io",
    api_key="your-api-key"
)
```

The rest of your code—creating collections, inserting vectors, searching—is identical. This makes it easy to develop locally and deploy to production without code changes.

LangChain Integration

LangChain provides a Qdrant integration that wraps the Qdrant client and presents it as a standard vector store. This means you can use Qdrant with all of LangChain's retrieval patterns, chain it with other components, and swap it out for other vector stores if needed. The integration handles batching, error handling, and connection management.

Using the integration is straightforward:

```
from langchain_community.vectorstores import Qdrant
from langchain_openai import OpenAIEmbeddings
from qdrant_client import QdrantClient

# Set up embeddings and Qdrant client
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
client = QdrantClient(":memory:")

# Create LangChain vector store
vector_store = Qdrant(
    client=client,
    collection_name="my_documents",
    embeddings=embeddings
)

# Add documents
```

```
from langchain.schema import Document
docs = [
    Document(page_content="First document", metadata={"source":
"doc1.txt"}),
    Document(page_content="Second document", metadata={"source":
"doc2.txt"})
]
vector_store.add_documents(docs)

# Search
results = vector_store.similarity_search("first", k=2)
for doc in results:
    print(f"{doc.metadata['source']}: {doc.page_content}")
```

You create a Qdrant client, wrap it in LangChain's Qdrant vector store, and use it like any other vector store. You can add documents, delete them, search by similarity, and filter by metadata—all through a consistent interface. This abstraction makes your code portable and easier to test.

Creating Collections and Configuring Retrievers

Collections in Qdrant are created with a dimension size and optional configuration. The dimension must match your embedding model. OpenAI's `text-embedding-3-small` produces 1,536-dimensional vectors, so your collection needs to be configured for 1,536 dimensions. If you try to insert vectors of the wrong size, you'll get an error.

Retrievers are how you query the collection. LangChain's retriever interface standardizes querying across different vector stores:

```
# Create a retriever from the vector store
retriever = vector_store.as_retriever(
    search_type="similarity",
    search_kwargs={"k": 5}  # Return top 5 results
)

# Use the retriever
docs = retriever.get_relevant_documents("What is photosynthesis?")

# The retriever can also filter by metadata
retriever_filtered = vector_store.as_retriever(
    search_type="similarity",
    search_kwargs={
        "k": 3,
        "filter": {"source": "biology_textbook.pdf"}
    }
)
```

The retriever handles the actual search and returns `Document` objects that include both the text and metadata. You can configure how many results to return, what type of search to use, and any metadata filters to apply.

Building StudyBuddy v3

Time to put everything together. We're going to rebuild StudyBuddy as a proper agent using LangChain's `create_agent` API, Qdrant for vector storage, and real document indexing. This is where v2's basic RAG becomes v3's intelligent tutoring agent.

Where We Left Off

StudyBuddy v2 had RAG capabilities but no document indexing. We embedded some sample text directly in the code and used a custom in-memory vector database. It could answer questions based on that hard-coded content, but you couldn't upload your own study materials. It also had no reasoning; it just retrieved and answered every time, regardless of whether retrieval was needed.

That was fine for learning RAG fundamentals, but it's not a real tutoring system. Students need to upload their own textbooks, notes, and study guides. The system needs to reason about whether it should search before answering or whether it already knows enough. And it needs production-grade infrastructure that actually persists data.

What We're Adding

StudyBuddy v3 adds document upload and indexing. Students can work with any text-based study materials, and we'll chunk them, embed them, and store them in Qdrant. The agent can then search these materials when answering questions. This implementation uses the 12 Sherlock Holmes stories from *The Adventures of Sherlock Holmes* as the document set, demonstrating how the system works with literary texts. The same patterns work for any domain: textbooks, course notes, research papers, historical documents—whatever students need to study.

We're also making StudyBuddy a real agent using LangChain's `create_agent`. The agent will have tools for searching materials and retrieving additional context. It decides when to use these tools based on the question. If a student asks a simple question the agent already knows, it can answer directly. If they ask about specific study materials, the agent will search before answering.

We're switching to Qdrant for vector storage. We'll start with in-memory mode for simplicity, but the code will be ready to switch to persistent mode when we deploy in later chapters. And we're using LangChain's document loaders and text splitters to handle the indexing pipeline properly.

Architecture

StudyBuddy v3 uses a hybrid architecture:

- **Backend:** Python/FastAPI with a LangChain agent, running on port 8000
- **Frontend:** Next.js with React, running on port 3000
- **Communication:** Frontend proxies `/api/*` requests to the backend

This separation lets you iterate on the UI independently from the AI logic and matches how production apps are typically structured.

Implementation Walkthrough

Let's build this step by step. First, we'll set up Qdrant and create our collection. Then we'll implement document processing and indexing. Next, we'll define tools for the agent. Finally, we'll create the agent itself, index all the stories from the `documents` directory, and wire everything together.

Setting up Qdrant with LangChain is straightforward. We'll use the in-memory client for now. Here's the code:

```
from langchain_qdrant import QdrantVectorStore
from langchain_openai import OpenAIEmbeddings
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams

# Initialize embeddings
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

# Create in-memory Qdrant client
client = QdrantClient(":memory:")

# Collection name
COLLECTION_NAME = "study_materials"

# Create collection with proper dimensions (1536 for
text-embedding-3-small)
client.create_collection(
    collection_name=COLLECTION_NAME,
    vectors_config=VectorParams(size=1536, distance=Distance.COSINE)
)

# Create LangChain vector store wrapper
vector_store = QdrantVectorStore(
    client=client,
    collection_name=COLLECTION_NAME,
    embedding=embeddings
)
```

This creates an in-memory Qdrant instance and wraps it in LangChain's `QdrantVectorStore`. The vector store handles embedding documents and searching for us. We're using cosine distance, which is standard for text embeddings.

Next, we need document processing and indexing. We'll use LangChain's document loaders and text splitters to handle this properly:

```
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import TextLoader

def index_document(file_path: str, doc_name: str):
    """Load, chunk, and index a document."""
    loader = TextLoader(file_path)
    documents = loader.load()

    # Split into chunks
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=500,
        chunk_overlap=50,
        length_function=len
    )
    chunks = text_splitter.split_documents(documents)

    # Add metadata
    for chunk in chunks:
        chunk.metadata['source'] = doc_name

    # Index in Qdrant
    vector_store.add_documents(chunks)

    return len(chunks)
```

The document loader handles extracting text from text files (such as `.txt` or `.md` files). The text splitter chunks the content using recursive character splitting, which tries to split on natural boundaries such as paragraphs and sentences. We add the document name to the metadata so we can track which chunks came from which source.

Now we'll define tools for the agent. We need a search tool that queries the vector store:

```
from langchain.tools import tool

@tool
def search_materials(query: str) -> str:
    """
    Search the indexed study materials for information about a topic.
    Use this when you need to find specific information from the
    student's uploaded study materials.
```

```
    Args:
        query: The search term or question to look up
    """
    results = vector_store.similarity_search(query, k=3)

    if not results:
        return "No relevant information found in study materials."

    # Format results with source attribution
    formatted = []
    for doc in results:
        source = doc.metadata.get('source', 'Unknown')
        formatted.append(f"[From {source}]:\n{doc.page_content}")

    return "\n\n".join(formatted)
```

The `tool` decorator tells LangChain this function is a tool. The docstring is crucial: it gets passed to the LLM so the model knows when and how to use this tool. We search for the top three most similar chunks and format them with source attribution so the agent knows where the information came from.

Now we can create the agent. We'll use `create_agent` with a system prompt that directs the agent to search for questions about the study materials:

```
from langchain.agents import create_agent

# System prompt
SYSTEM_PROMPT = """You are StudyBuddy, an AI tutoring assistant helping
students study Sherlock Holmes stories.

IMPORTANT: The student has uploaded study materials (Sherlock Holmes
stories) that you MUST search
before answering questions about characters, plots, or events. Always
use the search_materials tool
first for any question about:
- Characters (Holmes, Watson, Irene Adler, Moriarty, etc.)
- Story plots or mysteries
- Specific events or quotes
- Anything related to the study materials

Only answer from general knowledge for questions completely unrelated to
the study materials
(like math questions or general facts).

When you search, cite which story the information comes from."""

# Create tools list
tools = [search_materials]
```

```
# Create the agent
agent = create_agent(
    model="gpt-4o-mini",
    tools=tools,
    system_prompt=SYSTEM_PROMPT
)
```

The system prompt is doing a lot of work here. It explicitly tells the agent when to use tools for any question about the study materials. This ensures the agent searches the indexed documents rather than relying solely on its general knowledge.

Now let's index all the documents in the `documents` directory. We'll do this in a background thread so the server starts immediately:

```
from pathlib import Path
import threading

# Track indexing status
indexing_status = {"done": False, "count": 0, "chunks": 0, "current_
file": ""}

def index_all_documents():
    """Index all documents from the documents directory."""
    if indexing_status["done"]:
        return indexing_status["chunks"]

    documents_dir = Path("documents")
    story_files = sorted(
        list(documents_dir.glob("*.txt")) +
        list(documents_dir.glob("*.md"))
    )

    print("Indexing stories...")
    total_chunks = 0
    for filepath in story_files:
        doc_name = filepath.stem.replace("-", " ").replace("_", " ").
title()
        indexing_status["current_file"] = doc_name
        num_chunks = index_document(str(filepath), doc_name)
        total_chunks += num_chunks
        indexing_status["chunks"] = total_chunks
        print(f"Indexed {num_chunks} chunks from {doc_name}")

    indexing_status["done"] = True
    indexing_status["count"] = len(story_files)
    return total_chunks

# Start indexing in background thread on server startup
thread = threading.Thread(target=index_all_documents, daemon=True)
thread.start()
```

The indexing runs in the background, updating the status dictionary as it goes. The frontend can poll the `/api/status` endpoint to show progress to the user.

Now let's look at how we invoke the agent and extract its reasoning:

```
# Question that requires searching
response = agent.invoke({
    "messages": [{"role": "user", "content": "Who is Irene Adler?"}]
})

# The response contains a list of messages
messages = response["messages"]

# Extract reasoning from the message history
reasoning_parts = []
final_answer = ""

for msg in messages:
    msg_type = getattr(msg, 'type', None)

    # Check for tool calls (agent deciding to use a tool)
    if msg_type == 'ai' and hasattr(msg, 'tool_calls') and msg
.tool_calls:
        for tool_call in msg.tool_calls:
            reasoning_parts.append(f"Action: {tool_call['name']}")
            reasoning_parts.append(f"Input: {tool_call['args']}")

    # Check for tool responses
    elif msg_type == 'tool':
        tool_name = getattr(msg, 'name', 'search_materials')
        reasoning_parts.append(f"Observation from {tool_name}:\n{msg
.content[:500]}...")

    # The final AI message is the answer
    elif msg_type == 'ai' and msg.content:
        if not getattr(msg, 'tool_calls', []):
            final_answer = msg.content

print("Reasoning:", reasoning_parts)
print("Answer:", final_answer)
```

LangChain's `create_agent` uses native tool calling rather than text-based reasoning. The message history contains structured data about tool calls:

- **AIMessage with tool_calls:** The agent decides to call a tool.
- **ToolMessage:** The result from executing the tool.
- **AIMessage with content:** The final answer.

We parse this history to extract the agent's reasoning—what tools it called and what it observed—separate from the final answer. This lets us show users what the agent did behind the scenes.

Exposing Agent Reasoning

One of the key features of v3 is transparency. Users can see what the agent is "thinking"—which tools it called, what queries it made, and what results it found. This is exposed through a "Show reasoning" toggle in the UI.

The API returns both the final answer and the thought:

```
return {
    "reply": final_answer,
    "reasoning": "\n\n".join(reasoning_parts) if reasoning_parts
else None
}
```

The frontend displays a collapsible section showing the agent's actions when reasoning is present:

```
// In Message.tsx
const [showReasoning, setShowReasoning] = useState(false)

// After the message content:
{reasoning && (
    <div className="reasoning-container">
        <button
            className="reasoning-toggle"
            onClick={() => setShowReasoning(!showReasoning)}
        >
            {showReasoning ? 'Hide reasoning' : 'Show reasoning'}
        </button>
        {showReasoning && (
            <div className="reasoning-content">
                <pre>{reasoning}</pre>
            </div>
        )}
    </div>
)}
```

This helps users understand how the agent arrived at its answer and builds trust in the system.

The Frontend

The frontend is built with Next.js and React. It polls the `/api/status` endpoint to show indexing progress:

```
interface AgentStatus {
    indexing_complete: boolean
    documents_indexed: number
    chunks_in_db: number
    current_file: string
}
```

The status footer shows real-time progress:

- "Connecting. . ." when waiting for the backend
- "Indexing 'A Scandal in Bohemia'. . . (47 chunks)" during indexing
- "12 stories indexed (589 chunks)" when complete

The `ChatMessage` interface includes an optional `reasoning` field that gets populated when the agent uses tools:

```
export interface ChatMessage {
    id: string
    content: string
    role: MessageRole
    isLoading?: boolean
    reasoning?: string  // Agent thought
}
```

Testing Agentic Behavior

To really see the agent in action, try different types of questions:

Questions that trigger search:

- "Who is Irene Adler?"—Searches materials, finds info from "A Scandal in Bohemia"
- "What happened in The Red-Headed League?"—Searches for plot details
- "How does Holmes solve the speckled band case?"—Searches for story events

Questions answered directly:

- "What's 2 + 2?"—No search needed, answers from general knowledge
- "What's the capital of France?"—General knowledge, no tool call

Watch the thoughts. They'll show you exactly what the agent did at each step. The "Show reasoning" toggle reveals:

- **Action:** Which tool was called
- **Input:** The search query used
- **Observation:** What the tool returned

If the agent is making bad decisions, you can diagnose why from the traces. Maybe the system prompt needs adjustment. Maybe the tool description is unclear. The traces give you the visibility to iterate and improve.

Running Locally

First, make sure you have all your dependencies in `pyproject.toml`:

```
[project]
name = "studybuddy"
version = "3.0.0"
description = "AI-powered study assistant - Chapter 3: The Agent Loop"
requires-python = ">=3.12"
dependencies = [
    "fastapi>=0.115.0",
    "uvicorn[standard]>=0.32.0",
    "python-dotenv>=1.0.0",
    "langchain>=1.0.0",
    "langchain-openai>=1.0.0",
    "langchain-community>=0.4.0",
    "langchain-text-splitters>=1.0.0",
    "langchain-qdrant>=1.0.0",
    "qdrant-client>=1.12.0",
]
```

With that done, from the project root directory (`v3-the-agent-loop`), run:

```
uv sync
```

You need two terminals:

- One terminal runs the backend (port 8000):

```
cd v3-the-agent-loop
uv run uvicorn api.index:app --reload --port 8000
```

- The other terminal sets up the frontend:

```
cd v3-the-agent-loop/frontend
npm install
```

Now, on the frontend terminal, start the dev server (port 3000):

```
npm run dev
```

Visit `http://localhost:3000` to use the app.

Deploying to Vercel

To deploy v3 to Vercel, you first need to update `api/requirements.txt`:

```
fastapi>=0.115.0
uvicorn[standard]>=0.32.0
python-dotenv>=1.0.0
langchain>=1.0.0
langchain-openai>=0.3.0
langchain-community>=0.3.0
langchain-text-splitters>=0.3.0
langchain-qdrant>=0.2.0
qdrant-client>=1.12.0
```

Then you need to update two files in the repo root. First, update `vercel.json` to point to the `v3-the-agent-loop` directory:

```
{
  "version": 2,
  "builds": [
    {
      "src": "v3-the-agent-loop/api/index.py",
      "use": "@vercel/python"
    },
    {
      "src": "v3-the-agent-loop/frontend/package.json",
      "use": "@vercel/next"
    }
  ],
  "routes": [
    { "src": "/api/(.*)", "dest": "/v3-the-agent-loop/api/index.py" },
    { "src": "/(.*)", "dest": "/v3-the-agent-loop/frontend/$1" }
  ],
  "git": {
    "deploymentEnabled": {
      "main": true
    }
  },
  "ignoreCommand": "bash scripts/should-deploy.sh"
}
```

Second, update `scripts/should-deploy.sh` to trigger on v3 (or `vercel.json`) changes:

```
#!/bin/bash
# Deploy if changes are in v3-the-agent-loop or vercel config
# Exit 1 (true) = proceed with build, Exit 0 (false) = skip build
git diff HEAD^ HEAD --name-only | grep -qE "^v3-the-agent-loop/
|^vercel\.json" && exit 1 || exit 0
```

If you haven't done so already, set your OpenAI API key environment variable in Vercel:

```
vercel env add OPENAI_API_KEY
```

Deploy:

```
vercel --prod
```

Or simply push to GitHub. Vercel will automatically deploy when changes are detected in the `v3-the-agent-loop` directory.

Note: Vercel's serverless environment resets between requests, so documents need to be re-indexed on each cold start. For production, consider using Qdrant Cloud for persistent storage.

What's Next

StudyBuddy v3 is a real agentic tutoring system. It can reason about when to search, use tools intelligently, and adapt its behavior based on the question. Users can see the agent's reasoning process, building transparency and trust.

In Chapter 4, we'll rebuild this using LangGraph, which gives us complete control over the agent's state machine. We'll implement custom reasoning patterns, add reflection and confidence scoring, and set up observability with LangSmith. We'll also start building toward the flashcard generation feature that will become central to StudyBuddy in later chapters.

You just built a real agent—a system that reasons, uses tools, and adapts its behavior intelligently. This is the foundation that everything else builds on.

Let's keep building.

CHAPTER

4

Agentic RAG

LangChain's `create_agent` function works well. The agent can decide when to search your study materials versus when to answer from general knowledge. It uses tools, reasons through problems, and gives solid answers. That's a real agent.

But here's the thing: `create_agent` is doing a lot of work behind the scenes. It's making decisions for you about how the agent loop runs, how state flows between steps, and how tools get invoked. For many use cases, that's exactly what you want. Abstractions exist to make common patterns easy.

Sometimes, the abstraction isn't quite right for your use case, and you need more control. You might want to customize the agent's decision-making, add reflection or confidence scoring, or use dynamic retrieval strategies.

LangGraph is a framework that lets you explicitly design your agent as a state machine. Instead of trusting `create_agent` to do the right thing, you'll define exactly what happens at each step. You'll see every decision point. You'll control every transition.

This is also where we make the RAG system truly *agentic*. Instead of blindly retrieving on every query, your agent will analyze questions, decide whether retrieval is needed, adjust its search strategy based on complexity, and evaluate whether it has enough information before answering. It will know when it doesn't know.

We'll also set up LangSmith for observability. You'll be able to watch your agent think in real time, step through its decisions, and debug problems by seeing exactly what happened at each node in the graph. This is essential for building agents you can actually trust in production.

Let's keep building.

Beyond Basic RAG

The RAG system you built in Chapter 2 followed a simple pattern: take the user's query, embed it, find similar chunks in the vector database, stuff those chunks into a prompt, and generate a response. This "retrieve-then-generate" approach works surprisingly well for a lot of use cases. But it has limitations that become obvious when you push it.

Limitations of Static RAG

Static RAG treats every query the same way. Whether you ask "What is the airspeed velocity of an unladen swallow?" or "How many dragons were there in Lord of the Rings?" the system does the same thing: embed, search, retrieve *k* documents, generate. There's no intelligence in the retrieval process itself.

This creates several problems. First, some queries don't need retrieval at all. If someone asks, "What's 2 + 2?" or "How do I write a `for` loop in Python?" searching your knowledge base wastes time and tokens. The LLM already knows the answer. Second, simple queries might need one or two chunks, while complex queries might need ten or fifteen from different parts of your knowledge base. Fixed-*k* retrieval can't adapt. Third, the system has no way to know if the retrieved documents actually answer the question. It retrieves, generates, and hopes for the best.

There's also the problem of query-document mismatch. Users phrase questions differently than documents phrase answers. Someone might ask, "What is the torque spec for tightening a jar of pickles?" when the relevant document talks about "cucumber fermentation standard operating procedures." Basic similarity search might miss this connection entirely.

What Makes RAG Agentic

Agentic RAG adds a reasoning layer around retrieval. Instead of following a fixed pipeline, an agentic RAG system makes decisions at every step. Should I retrieve at all? What query should I use for retrieval? How many documents do I need? Are the results I got sufficient to answer the question? Should I try a different search strategy?

The key insight is that retrieval becomes a *tool* the agent uses rather than a mandatory step in a pipeline. Just as a human researcher decides when to search Google versus when to answer from memory, an agentic RAG system decides when to hit the vector database versus when to respond directly.

This decision-making capability opens up powerful patterns. The agent can reformulate queries to improve retrieval quality. It can decompose complex questions into sub-questions and retrieve answers for each one. It can evaluate whether retrieved documents actually contain the answer and try alternative strategies if they don't. It can express confidence in its answers based on how well the retrieved context supports them.

When You Need Agentic RAG

Not every RAG system needs to be agentic. If your queries are uniform and your retrieval quality is consistently good, static RAG is simpler and faster. Agentic RAG adds latency because of the extra reasoning steps. It adds cost because of the additional LLM calls. It adds complexity because you're building a state machine instead of a pipeline.

You need agentic RAG when query complexity varies significantly, when your knowledge base covers diverse topics with different retrieval needs, when you want the system to acknowledge uncertainty rather than hallucinate, when queries might require multistep reasoning or combining information from multiple sources, or when you need to handle edge cases gracefully rather than failing silently.

For StudyBuddy, agentic RAG makes sense. Students ask everything from simple definitions to complex conceptual questions. Some questions are about topics in the knowledge base; others are general programming questions the LLM can handle directly. We want the system to be smart about when and how it retrieves.

Looking Under the Hood

Before we dive into LangGraph, let's peek behind the curtain at what `create_agent` is doing and what it is not doing. In today's parlance, we would call `create_agent` a simple *agent harness*. The function simply creates an agent loop. Therefore, it is *fully agentic*. The LLM reasons and takes action, over and over. What's missing? There is no fixed workflow. Understanding this will make it clear why you'd want to use LangGraph and when the abstraction is the right choice.

Before we go further, we need to introduce the crucial term *state machine*, because it's central to everything LangGraph does. A state machine is a system

that can be in exactly one of a finite set of states at any given moment and that moves between states in response to specific conditions or inputs. You've encountered state machines your whole life: a traffic light cycles through red, yellow, and green based on a timer; a vending machine moves from "idle" to "item selected" to "payment received" to "dispensing" based on user actions. The logical structure of a state machine maps naturally onto a graph, where nodes represent the states the system can be in and edges represent the transitions between them. That's not a metaphor. It's literally how LangGraph works: nodes are the steps your agent can be at; edges define the conditions under which it moves between them; and state is the data it carries as it travels (see Figure 4.1). With that in mind, let's look at what `create_agent` is actually doing under the hood.

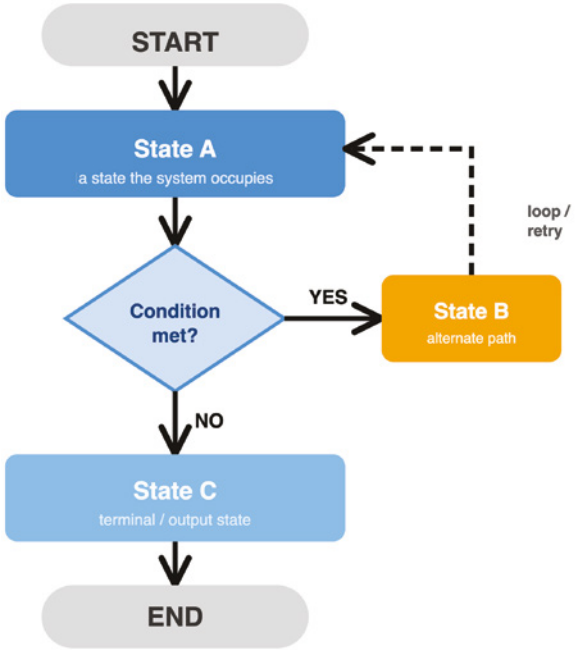

Figure 4.1: An abstract view of a state machine.

What *create_agent* Actually Does

When you call `create_agent` with a model, tools, and a system prompt, LangChain builds a state machine for you. It creates nodes for the LLM to reason and for tools to execute. It creates edges that route between them based on whether the LLM decides to call a tool. It sets up the loop that continues until the LLM produces a final answer without tool calls.

Figure 4.2 shows a simplified view of what that state machine looks like.

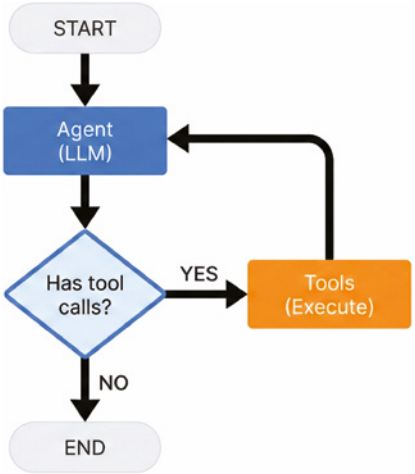

Figure 4.2: An example state machine for an agentic RAG system.

The Agent node runs the LLM with the current messages and available tools. If the LLM outputs tool calls, they get routed to the Tools node for execution. The results come back as tool messages, and the cycle repeats. When the LLM produces a response without tool calls, that's the final answer, and the graph ends.

This is a perfectly reasonable design for many use cases. But notice what you *can't* easily do with this abstraction. You can't add a step between the agent deciding to retrieve and actually retrieving. You can't evaluate the retrieval results before generating a response. You can't route to different nodes based on query complexity. You can't add reflection or self-critique. The structure is fixed in that the LLM decides what to do next, not you. `create_agent` maximizes agency.

When to Drop Down to LangGraph

LangGraph is the framework underneath `create_agent`. It lets you build arbitrary state machines where you define every node, every edge, and every routing decision. You get complete control at the cost of writing more code.

Use LangGraph when you need custom routing logic that goes beyond "did the LLM call a tool?" when you want to add intermediate steps such as query analysis or result evaluation, when you need fine-grained control over state management, when you're building complex multistep workflows, or when you need better observability into what's happening at each step. Essentially, anytime you want to "put your application on rails," building the graph yourself is one way to do that. Moreover, most business value in established companies today is created by building multistep, lower-agency workflows.

Stick with `create_agent` when the default ReAct loop is exactly what you need, when you're prototyping and want to move fast, when simplicity is

more important than customization, or when you don't need intermediate processing steps.

The good news is that the concepts transfer. Once you understand LangGraph, you'll understand `create_agent` better too. And you can always start with the abstraction and drop down to LangGraph when you hit its limits.

Introduction to LangGraph

LangGraph models agents as graphs. Nodes are functions that do work. Edges connect nodes and control flow. State flows through the graph, getting transformed at each node. This mental model is powerful because it makes the agent's behavior explicit and inspectable.

Core Concepts

A LangGraph application has three fundamental pieces: state, nodes, and edges. Let's look at each one.

State is a `TypedDict` that carries information through the graph. Every node receives the current state and returns updates to it. LangGraph merges those updates with the existing state and passes the result to the next node. Here's a state definition for an agentic RAG system:

```
from typing import TypedDict, Annotated
from langgraph.graph import add_messages

class AgentState(TypedDict):
    messages: Annotated[list, add_messages]   # Chat history
    query: str                                # Current query
    complexity: str                           # simple, moderate, complex
    context: list[str]                        # Retrieved documents
    confidence: float                         # Answer confidence
```

The `Annotated[list, add_messages]` syntax is special. It tells LangGraph to *append* new messages to the list rather than replacing it. This is how conversation history accumulates as the agent runs.

NOTE **You can find all of this chapter's non-StudyBuddy code in the Jupyter notebook `chapter04.ipynb`, available in this book's GitHub repo.**

Nodes are functions that take state and return state updates. Nodes are just dictionaries that contain the fields that can be changed. Nodes, in short, do the actual work of your agent: calling LLMs, executing tools, querying vector stores, analyzing queries, or whatever your application needs. Crucially, a node

doesn't need to return the entire state. It only returns the fields it updates, and LangGraph handles the merging automatically.

Edges connect nodes and control the flow through the graph. Simple edges always go from one node to another. Conditional edges branch based on the current state, enabling you to route to different nodes depending on what happened at the previous step. A conditional edge is just a function that reads the state and returns a string naming the next node:

```
def needs_retrieval(state: AgentState) -> str:
    if state["complexity"] == "simple":
        return "generate"  # Skip retrieval
    return "retrieve"      # Do retrieval
```

The conditional edge function returns a string that maps to the next node. In the preceding example, depending on the current value of the state's `complexity` field, the returned string will be either "generate" (meaning, in this example, to skip the retrieval phase entirely) or "retrieve" (meaning that the retrieval node runs next). This is how you implement routing logic: look at the state and decide where to go next.

Building Your First Graph

Let's build a simple working graph to see how these pieces fit together. We need four things: a state definition, a few node functions, a routing function, and the wiring that connects them.

We'll start by defining our state and the LLM we'll use.

```
from langgraph.graph import StateGraph, END
from typing import TypedDict, Literal
from langchain_openai import ChatOpenAI
from dotenv import load_dotenv

class SimpleState(TypedDict):
    question: str
    needs_search: bool
    context: str
    answer: str

load_dotenv()

llm = ChatOpenAI(model="gpt-4o-mini")
```

Our state holds four fields: the original question, a flag for whether we need to search (`needs_search`), space for retrieved context, and, finally, an answer.

Now let's define our first node: the classify node. Its job is to look at the question and decide whether we need to search for more information to answer the question or whether the LLM can answer directly from its existing knowledge.

```
def classify_question(state: SimpleState) -> SimpleState:
    """Decide if we need to search for information."""
    response = llm.invoke(
        f"Does this question need external information? Answer yes/no:
{state['question']}"
    )
    needs_search = "yes" in response.content.lower()
    return {"needs_search": needs_search}
```

This node calls the LLM with a simple prompt asking if external knowledge would help it answer. It returns an update to our state with the decision. Notice that the return value is just the one field that we're updating.

The search node and generate node follow the same pattern: each accepts the full state, does one focused job, and returns only the fields it changed. The search node queries a vector store; the generate node calls the LLM with whatever context is available, whether from search or the default "no additional context" message.

The routing function determines which node comes next after classification:

```
def route_after_classify(state: SimpleState) -> Literal["search",
"generate"]:
    """Route based on classification."""
    if state["needs_search"]:
        return "search"
    return "generate"
```

This is a conditional edge. It reads the `needs_search` field and routes either to `search` or directly to `generate`.

Finally, we assemble everything into the graph:

```
# Build the graph
graph = StateGraph(SimpleState)

graph.add_node("classify", classify_question)
graph.add_node("search", search_info)
graph.add_node("generate", generate_answer)

graph.set_entry_point("classify")
graph.add_conditional_edges("classify", route_after_classify)
```

```
graph.add_edge("search", "generate")
graph.add_edge("generate", END)

# Compile and run
app = graph.compile()
result = app.invoke({"question": "What is the capital of France?"})
print(result["answer"])
```

The flow is: start at `classify`, branch based on our routing function to either `search` or `generate`, then always move from `search` to `generate`, and finally end.

One of LangGraph's best features is built-in visualization. After compiling, you can call `app.get_graph().draw_ascii()` for a terminal-friendly diagram or `app.get_graph().draw_mermaid()` to get a Mermaid diagram you can paste into any compatible viewer. Being able to see your agent's structure is invaluable for debugging and explaining to others what your system does. The graph *is* the documentation.

The Agentic RAG Architecture

Now let's design a proper agentic RAG system. This architecture adds intelligence at every step: analyzing the query before retrieval, adapting the retrieval strategy based on complexity, evaluating results before generating, and reflecting on answer quality. This is a significant step up from the basic retrieve-then-generate workflow.

State Machine Design

Our agentic RAG state needs to track more than just messages. We need to know the original query, the results of our analysis, the retrieved documents, the evaluation scores, and the final response. Here's our state definition:

```
from typing import TypedDict, Annotated, Literal
from langgraph.graph import add_messages

class AgenticRAGState(TypedDict):
    messages: Annotated[list, add_messages]
    query: str
    query_type: str              # factual, conceptual, procedural
    complexity: str              # simple, moderate, complex
    needs_retrieval: bool
    search_queries: list[str]    # Generated search queries
    retrieved_docs: list         # Documents from vector store
    retrieval_sufficient: bool   # Did we get enough?
    response: str
    confidence: float            # 0-1 confidence score
    iteration: int               # Track retrieval iterations
```

This expanded state captures the agent's "working memory" as it processes a query. Each node reads what it needs and writes its outputs. The `iteration` counter is important here because it's how we prevent infinite loops when the retrieval-evaluation-retry cycle keeps coming up short. As you learn in the next section, we cap the number of retries at two, so the system never spins forever.

Node Definitions

Our agentic RAG system has five distinct nodes, each handling one responsibility in the pipeline.

The *analysis node* (`analyze_query_node`) is the brain. It examines the incoming query and decides how to handle it. This node uses the LLM to classify the query along four dimensions: what type of question this is, how complex it is, whether we need to retrieve from the vector store (stored in the `needs_retrieval` flag), and, if so, what search queries would work best to find the relevant content. The analysis node uses a structured prompt asking the LLM to respond in JSON, then unpacks the result into four state fields and initializes our `iteration` counter to zero for tracking retry loops.

That `needs_retrieval` flag is the key element of the analysis node. Before the analysis node existed, the system always retrieved, whether it needed to or not. Now the agent decides. For example, a query such as "What is 2 + 2?" goes straight to generation. By contrast, the query "How does cosine similarity work in the context of embedding search?" goes through retrieval. That decision alone avoids unnecessary vector store queries for many real-world requests.

The *retrieval node* (`retrieve_node`) fetches documents based on the results of the analysis node, but it does so adaptively. Simple queries get two documents; moderate queries get four; complex ones get six. If the analysis generated multiple search queries, the node runs each one and deduplicates the results by hashing document content. So, if two search queries return the same chunk, it only appears once in the retrieved set.

The *evaluation node* (`evaluate_node`) is where our feedback loop comes from. It looks at the retrieved documents and scores how well they can answer the original query by rating relevance, completeness, and overall confidence on a 0–1 scale.

```
sufficient = confidence >= 0.7 or iteration >= 2
```

This single line of code is the heart of the evaluation node. If `confidence` is below 0.7 and we haven't tried twice yet, we'll route back to retrieval for another attempt. Otherwise, the system moves on to generation.

The *generation node* (`generate_node`) creates the final response. It formats all retrieved documents with their source attribution, then instructs the LLM to synthesize rather than just regurgitate by citing sources, acknowledging gaps, and being honest about what the context does and doesn't cover. It also passes

the confidence score through to the final state so the API response can communicate certainty to the frontend.

The *direct answer node* (`direct_answer_node`) handles queries where `needs_retrieval` is false (that is, we don't need retrieval). It's simple by design: call the LLM with the question, then return the response. We assign a high confidence of 0.9 to these answers because we're trusting the model's general knowledge rather than potentially incomplete retrieval results.

Edge Definitions and Routing

Two routing functions connect the nodes. The `route_after_analysis` function fires after the analysis node and simply checks the `needs_retrieval` flag: if that flag is `true`, head to retrieval; if `false`, skip straight to the direct answer. The `route_after_evaluation` function fires after the evaluation node and implements the retry loop: if retrieval was sufficient, go ahead and generate; if not, go back and try again. The `iteration` counter in our state prevents infinite loops.

Assembling the Graph

With all nodes and routing functions defined, assembling the graph is straightforward. We add the nodes, set the entry point, and wire up the edges:

```
from langgraph.graph import StateGraph, END

# Create the graph
graph = StateGraph(AgenticRAGState)

# Add all nodes
graph.add_node("analyze", analyze_query_node)
graph.add_node("retrieve", retrieve_node)
graph.add_node("evaluate", evaluate_node)
graph.add_node("generate", generate_node)
graph.add_node("direct", direct_answer_node)

# Set entry point
graph.set_entry_point("analyze")

# Add edges
graph.add_conditional_edges("analyze", route_after_analysis, {
    "retrieve": "retrieve",
    "direct": "direct"
})

graph.add_edge("retrieve", "evaluate")

graph.add_conditional_edges("evaluate", route_after_evaluation, {
    "generate": "generate",
```

```
    "retrieve": "retrieve"  # Retry retrieval
})

graph.add_edge("generate", END)
graph.add_edge("direct", END)

# Compile
agentic_rag = graph.compile()
```

The compiled graph is ready to use. Call `agentic_rag.invoke` with an initial state—just the `query` and an empty `messages` list—and watch it flow through the nodes. The final state contains everything: query type, complexity, confidence score, and the response itself.

Query Planning and Analysis

Query analysis is the brain of agentic RAG. Good analysis leads to good retrieval, which leads to good answers. Let's explore how to make this node smarter.

Understanding User Intent

Users don't always phrase questions optimally for retrieval. Someone might ask, "How do I make my chatbot remember stuff?" when the relevant documents talk about "memory systems" and "conversation persistence." The analysis node needs to understand intent and translate it into effective search terms, thus bridging the gap between how users ask and how documents are written.

The way you bridge that gap is with an intent extraction prompt that asks the LLM to identify the core topic using technical terms, the specific aspect the user actually wants to know, related concepts that might be relevant, and a list of search terms likely to appear in relevant documents. For the chatbot memory example, you'd expect to get back something like "memory systems in LLM applications" as the core topic, "implementing conversation persistence" as the specific aspect, and terms like "context window," "vector stores," and "session state" as related concepts. Those terms will hit your vector store far more reliably than the raw user query.

Query Decomposition

Complex questions often contain multiple sub-questions packed into one. "Compare RAG and fine-tuning for customizing LLM behavior, considering cost, complexity, and flexibility" is really three separate retrieval problems. Breaking it down enables you to retrieve focused content for each aspect, rather than hoping that a single search catches everything.

The decomposition prompt instructs the LLM to break the query into a maximum of four independent sub-questions, each answerable with a focused search and simple enough to match specific document chunks. For simple queries, the prompt just returns the original unchanged. You don't pay the decomposition cost when you don't need it.

Complexity Assessment

Knowing query complexity helps allocate resources appropriately. Simple questions don't need extensive retrieval, while complex questions need comprehensive coverage.

We assess complexity based on several factors: the number of distinct concepts in the query, whether comparison or evaluation is required, whether the answer requires synthesizing multiple sources, and how domain-specific the topic is. The question "What is an embedding?" is simpler than "How do different embedding models compare for multilingual applications?" and our retrieval strategy should reflect that difference.

In practice, complexity assessment feeds directly into *k* selection in the retrieval node. Simple queries get fewer documents; complex queries get more. It also controls how many search queries the system generates: simple questions need one focused search; complex ones benefit from multiple angles on the same topic.

Dynamic Retrieval Strategies

Once we understand the query, we can tailor the retrieval strategy. Not every query requires the same approach, and a fixed pipeline can't adapt. Dynamic retrieval adapts to each query rather than using a one-size-fits-all approach. We make three key decisions in the analysis node: *Should* we retrieve at all? *What* should we search for? And *how much* should we retrieve?

Deciding When to Retrieve

Some queries don't need retrieval at all. General knowledge questions, simple calculations, and conversational messages can be handled directly by the LLM. Retrieving on these types of queries wastes time and might even hurt quality by injecting irrelevant context.

For the first decision (should we retrieve?), the system uses a two-stage approach. In the first stage, fast pattern-matching catches the obvious non-retrieval cases—such as greetings, simple arithmetic, and conversational filler—without burning an LLM call. The second stage is for the ambiguous cases, where an LLM prompt makes the judgment call, with clear instructions about what the knowledge base contains and what "confidently answerable from general

knowledge" means. Most of the time, the heuristics handle the easy cases and the LLM handles the edge cases.

Deciding What to Retrieve

The user's query isn't always the best search query. Query reformulation can dramatically improve retrieval quality. So, our second decision is: What exactly should we search for?

The system generates one, two, or three search queries depending on complexity. Simple questions get one focused query; complex ones get three queries attacking the topic from different angles. The number scales automatically based on the complexity assessment from the previous step.

The key principle here is to use technical vocabulary that would appear in the source documents, not the colloquial phrasing the user might use. A user asking, "How do I make search faster?" should trigger searches for "approximate nearest neighbor search" and "HNSW indexing," not a literal search for the user's words.

Deciding How Much to Retrieve

The number of documents to retrieve should match query complexity. Simple queries need focus because too many documents adds noise and dilutes the relevant signal. Complex queries need coverage because too few documents misses important aspects of a multifaceted answer. So, our third decision is: How many documents do we need?

Both complexity and query type influence the count. Comparison questions need more sources to represent multiple perspectives; factual questions usually have one definitive source. The retrieval node combines these two dimensions into a single k value, then clamps the result between 1 and 10 to prevent pathological cases. A simple factual question might end up with `k=2`; a complex comparison query might reach `k=9`.

Multi-Source Retrieval

For complex queries, the system runs multiple search queries and combines the results intelligently. Each search uses one of the generated query variants, and results are deduplicated by hashing a prefix of the document content; the first 500 characters are usually enough to identify duplicate chunks without hashing the full text. This multi-query approach improves coverage at the cost of more computation, which is exactly the trade-off you want for complex questions that justify the extra work.

Result Synthesis and Response Generation

Retrieval is only half the battle. The generation step needs to synthesize retrieved information into a coherent, accurate response, especially when multiple documents cover different aspects of the answer.

Combining Multiple Sources

When we retrieve from multiple queries, we might get documents that cover different aspects of the answer. The generation prompt needs to guide the LLM to synthesize rather than just summarize by weaving information together, citing sources naturally, and presenting a unified answer rather than a list of document summaries. Source attribution is embedded directly in the prompt by prepending each document with its source label, so the LLM can cite inline without guessing where information came from.

The confidence score also shapes the generation prompt. If confidence is below 0.7, the prompt adds an explicit instruction to acknowledge any gaps and avoid overstating certainty. This is how the system "knows what it doesn't know"—not by a separate step, but by conditioning generation on the evaluation score from earlier in the pipeline.

Handling Conflicting Information

Sometimes retrieved documents contain conflicting information. Maybe one discusses an older approach while another describes current best practices. The system can detect this with a dedicated conflict-checking prompt that asks the LLM to review the retrieved set and identify contradictions. When conflicts exist, the generation node acknowledges them explicitly rather than arbitrarily picking one version. "Sources disagree on this point: older documentation suggests X, while recent guides recommend Y" is a much more useful answer than silently choosing the wrong one.

Confidence Scoring

A crucial feature of agentic RAG is knowing what you don't know. The confidence score helps users understand how much to trust the answer, and it feeds back into system behavior: low confidence triggers retrieval retries and conditions the generation prompt to be more appropriately hedged.

Confidence is a weighted combination of three factors: retrieval relevance (how topically on-target the documents are), coverage (how completely they address the specific question), and consistency (whether multiple sources agree).

The weights reflect importance: relevance and coverage each contribute 40%, consistency, 20%. A single isolated source is inherently less reliable than two sources saying the same thing, and the scoring reflects that.

Handling Ambiguity and Uncertainty

Real users ask ambiguous questions. They use pronouns without antecedents; they assume context that doesn't exist; and they ask about things outside your knowledge base. A good agentic RAG system handles these cases gracefully rather than hallucinating confidently.

Detecting Ambiguous Queries

Some queries are clear: "What is cosine similarity?" Others are ambiguous: "How does it work?" The system detects ambiguity by checking for unclear pronouns without referents, missing context that makes the query interpretable in multiple ways, vague terms, and references to prior conversation that isn't available. It uses the recent conversation history (the last three turns) to resolve ambiguity wherever possible before flagging it.

The detection prompt returns structured output: whether the query is ambiguous, what kind of ambiguity it is, a clarifying question to ask if needed, and the system's best interpretation if we decide to proceed anyway. That last field is important because even when we detect ambiguity, we often have a reasonable guess at what the user means.

When to Ask vs. When to Assume

Asking too many clarifying questions is annoying. Assuming wrongly is worse. The balance depends on ambiguity severity and context. Pronoun ambiguity ("How does it handle errors?") and reference ambiguity ("Can you expand on that?") usually warrant a clarifying question because we can't proceed sensibly without knowing the referent. Vague queries ("Tell me more about this topic") can usually be handled with a best-guess interpretation, as long as the response acknowledges the assumption explicitly. The system notes the interpretation in its answer: "I'm interpreting this as a question about X—let me know if you meant something else."

Acknowledging Uncertainty

When retrieval fails or confidence is low, the system should communicate uncertainty clearly rather than making up answers. The uncertain response

prompt asks the LLM to share whatever relevant information it does have, acknowledge what it doesn't have, and suggest how the user might find better information—all without excessive apologizing. "Here's what I found, though the documents don't directly address your specific question. For more detail on X, you might want to check Y" is far more useful than either a confident wrong answer or a flat refusal.

Observability and Monitoring with LangSmith

You can't improve what you can't see. Agents make autonomous decisions, and understanding those decisions is essential for debugging, optimization, and building trust. LangSmith provides the observability layer that makes agents inspectable.

When a traditional program fails, you look at the stack trace. When an agent fails, you need to understand a sequence of decisions: What did the query analysis conclude? What did it retrieve? Why did it decide the retrieval was sufficient? What context did it use for generation? Without visibility into these steps, debugging is guesswork.

Observability also helps you improve the system over time. Which queries consistently get low confidence scores? Where does the routing logic make wrong decisions? Which types of questions lead to multiple retrieval iterations? These patterns are invisible without tracing.

Setting Up LangSmith Traces

LangSmith tracing requires minimal setup. You need an API key and a few environment variables:

```
import os

os.environ["LANGSMITH_API_KEY"] = "your-api-key"
os.environ["LANGSMITH_TRACING"] = "true"
os.environ["LANGSMITH_PROJECT"] = "studybuddy-v4"
```

Once these are set, LangChain and LangGraph automatically trace all LLM calls, tool executions, and graph traversals. Every operation gets logged to LangSmith where you can inspect it. You don't need to change any of your application code.

For custom functions that LangGraph doesn't automatically trace, such as utility helpers or preprocessing steps, add the `@traceable` decorator from `langsmith`. That's it. The decorator wraps the function and reports inputs, outputs, and latency to the same project trace.

Debugging and Analyzing Traces

LangSmith Studio v2 provides a web interface at `smith.langchain.com` where you can view traces. Each trace shows the complete execution flow of your agent, including every node visited, every LLM call made, every tool executed, inputs and outputs at each step, latency for each operation, and token usage.

The graph view is particularly useful for LangGraph applications. You can see which nodes were visited, the order of execution, and which conditional edges were taken. This makes it easy to understand why the agent made specific decisions.

LangSmith Studio v2 introduced time-travel debugging. You can step backward and forward through an agent's execution, examining the state at each point. If something went wrong at the generation step, you can go back to see what the retrieval step returned, what the evaluation step concluded, and where things went off track.

This is incredibly powerful for debugging complex agent behaviors. Instead of adding print statements and rerunning, you can explore the execution interactively.

Beyond debugging, traces enable systematic improvement. You can filter traces by confidence score to find queries where the system struggled, look at latency patterns to identify slow nodes, examine low-confidence responses to understand knowledge gaps, and compare traces before and after changes to verify improvements.

LangSmith also supports feedback collection. You can add scores to traces indicating whether the response was good or bad, then use this data to identify patterns in failures.

Using Open-Source Models

Here's something worth talking about: you don't have to use OpenAI. The examples in this book default to OpenAI's models because they're fast, reliable, and easy to get started with. But open-source models have gotten quite good, and they come with some notable advantages.

Open-source models are LLMs that you can download and run on your own hardware. Models like Llama, Mistral, and Qwen are freely available, fully documented, and, in many cases, competitive with proprietary models for specific tasks. When you use open-source models, you're generally not calling an API. Instead, you're running the model locally or on your own servers. This gives you considerably more control.

Why would you want to do this? A few reasons:

- **Cost:** API calls add up fast when you're building agents that make dozens of LLM calls per task. Running models locally means zero per-token costs after your initial hardware investment.

- **Privacy:** If you're processing sensitive data, you might not want it leaving your infrastructure. Open-source models keep everything in-house.
- **Customization:** You can fine-tune open-source models for your specific domain, something that's expensive or impossible with proprietary APIs.
- **Reliability:** You're not dependent on external API uptime, rate limits, or service changes.

The trade-offs? Open-source models generally lag behind frontier models in raw capability. GPT-5 and Claude Opus are still ahead on complex reasoning tasks. You also need to run these models somewhere. The good news is that smaller quantized models (like `llama3.2:3b`) run fine on modern laptops, especially Apple Silicon Macs. Larger models benefit from dedicated GPUs. And you're responsible for the infrastructure, which means more operational complexity.

But for many production use cases, open-source models are capable enough. A Llama 3.2 model running locally might be all you need for a tutoring agent, especially if you're fine-tuning it on educational content. The cost savings alone can justify the extra setup work.

Here's how to get started with open-source models using Ollama, which simplifies local model deployment. Ollama handles downloading models, managing them, and serving them through a local API that works seamlessly with LangChain.

First, head over to `https://ollama.com/` to install Ollama (if you haven't already done that). Then switch to a terminal to pull the models you want to use:

```
ollama pull llama3.2:3b
ollama pull nomic-embed-text
```

Verify Ollama is running:

```
ollama serve
```

Open your `pyproject.toml` and add the following to the dependencies array:

```
"langchain-ollama>=0.2.0",
```

Then install:

```
cd v4-agentic-rag    # Or wherever
uv add langchain-ollama
```

Here's some code you can run to verify Ollama is running and the models are available:

```
from langchain_ollama import ChatOllama, OllamaEmbeddings

# Test connection to Ollama
```

```
try:
    # Try a smaller model first (3B instead of 20B)
    test_llm = ChatOllama(model="llama3.2:3b", temperature=0)
    test_response = test_llm.invoke("Say 'Ollama is working!' in exactly
3 words.")
    print(f"Chat Model Test: {test_response.content}")

    test_embeddings = OllamaEmbeddings(model="nomic-embed-text")
    test_vector = test_embeddings.embed_query("test")
    print(f"Embedding Model Test: Vector dimension = {len(test_
vector)}")
    print("\nOllama is ready!")
except Exception as e:
    print(f"Error connecting to Ollama: {e}")
    print("\nMake sure:")
    print("1. Ollama is installed: https://ollama.com/")
    print("2. Ollama is running: 'ollama serve'")
    print("3. Models are pulled: 'ollama pull llama3.2:3b' and 'ollama
pull nomic-embed-text'")
```

This code tests both the chat model and embeddings. If Ollama is running and you've pulled the models, you'll see confirmation that everything works. If not, you'll get clear instructions on what to fix.

Once Ollama is working, swapping models in your code is trivial. Instead of `ChatOpenAI(model="gpt-4o-mini")`, you use `ChatOllama(model="llama3.2:3b")`. Same interface, different backend. That's the beauty of LangChain's component architecture.

The `llama3.2:3b` model is a good starting point because it's small enough to run on most modern hardware but capable enough for real tasks. As you scale up, you can try larger models like `llama3.2:20b`, `gpt-oss:20b`, or specialized models fine-tuned for specific domains. The Ollama library makes it easy to experiment.

Building StudyBuddy v4

Alright, let's put everything together. We're rebuilding StudyBuddy using LangGraph, implementing all the agentic RAG patterns we've discussed, and adding observability with LangSmith. This version is a significant step up in sophistication.

What We're Building

StudyBuddy v4 transforms from a simple agent into a sophisticated learning assistant with intelligent retrieval. The knowledge base changes too. Instead of Sherlock Holmes stories, we're using the AI Engineering reference guides.

This makes StudyBuddy a tool that helps you learn the very concepts you're studying in this book. Meta, right?

The new features include a LangGraph-based architecture with explicit state management; query analysis that determines complexity and retrieval strategy; dynamic retrieval that adapts based on query needs; evaluation and reflection that assess answer quality; confidence scoring so the system knows what it doesn't know; a dedicated flashcard endpoint for study mode (the foundation for the multi-agent system in Chapter 5); and LangSmith observability for debugging and improvement.

The Starting Point

We're building on the concepts from v3 but changing the architecture significantly. The vector store and embedding setup remain similar, but the agent logic is completely new and based on LangChain. Here's our foundation:

```
from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from pathlib import Path
import os
import json
import re
import random
from typing import TypedDict, Annotated, Literal, Optional
from dotenv import load_dotenv

# LangChain imports
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_qdrant import QdrantVectorStore
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import TextLoader

# LangGraph imports
from langgraph.graph import StateGraph, END, add_messages

# Qdrant imports
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams

# Load environment variables from parent directory
env_path = Path(__file__).parent.parent / ".env"
load_dotenv()

# LangSmith setup
os.environ["LANGSMITH_TRACING"] = "true"
os.environ["LANGSMITH_PROJECT"] = "studybuddy-v4"
```

The Knowledge Base

StudyBuddy v4's knowledge base consists of nearly three dozen reference guides, all markdown files covering RAG, agents, LangGraph, embeddings, tool use, memory systems, and more. These files should be placed in a `documents` folder:

```
documents/
├── ref-agentic-rag-pattern.md
├── ref-agents-and-agency.md
├── ref-chunking-strategies.md
├── ref-embeddings.md
├── ref-evaluation-metrics.md
├── ref-langgraph-quickref.md
├── ref-langsmith-quickref.md
├── ref-memory-systems.md
├── ref-rag-fundamentals.md
├── ref-reflection-pattern.md
├── ref-retrieval-patterns.md
├── ref-tool-use.md
└── ... (other ref-*.md files)
```

The indexing code loads and chunks these files:

```
# Initialize embeddings and vector store
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
qdrant_client = QdrantClient(":memory:")

COLLECTION_NAME = "ai_engineering_guides"

qdrant_client.create_collection(
    collection_name=COLLECTION_NAME,
    vectors_config=VectorParams(size=1536, distance=Distance.COSINE)
)

vector_store = QdrantVectorStore(
    client=qdrant_client,
    collection_name=COLLECTION_NAME,
    embedding=embeddings
)

indexing_status = {"done": False, "count": 0, "chunks": 0, "current_
file": ""}

def index_reference_guides():
    """Index all reference guide markdown files."""
    documents_dir = Path(__file__).parent.parent / "documents"
    if not documents_dir.exists():
        print(f"Documents directory not found: {documents_dir}")
        indexing_status["done"] = True
        return 0
```

```
    guide_files = sorted(documents_dir.glob("ref-*.md"))
    if not guide_files:
        print("No ref-*.md files found in documents directory")
        indexing_status["done"] = True
        return 0

    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=1000,
        chunk_overlap=100,
        length_function=len
    )

    print("Indexing reference guides...")
    total_chunks = 0
    for filepath in guide_files:
        doc_name = filepath.stem.replace("ref-", "").replace("-", " ")
.title()
        indexing_status["current_file"] = doc_name

        loader = TextLoader(str(filepath))
        documents = loader.load()
        chunks = text_splitter.split_documents(documents)

        # Add source metadata
        for chunk in chunks:
            chunk.metadata["source"] = doc_name

        vector_store.add_documents(chunks)
        total_chunks += len(chunks)
        indexing_status["chunks"] = total_chunks
        print(f"Indexed {len(chunks)} chunks from {doc_name}")

    print(f"\nTotal: {total_chunks} chunks indexed from {len(guide_
files)} guides")

    indexing_status["done"] = True
    indexing_status["count"] = len(guide_files)
    indexing_status["current_file"] = ""

    return total_chunks
```

Node Implementations

Let's implement each node for StudyBuddy.

First, we define our state type and initialize the LLM:

```
class StudyBuddyState(TypedDict):
    # Conversation
    messages: Annotated[list, add_messages]
```

```
    # Query analysis
    query: str
    query_type: str
    complexity: str
    needs_retrieval: bool
    search_queries: list[str]

    # Retrieval
    retrieved_docs: list
    retrieval_sufficient: bool
    iteration: int

    # Response
    response: str
    confidence: float

llm = ChatOpenAI(model="gpt-4o-mini")
```

Our state is divided into three sections: conversation history, query analysis results, and response generation fields. This mirrors the flow of information through our pipeline.

Our query analysis node is StudyBuddy's first step:

```
def analyze_query_node(state: StudyBuddyState) -> dict:
    """Analyze the incoming query to determine handling strategy."""

    query = state["query"]

    analysis_prompt = f"""You are StudyBuddy, an AI engineering tutor.
Analyze this student query:

Query: "{query}"

Your knowledge base covers: RAG, embeddings, vector databases, agents,
LangChain,
LangGraph, tool use, memory systems, evaluation, and related AI
engineering topics.

Determine:
1. query_type: "factual" | "conceptual" | "procedural" | "comparison"
2. complexity: "simple" | "moderate" | "complex"
3. needs_retrieval: true if this needs info from the knowledge base,
false if
   it's general knowledge or conversational
4. search_queries: if retrieval needed, list 1-3 effective search
queries
```

```
Respond with JSON only."""

    response = llm.invoke(analysis_prompt)

    try:
        # Handle potential markdown code blocks in response
        content = response.content.strip()
        if content.startswith("```"):
            content = content.split("```")[1]
            if content.startswith("json"):
                content = content[4:]
        analysis = json.loads(response.content)
    except json.JSONDecodeError:
        # Fallback for parsing errors
        analysis = {
            "query_type": "conceptual",
            "complexity": "moderate",
            "needs_retrieval": True,
            "search_queries": [query]
        }

    return {
        "query_type": analysis.get("query_type", "conceptual"),
        "complexity": analysis.get("complexity", "moderate"),
        "needs_retrieval": analysis.get("needs_retrieval", True),
        "search_queries": analysis.get("search_queries", [query]),
        "iteration": 0
    }
```

Notice that we handle JSON parsing robustly by stripping markdown code blocks and falling back gracefully if the LLM doesn't return valid JSON. This prevents a malformed response from crashing our agent.

The retrieval node adapts to complexity:

```
def retrieve_node(state: StudyBuddyState) -> dict:
    """Retrieve relevant documents with adaptive k based on
complexity."""

    complexity = state.get("complexity", "moderate")
    search_queries = state.get("search_queries", [state["query"]])

    # Adaptive k based on complexity
    k_map = {"simple": 2, "moderate": 4, "complex": 6}
    k = k_map.get(complexity, 3)

    # Multi-query retrieval with deduplication
    all_docs = []
    seen = set()
```

```
    for query in search_queries:
        docs = vector_store.similarity_search(query, k=k)
        for doc in docs:
            content_hash = hash(doc.page_content[:200])
            if content_hash not in seen:
                seen.add(content_hash)
                all_docs.append(doc)

    return {"retrieved_docs": all_docs}
```

We use a 200-character prefix for deduplication. Less than our 500-character prefix earlier but still enough to identify duplicates without hashing entire documents. This keeps memory usage reasonable.

The evaluation node decides whether retrieval was successful:

```
def evaluate_node(state: StudyBuddyState) -> dict:
    """Evaluate if retrieved documents are sufficient to answer the
query."""

    query = state["query"]
    docs = state["retrieved_docs"]
    iteration = state.get("iteration", 0)

    if not docs:
        return {
            "retrieval_sufficient": False,
            "confidence": 0.0,
            "iteration": iteration + 1
        }

    context_preview = "\n".join([doc.page_content[:300] for doc in
docs[:5]])

    eval_prompt = f"""Evaluate if these documents can answer the query:

Query: {query}

Retrieved content preview:
{context_preview}

Rate confidence 0-1 that you can fully answer the query with this
context.
Consider:
- Is the topic covered?
- Is there enough detail?
- Are there gaps that would need to be filled?

Respond with JSON only, no markdown: {{"confidence": X}}"""
```

```
    response = llm.invoke(eval_prompt)

    try:
        content = response.content.strip()
        if content.startswith("```"):
            content = content.split("```")[1]
            if content.startswith("json"):
                content = content[4:]
        result = json.loads(response.content)
        confidence = result.get("confidence", 0.5)
    except:
        confidence = 0.5

    # Stop iterating after 2 attempts or if confidence is good
    sufficient = confidence >= 0.7 or iteration >= 2

    return {
        "retrieval_sufficient": sufficient,
        "confidence": confidence,
        "iteration": iteration + 1
    }
```

The early return for empty docs handles the edge case where nothing was retrieved. We also limit retries to two iterations (as checked by `` `sufficient = ... or iteration >= 2` ``).

The generation node creates our final response:

```
def generate_node(state: StudyBuddyState) -> dict:
    """Generate educational response with optional flashcard
suggestion."""

    query = state["query"]
    docs = state["retrieved_docs"]
    confidence = state.get("confidence", 0.5)

    # Format context with sources
    context_parts = []
    for doc in docs:
        source = doc.metadata.get("source", "Reference Guide")
        context_parts.append(f"[{source}]:\n{doc.page_content}")

    context = "\n\n---\n\n".join(context_parts)

    generate_prompt = f"""You are StudyBuddy, an AI engineering tutor.

Student question: {query}

Reference material:
{context}
```

```
Instructions:
- Explain clearly and educationally
- Cite sources when using specific information
- Use examples to illustrate concepts
- If the context doesn't fully cover the question, acknowledge gaps
- Match your explanation depth to the question complexity"""

    response = llm.invoke(generate_prompt)

    return {
        "response": response.content,
        "confidence": confidence
    }
```

We format each document with its source so the LLM can cite properly. The prompt emphasizes being educational and matching complexity to the query.

Finally, the direct answer node for queries that don't need retrieval:

```
def direct_answer_node(state: StudyBuddyState) -> StudyBuddyState:
    """Answer without retrieval for simple queries."""

    query = state["query"]

    response = llm.invoke(
        f"You are StudyBuddy, a friendly AI tutor. Answer this briefly:
{query}"
    )

    return {
        "response": response.content,
        "confidence": 0.9,
        "retrieved_docs": []
    }
```

We give high confidence again since we're using the LLM's trained knowledge directly on general queries that don't require retrieval.

Routing Functions

The routing logic determines flow through the graph:

```
def route_after_analysis(state: StudyBuddyState) -> Literal["retrieve",
"direct"]:
    """Route based on whether retrieval is needed."""
    if state.get("needs_retrieval", True):
        return "retrieve"
    return "direct"
```

```
def route_after_evaluation(state: StudyBuddyState) ->
Literal["generate", "retrieve"]:
    """Route based on retrieval quality."""
    if state.get("retrieval_sufficient", False):
        return "generate"
    return "retrieve"
```

Building the Graph

Now we assemble everything into the complete graph:

```
def create_studybuddy_graph():
    """Create and compile the StudyBuddy v4 agent graph."""

    graph = StateGraph(StudyBuddyState)

    # Add nodes
    graph.add_node("analyze", analyze_query_node)
    graph.add_node("retrieve", retrieve_node)
    graph.add_node("evaluate", evaluate_node)
    graph.add_node("generate", generate_node)
    graph.add_node("direct", direct_answer_node)

    # Set entry point
    graph.set_entry_point("analyze")

    # Add edges
    graph.add_conditional_edges(
        "analyze",
        route_after_analysis,
        {"retrieve": "retrieve", "direct": "direct"}
    )

    graph.add_edge("retrieve", "evaluate")

    graph.add_conditional_edges(
        "evaluate",
        route_after_evaluation,
        {"generate": "generate", "retrieve": "retrieve"}
    )

    graph.add_edge("generate", END)
    graph.add_edge("direct", END)

    return graph.compile()

# Create the agent (will be initialized after indexing)
studybuddy = None
```

```
_initialized = False


def ensure_initialized():
    """Ensure documents are indexed and agent is ready. Thread-safe."""
    global studybuddy, _initialized
    if _initialized:
        return

    print("Initializing StudyBuddy...")
    index_reference_guides()
    studybuddy = create_studybuddy_graph()
    _initialized = True
    print("Agent ready!")
```

The API Layer

The preceding code examples focus on the core agentic RAG architecture. The actual StudyBuddy v4 implementation extends this with a flashcard-first user interface, where students see flashcards as their primary study mode, with chat available as a secondary feature for deeper exploration.

This required a few additions to the API layer:

- An `/api/chapters` endpoint that parses a `topic-list.md` file for chapter/topic navigation
- An `/api/flashcard` endpoint that generates scoped flashcards using dedicated flashcard generation logic
- A `card_context` parameter in the chat endpoint so the tutor knows which flashcard you're studying

The chat feature focuses on tutoring—answering questions and explaining concepts—while flashcard generation is handled separately by the dedicated `/api/flashcard` endpoint. This separation keeps chat responses clean and focused. Refer to the StudyBuddy repo to examine the API code.

Running Locally

First, make sure you have all your dependencies in `pyproject.toml`:

```
[project]
name = "studybuddy"
version = "4.0.0"
description = "AI-powered study assistant - Agentic RAG with LangGraph"
requires-python = ">=3.12"
dependencies = [
    "fastapi>=0.115.0",
    "uvicorn[standard]>=0.32.0",
```

```
    "python-dotenv>=1.0.0",
    "langchain>=1.0.0",
    "langchain-core>=1.0.0",
    "langchain-openai>=1.0.0",
    "langchain-ollama>=1.0.0",
    "langchain-community>=0.4.0",
    "langchain-text-splitters>=0.3.0",
    "langchain-qdrant>=1.0.0",
    "langgraph>=1.0.0",
    "qdrant-client>=1.12.0",
]
```

With that done, from the project root directory (`v4-agentic-rag`), run:

```
uv sync
```

StudyBuddy v4 runs with two terminals.

- One terminal runs the backend (port 8000):

```
cd v4-agentic-rag
uv run uvicorn api.index:app --reload --port 8000
```

- The other terminal sets up the frontend:

```
cd v4-agentic-rag/frontend
npm install
```

Now, on the frontend terminal, start the dev server (port 3000):

```
npm run dev
```

Visit `http://localhost:3000` to use the app.

Testing the System

Let's test StudyBuddy v4 with different types of questions.

Simple factual query (should use retrieval, low complexity):

```
# "What is cosine similarity?"
#
# Expected behavior:
# - Query analysis: factual, simple, needs_retrieval=True
# - Retrieval: k=2, single search query
# - High confidence, quick response
```

Complex conceptual query (should use multi-query retrieval):

```
# "Compare different chunking strategies for RAG systems"
#
# Expected behavior:
# - Query analysis: comparison, complex, needs_retrieval=True
```

```
# - Retrieval: k=6, multiple search queries
# - Multiple iterations possible if first retrieval insufficient
# - Detailed response citing multiple sources
```

General knowledge query (should skip retrieval):

```
# "How do I write a for loop in Python?"
#
# Expected behavior:
# - Query analysis: procedural, simple, needs_retrieval=False
# - Routes directly to direct_answer_node
# - Quick response from general knowledge
```

Viewing Traces in LangSmith

With `LANGSMITH_TRACING` enabled, every request generates a trace. Go to `smith.langchain.com` and find your "studybuddy-v4" project. You'll see traces for each request showing the analyze node execution and its output, routing decisions (retrieve vs. direct), retrieval queries and results, evaluation scores, and generation inputs and outputs.

Click into any trace to see the full execution flow. Use the graph view to see which path through the state machine was taken. If something looks wrong, use time travel to step through and find where things went off track.

Deploying to Vercel

To deploy v4 to Vercel, your first need to update `api/requirements.txt`:

```
fastapi>=0.115.0
uvicorn[standard]>=0.32.0
python-dotenv>=1.0.0
langchain>=1.0.0
langchain-core>=1.0.0
langchain-openai>=1.0.0
langchain-community>=0.4.0
langchain-text-splitters>=0.3.0
langchain-qdrant>=1.0.0
langgraph>=1.0.0
qdrant-client>=1.12.0
```

Then you need to update two files in the repo root. First, update `vercel.json` to point to the `v4-agentic-rag` directory:

```
{
  "version": 4,
  "builds": [
```

```
    {
      "src": "v4-agentic-rag/api/index.py",
      "use": "@vercel/python"
    },
    {
      "src": "v4-agentic-rag/frontend/package.json",
      "use": "@vercel/next"
    }
  ],
  "routes": [
    { "src": "/api/(.*)", "dest": "/v4-agentic-rag/api/index.py" },
    { "src": "/(.*)", "dest": "/v4-agentic-rag/frontend/$1" }
  ],
  "git": {
    "deploymentEnabled": {
      "main": true
    }
  },
  "ignoreCommand": "bash scripts/should-deploy.sh"
}
```

Second, update `scripts/should-deploy.sh` to trigger on v4 changes:

```
#!/bin/bash
# Deploy if changes are in v4-agentic-rag or vercel config
# Exit 1 (true) = proceed with build, Exit 0 (false) = skip build
git diff HEAD^ HEAD --name-only | grep -qE "^v4-agentic-rag/|^vercel\
.json" && exit 1 || exit 0
```

If you haven't done so already, set your OpenAI API key environment variable in Vercel:

```
vercel env add OPENAI_API_KEY
```

Deploy:

```
vercel --prod
```

Or simply push to GitHub. Vercel will automatically deploy when changes are detected in the `v4-agentic-rag` directory.

What's Next

StudyBuddy v4 is a sophisticated agentic RAG system. It understands query complexity, adapts its retrieval strategy, evaluates its own results, and knows when it's uncertain. The dedicated flashcard generation endpoint lays the groundwork for the multi-agent system we'll build in Chapter 5.

In the next chapter, we'll expand StudyBuddy into a true multi-agent application. We'll have separate agents for tutoring, flashcard generation, quality checking, and scheduling, all coordinated by a supervisor. The single-agent pattern we built here will become one piece of a larger, more capable system.

Great work making it through this chapter. You now understand how to build agents with explicit state machine control, how to implement intelligent retrieval strategies, and how to observe and debug agent behavior. These skills will serve you well in everything that follows.

Let's keep building.

CHAPTER

5

Multi-Agent Applications

Okay, you've built agents. You've seen them retrieve, reason, and respond. You've watched them loop through problems and find solutions. But one agent can only do so much. When complexity goes up, when you're trying to handle multiple concerns at once, when you need specialization and coordination, that's when you turn to multi-agent systems. Each agent has its own role, its own expertise, its own context. They communicate, they delegate, they coordinate. The result? Applications that can handle much more complexity while keeping each piece manageable and focused.

But multi-agent systems aren't always the answer. Sometimes they're exactly what you need. Sometimes they're overkill that adds complexity without adding value. This chapter teaches you when to reach for multiple agents and when not to. And if using multiple agents is the way to go for your app, you'll learn how to architect them and how to build them well.

When to Use Multiple Agents

Before you start architecting elaborate agentic teams, ask yourself two important questions. First, the single-agent question: "Do I really need dynamic reasoning to solve the task more effectively than a rigid workflow could?" Second, the multi-agent question: "Do I really need several specialized dynamic reasoning

machines collaborating to solve the task more effectively than a rigid workflow could?"

You see, the reality is that multi-agent systems add complexity. Real complexity. More components to build, more failure modes to handle, more interactions to debug. If a single agent can do the job, remember that the simplest solution is often the best. When does it make sense to go multi-agent? Let's take a look.

The Case for Multi-Agent Systems

Multi-agent architectures work well in several scenarios:

- **Task decomposition:** When your workflow has clear phases that benefit from different prompts, tools, and models. A research phase might need web search tools and a model good at information synthesis. A writing phase might need no tools at all, just a model with strong language generation. Different agents can use different models. Maybe a smaller, faster model for quick decisions and a larger, more capable model for complex synthesis.
- **Parallel execution:** Some tasks don't need to happen sequentially. If you need to research three different topics, three researcher agents can work simultaneously rather than having one agent search three times in sequence. This can dramatically reduce latency for complex workflows.
- **Maintainability:** When each agent has a focused role with a clear system prompt, it's easier to iterate and improve individual components. Want to make your editor more aggressive about cutting fluff? Update one prompt. Want your researcher to prioritize primary sources? Change one agent. You're not wading through a monolithic prompt trying to balance competing instructions.

When *Not* to Build Multi-Agent Systems

Here's the flip side, and this is important. Don't build multi-agent systems just because you can. If your task is straightforward enough that one agent with the right tools handles it well, adding more agents adds latency, cost, and failure points without adding value.

Customer support chatbots are a great example of where people overengineer. Yes, you could have a routing agent, a product specialist agent, a billing specialist agent, and a coordination agent. Or you could have one well-prompted agent with access to the right knowledge base and tools. For most support interactions, the single agent wins on latency, cost, and reliability.

Also avoid multi-agent systems when latency is critical. Every handoff between agents adds round-trip time. If you're building something that needs to feel snappy and responsive—such as an interactive coding assistant or a real-time

conversation partner—the overhead of supervisor coordination might hurt the user experience.

A Decision Framework

When you're staring at a new project trying to decide on an architecture, work through these questions:

- Can a single well-prompted agent with the right tools handle this? If yes, start there.
- Does the task decompose into phases requiring genuinely different expertise? If yes, a multi-agent approach might help.
- Is context accumulation becoming a problem? If your single agent is degrading over long interactions, agent specialization can help.
- Are there opportunities for parallel execution? Multiple agents can work simultaneously.
- Is latency critical? If yes, be very careful about coordination overhead.

Start simple. Add complexity only when you hit real limitations. This is how you build systems that actually work.

Context Optimization

Let's dig deeper into one of the strongest arguments for multi-agent systems: context management. This is where the rubber meets the road for complex applications.

The Context Rot Problem

Imagine a single agent handling an extensive research and writing task. The user asks for a comprehensive report on renewable energy trends. The agent searches for solar energy statistics, and the search results are added to context. Then wind energy data: more context. Battery storage developments: more context. Government policy summaries: even more context. By the time the agent starts writing, its context window is stuffed with raw search results, many of which are only tangentially relevant to the actual report sections it needs to write.

This is *context rot* in action: the model's attention gets diluted across too much information. Important details get buried. The agent might start hallucinating connections between sources because it's pattern-matching across noise. Or it might simply forget earlier instructions about formatting and tone because those instructions are now buried under megabytes of search results.

Even with today's large context windows—128K tokens, 200K tokens—this problem doesn't disappear. Larger context windows mean you can fit more information, but they don't mean the model attends equally well to all of it. Research consistently shows that information in the middle of long contexts gets less attention than information at the beginning or end. More context isn't always better context.

How Multiple Agents Manage Context

Multi-agent architectures address context rot through separation. Each agent maintains its own focused context, optimized for its specific role. The researcher's context contains search queries and results. The writer's context contains research summaries and drafting instructions. The editor's context contains the draft and style guidelines.

Instead of everything piling into one context window, information flows between agents in distilled form. The researcher doesn't dump raw search results to the writer; it produces a synthesis: key findings, relevant quotes, source citations. The writer doesn't send its entire drafting history to the editor; it sends the current draft. Each handoff filters and compresses information.

This mirrors how human teams actually work. The researcher sends a briefing document, not their entire browser history. The writer sends a draft, not their stream-of-consciousness notes. Agents, like humans, benefit from receiving preprocessed, relevant information rather than raw data dumps.

Context Isolation Benefits

Beyond avoiding context rot, isolation provides other benefits. Agents can be optimized independently. Your researcher might benefit from a long context window to compare multiple sources simultaneously. Your editor might work better with a short, focused context containing just the draft and style guide. Different agents can use different context lengths.

Isolation also improves security and privacy. In enterprise applications, you might have some agents that can access sensitive data and others that cannot. A research agent might query internal databases, then pass only sanitized summaries to agents that produce user-facing outputs. Context isolation makes this separation natural to implement.

Finally, isolation helps with debugging. When something goes wrong in a single-agent system, you're digging through one enormous context trying to figure out where things went sideways. With multi-agent systems, you can examine each agent's context independently. Did the researcher find bad sources? Did the writer misinterpret the research? Did the editor introduce errors? The boundaries make diagnosis tractable.

Token Economics

Let's talk costs. You might think multi-agent systems are more expensive because you're making more API calls. Sometimes that's true. But context management affects costs too.

With large context windows, you pay for every token in that context on every generation. A single agent with 50K tokens of accumulated context pays for those 50K tokens every time it generates a response. If it generates ten responses during a workflow, that's 500K input tokens just from context.

Multi-agent systems with smaller, focused contexts might make more calls, but each call has lower input token costs. If your researcher works with 10K tokens, your writer works with 8K tokens, and your editor works with 5K tokens, the total per-step costs are lower even though you're making more steps.

Anthropic studied the tokenomics behind their multi-agent research system and found that while agents use about 4 times more tokens than simple chats, multi-agent systems use 15 times more. The math depends on your specific workflow, but don't assume a single-agent system is always cheaper. Model the actual token flows for your use case.

Multi-Agent Architectures

Now that you understand why multi-agent systems can be valuable, let's look at how to structure them. There are several well-established architectural patterns, each with different trade-offs.

The Supervisor Pattern

The supervisor pattern shown in Figure 5.1 is the most common multi-agent architecture, and it's the one you'll implement later in this chapter. A central supervisor agent receives user requests, analyzes what needs to be done, delegates tasks to specialized worker agents, collects their results, and synthesizes a final response.

The supervisor pattern mirrors how human teams often work. There's a manager who understands the big picture and coordinates specialists. The manager doesn't do the detailed work; it orchestrates. Workers focus on their specialties without worrying about the broader workflow.

The architecture looks like a hub and spokes: the supervisor at the center, workers around the edges, all communication flowing through the hub. The supervisor decides which workers to engage, what tasks to give them, and how to combine their outputs.

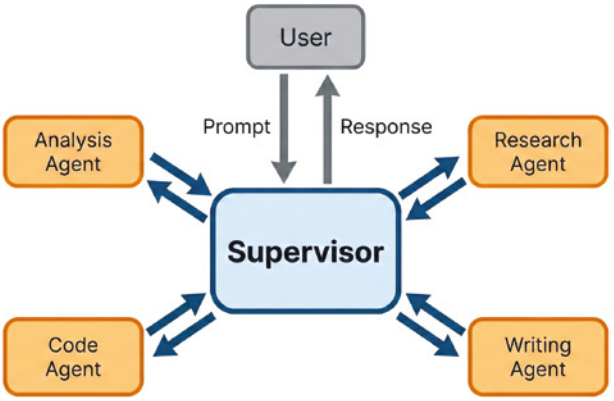

Figure 5.1: A multi-agent architecture using the supervisor pattern.

This pattern works well when tasks decompose into distinct subtasks, different subtasks require different expertise, you need coordinated multi-step workflows, and there's clear delegation logic the supervisor can follow.

The downsides are latency (every task goes through the supervisor) and a single point of failure (if the supervisor makes bad decisions, everything downstream suffers). For simple requests that don't need coordination, the supervisor adds overhead without value.

The Swarm Pattern

In contrast to centralized supervision, the swarm pattern shown in Figure 5.2 uses decentralized, peer-to-peer coordination. Each agent can hand off to other agents based on the conversation context. There's no central coordinator making decisions; the agents self-organize.

Imagine a software incident response system built with swarm coordination. A triage agent receives the initial alert, which might be a spike in error rates on a payment service. If it detects symptoms of a database bottleneck, it hands off to the database specialist. If the database specialist rules out slow queries and spots CPU saturation instead, it hands off to the infrastructure specialist. If that agent notices the pattern resembles a traffic anomaly rather than a hardware issue, it brings in the security specialist to check for an attack. Each agent knows what it's good at and what should go to someone else.

Swarm patterns work well for routing, triage, and progressive-disclosure scenarios where the true nature of the problem only becomes clear as you dig in. They're useful when the path through the system depends heavily on the conversation content, and that content reveals itself progressively. Incident response, fraud investigation, and complex technical troubleshooting are natural fits.

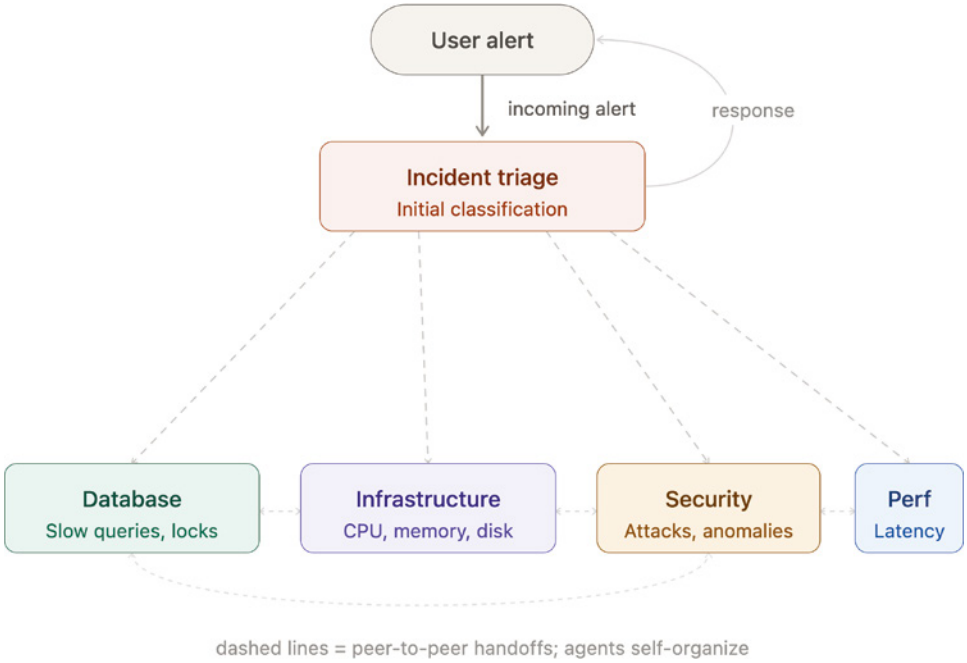

Figure 5.2: A swarm pattern uses decentralized, peer-to-peer coordination to implement the multi-agent system.

The challenges are unpredictability and the potential for infinite loops. Without a central coordinator, it's harder to guarantee termination or track the overall state of a complex task. You need careful handoff logic to prevent agents from ping-ponging requests back and forth forever.

Hierarchical Structures

When tasks get really complex, you might need supervisors managing other supervisors. A top-level coordinator delegates to mid-level managers, who coordinate their own teams of specialists. Figure 5.3 shows an example.

Consider an ambitious content production system. The top supervisor receives a request for a comprehensive industry report. It delegates the research section to a research manager, the analysis section to an analysis manager, and the writing to a content manager. Each manager coordinates their own team: the research manager works with web searchers, academic paper fetchers, and interview synthesizers. The content manager works with writers, editors, and fact-checkers.

Hierarchical patterns handle complexity by decomposing it recursively. But they also add significant coordination overhead and make debugging much harder. Use them only when the task genuinely demands them.

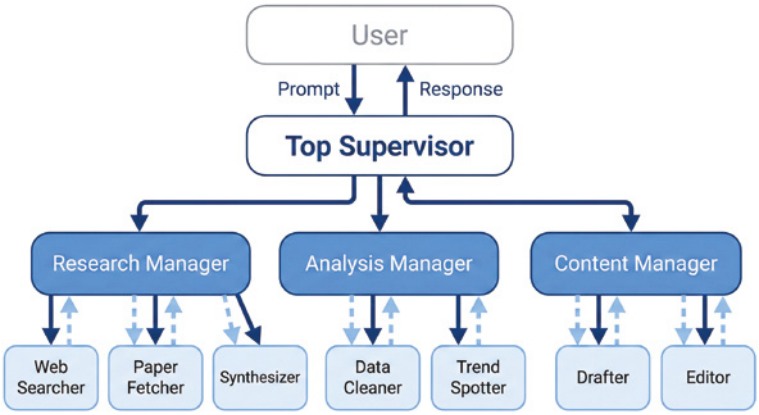

Figure 5.3: For complex tasks, a supervisor agent can coordinate a team of specialist agents.

The Pipeline Pattern

Sometimes you don't need much dynamic coordination. If your workflow is generally understood and consistent—always research, then write, then edit—a simple agent pipeline works fine, as shown in Figure 5.4. Each agent within the pipeline processes the output of the previous step in a predetermined sequence.

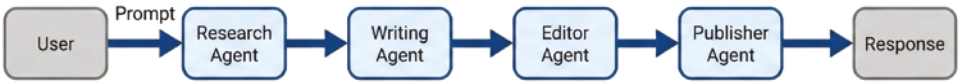

Figure 5.4: The pipeline pattern runs agents in a predefined sequence.

Pipelines are the simplest multi-agent architecture. No complex routing logic, no coordination decisions, just a chain of transformations and simple decisions along the way. They're fast, predictable, and easy to debug. If your workflow doesn't need much dynamic behavior, prefer a pipeline over more complex architectures.

We might just as well call this pattern an agentic workflow. These are the workhorses of industry and production environments and will continue to be primary drivers of return on agentic AI investment. Again, we see the principle that the simplest solution is the best: we optimize agents at production scale by removing agency that is not required.

It's worth recalling here the distinction we made between the LangChain `create_agent` construct and the choice to use LangGraph. We might take a graph-based approach to "put our agent on rails" or to "give it more of a predetermined workflow" that is more consistent for production. This is the agentic workflow in action. This is the pipeline pattern. This is the unsexy agentic AI that connects one workflow in your organization to another. This is how AI integrates into the way your company does work.

The limitation, of course, is inflexibility. A pipeline can't skip steps that aren't needed or go back to redo earlier work if later stages reveal problems. For truly dynamic (agentic) flows, you need something with more coordination capability. The dynamism in the pipeline pattern is fundamentally constrained to choosing one of a limited number of paths and taking action based on that choice.

Agent Roles and Specialization

Multi-agent systems succeed or fail based on how well you define agent roles. Fuzzy boundaries lead to confusion, overlap, and wasted effort. Clear specialization leads to efficiency and quality.

Manager Agents

Manager agents (supervisors, coordinators) have a fundamentally different job than workers. They don't do the actual work; they orchestrate it. Their system prompts focus on analysis, planning, and decision-making rather than execution.

A good supervisor prompt clearly enumerates the available workers and their capabilities, describes when to use each worker, provides decision-making frameworks or heuristics, and specifies what the supervisor should produce (delegation decisions, not final outputs).

Here's an example of effective supervisor prompting:

```
SUPERVISOR_PROMPT = """You coordinate a team of specialists.

Your team:
- RESEARCHER: Use for finding information, verifying facts, gathering
data
- WRITER: Use for creating content, drafting documents, structuring
information
- EDITOR: Use for reviewing content, fixing errors, improving quality

For each request:
1. Analyze what the user needs
2. Decide which specialist(s) to engage and in what order
3. Provide clear task instructions to each specialist
4. Review results and decide if more work is needed
5. Synthesize the final response

Guidelines:
- Simple requests may only need one specialist
- Complex requests often need research before writing
- Always have the editor review important content
- You may engage the same specialist multiple times if needed"""
```

Notice how the prompt defines clear criteria for choosing workers. The supervisor isn't guessing because it has explicit guidance about which situations call for which specialists.

Note: You can find all of this chapter's non-StudyBuddy code in the Jupyter notebook `chapter05.ipynb`, available on this book's GitHub repo.

Worker Agents

Worker agents are specialists. They're really good at one thing and shouldn't try to do everything. Their system prompts are focused and specific, emphasizing their particular skill and the quality standards they should maintain.

A researcher agent might have a prompt emphasizing source evaluation, fact-checking, and synthesis. A writer agent might focus on clarity, structure, and tone. An editor might emphasize grammar, consistency, and improvement suggestions.

The key is non-overlapping responsibilities. If your researcher is also trying to write polished prose and your writer is also trying to edit, you've blurred the boundaries. Each agent should have a clear lane.

Designing Agent Personalities

Beyond functional specialization, you can give agents distinct personalities that affect how they approach their work. A cautious researcher might flag uncertainty more aggressively. A creative writer might take more risks with language. A strict editor might be more aggressive about cutting.

These personalities aren't just flavor because they have real effects on outputs. A team with a bold researcher and a conservative editor produces different content than a team with a conservative researcher and a permissive editor. Think about what balance serves your use case.

Some teams benefit from tension. Having an optimistic agent propose ideas and a skeptical agent critique them can produce more balanced outputs than having everyone agree all the time. Adversarial collaboration is a valid design pattern.

Avoiding Role Confusion

The most common failure mode in multi-agent systems is role confusion. Agents step on each other's toes, duplicate work, or leave gaps because responsibilities weren't clear.

To avoid this, make responsibilities explicit in prompts, test agents individually before combining them, log which agent handles which tasks during debugging, and watch for patterns where multiple agents produce similar outputs.

If you find yourself writing increasingly elaborate prompts to prevent overlap, that's a sign your role definitions need work. Go back to basics and clarify which agent owns what.

Coordination and Communication

Here's the operational side of multi-agent systems: how agents actually talk to each other, pass work around, and handle things when something goes wrong. This is the agent coordination we saw in the swarm pattern, now in action.

Message Passing and Handoffs

The simplest coordination mechanism is explicit tool-based handoffs. The supervisor has tools for delegating to each worker, and calling those tools invokes the appropriate agent with the specified task.

```
@tool
def delegate_to_researcher(task: str) -> str:
    """
    Delegate a research task to the research specialist.
    Use when information gathering or fact verification is needed.

    Args:
        task: Description of what to research
    """
    result = researcher.invoke({"input": task})
    return result["output"]

@tool
def delegate_to_writer(task: str, context: str) -> str:
    """
    Delegate a writing task to the content writer.
    Use when content needs to be created from research.

    Args:
        task: Description of what to write
        context: Research and background information
    """
    result = writer.invoke({"input": task, "context": context})
    return result["output"]
```

The supervisor calls these tools just like any other tools. The tool implementation handles actually invoking the target agent and returning its result. This keeps the coordination logic clean and observable.

For more structured handoffs, you can use Pydantic models to make the inter-agent contract explicit by locking down exactly which fields flow in each direction and what types they carry:

```
from pydantic import BaseModel

class ResearchHandoff(BaseModel):
    task: str
    sources_to_check: list[str] = []
    deadline_priority: str = "normal"

class ResearchResult(BaseModel):
    findings: str
    sources: list[str]
    confidence: str
    gaps: list[str] = []
```

Structured handoffs make the interface between agents explicit and catch malformed communications early.

Shared State with LangGraph

LangGraph provides a natural way to manage shared state across agents. Instead of passing everything through tool calls, agents read from and write to a common state object. This is particularly powerful for complex workflows where multiple pieces of information need to be available to multiple agents.

```
from typing import TypedDict
from langgraph.graph import StateGraph, END

class TeamState(TypedDict):
    """Shared state for multi-agent coordination."""
    messages: Annotated[list, add_messages]   # Conversation history
    user_request: str                         # Original user request
    research_findings: str                    # Output from researcher
    draft_content: str                        # Output from writer
    edited_content: str                       # Output from editor
    next_action: str                          # Supervisor's routing
decision
    final_response: str                       # Final output to user
```

Agent Node Functions

Each agent is implemented as a node function that reads from state, performs its task, and returns state updates:

```
def supervisor_node(state: TeamState) -> dict:
    """Supervisor analyzes state and decides the next action."""
```

```
    # Format the prompt with current state
    prompt = SUPERVISOR_PROMPT.format(
        user_request=state.get("user_request", ""),
        research_findings=state.get("research_findings", "None yet"),
        draft_content=state.get("draft_content", "None yet"),
        edited_content=state.get("edited_content", "None yet")
    )

    response = llm.invoke([SystemMessage(content=prompt)])

    # Parse the JSON response
    default = {"next_action": "researcher", "reasoning": "Starting with
research"}
    result = parse_json_response(response.content, default)

    next_action = result.get("next_action", "researcher")
    reasoning = result.get("reasoning", "")

    print(f"[Supervisor] Decision: {next_action}")
    print(f"[Supervisor] Reasoning: {reasoning}\n")

    return {"next_action": next_action}
```

The supervisor formats a prompt with the current state, asks the LLM to decide which agent should act next (researcher, writer, editor, or complete), and returns that decision. You'll see this pattern where each node reads from state, does its specialized work, and writes results back.

The researcher node takes a user request and produces synthesized findings by searching for relevant information:

```
def researcher_node(state: TeamState) -> dict:
    """Researcher gathers information using web search."""

    user_request = state.get("user_request", "")

    print("[Researcher] Starting research...")

    # Generate search queries based on the request
    query_prompt = f"""Based on this content request, generate 2-3
search queries to gather relevant information.

Request: {user_request}
Respond with JSON: {{"queries": ["query1", "query2"]}}"""

    query_response = llm.invoke([SystemMessage(content=query_prompt)])
    queries_result = parse_json_response(query_response.content,
{"queries": [user_request]})
    queries = queries_result.get("queries", [user_request])
```

```
    # Perform searches
    all_results = []
    for query in queries[:3]:  # Limit to 3 queries
        print(f"[Researcher] Searching: {query}")
        results = web_search(query)
        all_results.append(f"Search: {query}\nResults: {results}\n")

    combined_results = "\n".join(all_results)

    # Synthesize findings
    synthesis_prompt = RESEARCHER_PROMPT.format(user_request=user_
request)
    synthesis_prompt += f"\n\nSearch Results:\n{combined_results}\n\
nSynthesize these findings into a comprehensive summary."

    synthesis = llm.invoke([SystemMessage(content=synthesis_prompt)])

    print("[Researcher] Research complete!\n")

    return {"research_findings": synthesis.content}
```

Notice the two-step process: first generate search queries, then actually search and synthesize. This mirrors how you'd approach research manually, by figuring out what to look for before looking.

The writer node transforms research findings into a draft:

```
def writer_node(state: TeamState) -> dict:
    """Writer creates content based on research findings."""

    print("[Writer] Creating draft...")

    prompt = WRITER_PROMPT.format(
        user_request=state.get("user_request", ""),
        research_findings=state.get("research_findings", "No research
available")
    )

    response = llm.invoke([SystemMessage(content=prompt)])

    print("[Writer] Draft complete!\n")

    return {"draft_content": response.content}
```

This is pretty straightforward: it reads the user request and research, then generates a draft. Simple nodes are good nodes.

The editor node takes a draft and improves quality:

```
def editor_node(state: TeamState) -> dict:
    """Editor reviews and polishes the content."""
```

```
    print("[Editor] Editing content...")

    prompt = EDITOR_PROMPT.format(
        user_request=state.get("user_request", ""),
        draft_content=state.get("draft_content", "No draft available")
    )

    response = llm.invoke([SystemMessage(content=prompt)])

    print("[Editor] Editing complete!\n")

    return {"edited_content": response.content}
```

Same structure as the writer: specialized prompt, returns updated state. The final node just packages everything for the user:

```
def complete_node(state: TeamState) -> dict:
    """Finalize the response."""

    print("[Complete] Finalizing response...")

    final_content = state.get("edited_content", state.get("draft_
content", ""))

    return {"final_response": final_content}
print("Agent node functions defined!")
```

Because each node only returns the fields it touches, the nodes are independent. For example, the researcher doesn't need to know the editor exists, and the editor doesn't care how the research was gathered.

Routing Logic

The router function reads the `next_action` field that the supervisor wrote to state and returns it as a string, and LangGraph uses that string as an edge label to dispatch execution to the correct node:

```
def supervisor_router(state: TeamState) -> Literal["researcher",
"writer", "editor", "complete"]:
    """Route to the appropriate agent based on supervisor's decision."""
    next_action = state.get("next_action", "researcher")

    if next_action in ["researcher", "writer", "editor", "complete"]:
        return next_action

    # Default to researcher if unknown action
    return "researcher"
```

Building and Running the StateGraph

Now we assemble all the pieces into a LangGraph workflow. The process follows three phases: define nodes, connect them with edges, then compile.

```
# Build the multi-agent graph
graph = StateGraph(TeamState)
# Add all agent nodes
graph.add_node("supervisor", supervisor_node)
graph.add_node("researcher", researcher_node)
graph.add_node("writer", writer_node)
graph.add_node("editor", editor_node)
graph.add_node("complete", complete_node)
```

Each `add_node()` call registers a function as a step in your workflow. The name string is how you'll reference it when connecting edges.

Next, we'll set where the graph starts, then add conditional routing from the supervisor:

```
# Set the entry point
graph.set_entry_point("supervisor")
# Add conditional edges from supervisor to workers
graph.add_conditional_edges(
    "supervisor",
    supervisor_router,
    {
        "researcher": "researcher",
        "writer": "writer",
        "editor": "editor",
        "complete": "complete"
    }
)
```

`add_conditional_edges()` creates branching paths. The router function you defined earlier determines which branch executes based on the current state.

Next, we'll connect worker nodes back to the supervisor. Once the workers complete their tasks, they return control to the supervisor for the next decision:

```
# Workers return to supervisor for next decision
graph.add_edge("researcher", "supervisor")
graph.add_edge("writer", "supervisor")
graph.add_edge("editor", "supervisor")
# Complete leads to END
graph.add_edge("complete", END)
```

The `END` special node marks where the workflow terminates.

Lastly, we'll compile and run:

```
# Compile the graph
multi_agent_app = graph.compile()
print("Multi-agent graph built and compiled!")
# Test the multi-agent system
test_request = "Write a brief article about the benefits of multi-agent
AI systems"
# Run the multi-agent workflow
result = multi_agent_app.invoke({
    "user_request": test_request,
    "messages": []
})
print(result.get("final_response", result.get("edited_content", "No
output")))
```

The `compile()` method validates your graph and returns an executable application. Calling `invoke()` with an initial state starts the workflow.

Conflict Resolution

What happens when agents disagree? In a research team, two agents might find contradictory information. In a review pipeline, an editor might reject content the writer thought was finished.

The simplest approach is supervisor arbitration. When conflicts arise, escalate to the supervisor for a decision. The supervisor can request clarification, ask for additional work, or make a judgment call based on the available information.

For more sophisticated conflict handling, you can build explicit reconciliation steps. After parallel research, a synthesis step compares findings and resolves contradictions. After editing, a revision step incorporates feedback. These steps can be their own agents or integrated into existing ones.

Error Handling

Multi-agent systems have more failure modes than single agents. Any worker can fail, and failures can cascade. Robust error handling is essential.

The first line of defense is catching exceptions in worker invocations and deciding how to proceed. You might retry with the same worker, try a backup worker, skip the failed step if it's optional, or, when the step is truly optional, continue without it, returning a response that notes the gap rather than blocking entirely:

```
def handle_worker_failure(worker: str, error: Exception, state:
TeamState):
    """Handle when a worker agent fails."""
```

```
    logger.error(f"Worker {worker} failed: {error}")

    # Retry if it's a transient error
    if is_transient_error(error):
        return retry_worker(worker, state)

    # Try backup if available
    if has_backup(worker):
        return delegate_to_backup(worker, state)

    # Skip if optional
    if is_optional_step(worker):
        return continue_without(worker, state)

    # Otherwise fail gracefully
    return fail_gracefully(worker, error, state)
```

Detecting and Breaking Stuck Loops

A pernicious failure mode is agents getting stuck in loops. The supervisor keeps delegating to the researcher, the researcher keeps returning unsatisfactory results, and the supervisor keeps delegating again. Without intervention, this continues forever (or until you run out of money).

Build loop detection into your coordination logic:

```
def detect_stuck(state: TeamState) -> bool:
    """Detect if the team is stuck in a loop."""

    recent_actions = state.get("action_history", [])[-10:]

    # Same agent called 4+ times in a row
    if len(recent_actions) >= 4:
        last_four = [a["agent"] for a in recent_actions[-4:]]
        if len(set(last_four)) == 1:
            return True

    # Cycling between same two agents
    if len(recent_actions) >= 6:
        agents = [a["agent"] for a in recent_actions[-6:]]
        if len(set(agents)) == 2:
            return True

    return False
```

The fix works by adding an `action_history` list to your shared state and appending every routing decision to it as the workflow runs. The `detect_stuck` function then examines the tail of that history, looking for two signatures.

The first is a stuck-on-one-agent situation: if the same agent appears in the last four consecutive slots, the system is spinning, meaning the supervisor keeps routing back to the same worker no matter what it returns. The second is a two-agent ping-pong: if only two distinct agents appear across the last six actions, the supervisor and a worker are bouncing the task between themselves without making progress. When either condition is true, `detect_stuck` returns `True` and you force the system out of the loop, either by skipping the stuck step and working with what you have, by returning a partial result to the user, or by surfacing the impasse explicitly so the user can intervene.

Multi-Agent Patterns and Use Cases

Let's look at concrete examples of multi-agent systems in practice. These patterns show up across industries and use cases.

Report Generation Teams

Generating comprehensive reports is a natural fit for multi-agent collaboration in general, and the supervisor pattern in particular, because there's a clear orchestrator role, distinct specialists, and the kind of adaptive routing that a rigid pipeline can't handle. A typical team might include a researcher that gathers data and sources, an analyst that interprets data and identifies insights, a writer that structures and drafts the report, an editor that polishes and fact-checks, and, optionally, a visualizer that creates charts and diagrams.

The workflow usually proceeds from research to analysis to writing to editing. But a supervisor can adapt: if the analyst identifies gaps, loop back to research. If the editor finds factual errors, send specific sections back for revision.

Code Generation Workflows

Code generation benefits from specialization too and maps well to a pipeline architecture because the stages are well-defined, the order is fixed, and each agent's output is exactly what the next agent needs as input. Consider a team with a planner that breaks down the coding task into steps, an implementer that writes the actual code, a reviewer that checks for bugs and style issues, and a tester that generates and runs test cases.

The planner analyzes requirements and produces a high-level design. The implementer turns that design into code. The reviewer catches issues the implementer missed. The tester validates that the code actually works. This mirrors how professional development teams operate.

Where this gets tricky is when the reviewer finds a fundamental design flaw. A pipeline can't send work back upstream, so you'd need to escalate to a supervisor layer to handle that case cleanly.

Content Creation Workflows

Marketing and content teams can leverage multi-agent patterns for campaign creation. Content campaigns often call for a hybrid of the pipeline and supervisor architectures: a pipeline moves work through the major phases in order, while a supervisor coordinates the specialists within each phase. An ideation agent brainstorms concepts. A research agent validates ideas and gathers supporting data. A content agent produces drafts for different channels. A brand agent ensures consistency with brand guidelines.

For multi-channel campaigns, you might have specialists for each channel: a social media agent that knows platform conventions, an email agent that understands inbox dynamics, and a long-form agent for blog posts and articles. A campaign coordinator ensures consistent messaging across channels.

Parallel Research Patterns

When research tasks are independent, parallelize them. Instead of one researcher doing three searches sequentially, spawn three researchers working simultaneously. Gather their results and synthesize.

This parallel execution isn't a separate architecture; it's a technique you apply within a supervisor or pipeline to eliminate unnecessary sequential waits. When research tasks are independent, parallelize them. This technique dramatically reduces latency for research-heavy workflows. If each search takes 5 seconds, sequential searches take 15 seconds, while parallel searches take 5 seconds plus coordination overhead.

LangGraph makes parallel execution straightforward. Define multiple entry points or use fan-out edges that route to multiple nodes simultaneously. A join node collects results and continues.

Visualization and Debugging with LangSmith Studio

Multi-agent systems are harder to debug than single agents. When something goes wrong, you need to figure out which agent misbehaved, what inputs it received, and how its outputs affected downstream agents. LangSmith Studio is an essential tool for this.

What Is LangSmith Studio v2?

LangSmith Studio is LangChain's platform for observability, debugging, and evaluation of LLM applications. The v2 release (2025) added significantly enhanced visualization capabilities, including graph mode for viewing agent execution as interactive diagrams.

When your agents run with LangSmith tracing enabled, every LLM call, tool invocation, and state change is logged. Studio lets you explore these traces visually, understand execution flow, and identify exactly where things went wrong.

Graph Mode vs. Chat Mode

Studio offers two primary viewing modes. Chat mode shows the conversation as a linear sequence of messages, much like you'd see in a chat interface. This is great for understanding the user experience and following the dialogue flow.

Graph mode (see Figure 5.5) shows execution as an interactive diagram. You see nodes representing agents and decision points, edges showing how control flows between them, and you can click any node to inspect its inputs, outputs, and internal state. For multi-agent systems, graph mode is essential.

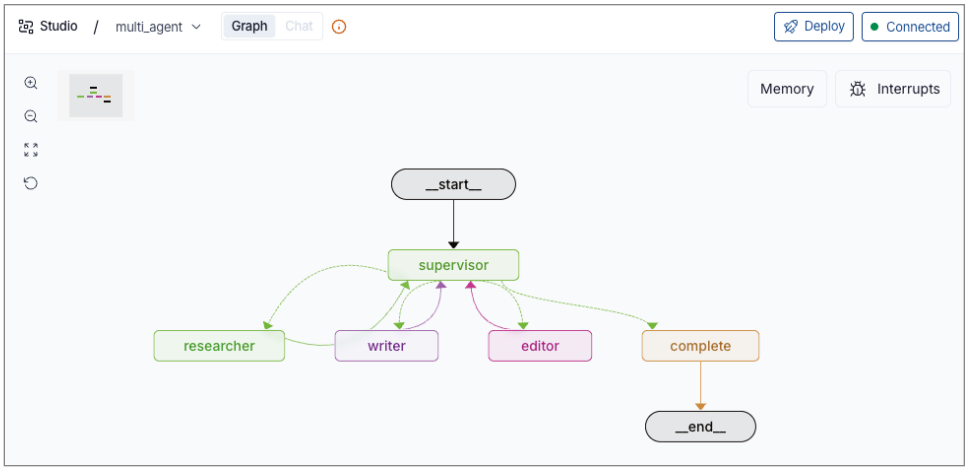

Figure 5.5: LangSmith Studio's interactive graph mode.

The graph view makes coordination visible. You can see the supervisor receiving a request, delegating to the researcher, getting results, delegating to the writer, reviewing the draft, delegating to the editor, and finally responding. The entire orchestration unfolds visually.

Debugging Multi-Agent Interactions

When a multi-agent workflow produces bad results, start by identifying which agent was the source. Look at the graph view and trace the flow. Where did good inputs become bad outputs?

Common debugging patterns include examining supervisor decisions (did it choose the right workers?), inspecting handoff contents (did enough context pass between agents?), checking for loops (did the same agent get called repeatedly?), and validating worker outputs (did each specialist produce quality results?).

Studio's time travel debugging lets you click on any point in the execution and see the exact state at that moment. You can modify inputs and rerun from that point, testing hypotheses about what went wrong without rerunning the entire workflow.

Testing and Iteration

While Studio is great for post-hoc debugging, it's also a development tool. Run your multi-agent system with test inputs, examine the traces, identify improvements, and iterate.

Build a habit of checking Studio traces during development, not just when things break. You'll often spot inefficiencies (unnecessary delegations, bloated contexts) and opportunities (places where parallelization would help) that wouldn't surface any other way.

Building StudyBuddy v5

Now it's time to put all of this into practice. You're going to transform StudyBuddy from a single-agent RAG system into a full multi-agent architecture using the supervisor pattern.

Where We Left Off

In Chapter 4, you built StudyBuddy v4: an agentic RAG system using LangGraph. It could analyze queries, decide when to search the knowledge base, reflect on its answers, and generate flashcards when explaining important concepts. All of this happened in a single agent with a state machine controlling its behavior.

That single agent is doing a lot. It's tutoring (explaining concepts), generating content (creating flashcards), and evaluating quality (deciding whether flashcards are good). These are genuinely different tasks that benefit from specialization.

What We're Adding

StudyBuddy v5 splits into a team of specialized agents, each focused on doing one thing well:

- **Tutor agent:** Explains concepts conversationally. This is the friendly expert who helps you understand difficult material.
- **Card Generator agent:** Creates flashcards from explanations and source materials. It focuses entirely on producing high-quality cards.
- **Quality Checker agent:** Validates that flashcards are clear, accurate, and useful. It acts as the editor for generated content.
- **Learning Coordinator (Supervisor):** Orchestrates the team. It decides which agents to engage based on what the student needs.

This architecture focuses on the learning experience. When you ask a question, the Tutor explains. When flashcards would help, the Coordinator engages the Card Generator, then the Quality Checker validates them before showing them to the student—all orchestrated seamlessly.

Defining the Agent Team

We're using LangGraph directly to build the StudyBuddy agent team using a supervisor architecture. It gives us full control over the coordination logic while keeping the code clean and understandable.

Each agent is defined as a module with its own prompt, LLM configuration, and functions.

Let's start with the Tutor:

```
# api/agents/tutor.py
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

TUTOR_PROMPT = """You are StudyBuddy's Tutor, an expert at explaining
AI engineering concepts clearly and engagingly.

Your job: Help students understand concepts from the reference guides.

Guidelines:
- Use clear, conversational language
- Start with the big picture, then details
- Use analogies to make abstract concepts concrete
- Check understanding by asking follow-up questions
- Reference specific sources when explaining
- Keep explanations focused and digestible
```

```
When given context from the knowledge base, use it to provide accurate,
sourced explanations. If no context is provided, use your general
knowledge
but be clear about what's from the reference materials vs general
knowledge."""


def create_tutor_agent(model_name: str = "gpt-4o"):
    """Create the Tutor agent."""
    return ChatOpenAI(model=model_name, temperature=0.7)


def tutor_explain(
    llm: ChatOpenAI,
    query: str,
    context: str = "",
    card_context: dict | None = None,
) -> str:
    """Have the Tutor explain a concept."""
    system_content = TUTOR_PROMPT

    # Add card context if student is studying a specific flashcard
    if card_context:
        system_content += f"""

The student is currently studying a flashcard:
- Topic: {card_context.get('topic', 'Unknown')}
- Question: {card_context.get('question', '')}
- Answer: {card_context.get('answer', '')}

Focus your explanation on helping them understand this specific concept
better."""

    user_content = f"Student question: {query}"
    if context:
        user_content += f"""

Reference material:
{context}

Use this reference material to provide an accurate, sourced
explanation."""

    messages = [
        SystemMessage(content=system_content),
        HumanMessage(content=user_content),
    ]

    response = llm.invoke(messages)
    return response.content
```

The Tutor uses `gpt-4o` with a higher temperature (0.7) for conversational quality. It has access to the knowledge base through the `context` parameter and can adapt its explanations when students are studying specific flashcards.

Now the Card Generator:

```
# api/agents/card_generator.py
import json
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

CARD_GENERATOR_PROMPT = """You are StudyBuddy's Card Generator,
specialized in creating focused, recallable flashcards.

CRITICAL RULES:
1. ONE testable fact per card - never combine multiple questions
2. Answer must be 1-3 sentences maximum (under 200 characters preferred)
3. Question cannot contain "and" connecting two different concepts
4. This is quick recall practice - avoid essay-length explanations

BAD EXAMPLES (compound questions - never do this):
- "What is X and what are its advantages?"
- "Explain the role of A and B in the system"
- "What are the key benefits and drawbacks?"

GOOD EXAMPLES (atomic, focused):
- "What is the primary purpose of X?"
- "What problem does X solve?"
- "How does X differ from Y?"

Output format - respond with ONLY a JSON array:
[
    {
        "question": "Clear, specific question testing ONE thing",
        "answer": "1-3 sentence answer, under 200 characters if
possible",
        "topic": "Main topic or concept",
        "difficulty": "basic|intermediate|advanced"
    }
]

Generate 3-5 atomic flashcards per request."""


def create_card_generator_agent(model_name: str = "gpt-4o-mini"):
    """Create the Card Generator agent."""
    return ChatOpenAI(model=model_name, temperature=0.3)


def generate_cards(
    llm: ChatOpenAI,
```

```
    topic: str,
    context: str = "",
    explanation: str = "",
) -> list[dict]:
    """Generate flashcards for a topic."""
    user_content = f"Create flashcards for: {topic}"

    if explanation:
        user_content += f"\n\nBased on this explanation:\n{explanation}"

    if context:
        user_content += f"\n\nReference material:\n{context}"

    messages = [
        SystemMessage(content=CARD_GENERATOR_PROMPT),
        HumanMessage(content=user_content),
    ]

    response = llm.invoke(messages)

    # Parse the JSON response
    try:
        content = response.content.strip()

        # Handle markdown code blocks
        if content.startswith("```"):
            content = content.split("```")[1]
            if content.startswith("json"):
                content = content[4:]
            content = content.strip()

        cards = json.loads(content)

        # Ensure it's a list
        if isinstance(cards, dict):
            cards = [cards]

        return cards

    except json.JSONDecodeError:
        return []
```

Notice we're using `gpt-4o-mini` with a lower temperature (0.3) for the Card Generator. Card generation benefits from structured output capabilities and determinism. The prompt explicitly forbids compound questions, a common failure mode in which cards try to test multiple concepts at once.

The Quality Checker validates generated cards:

```
# api/agents/quality_checker.py
import json
```

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

QUALITY_CHECKER_PROMPT = """You are StudyBuddy's Quality Checker,
ensuring flashcards meet learning standards.

Your job: Evaluate flashcard quality and suggest improvements.

Evaluation criteria:
1. Clarity: Is the question unambiguous?
2. Accuracy: Is the answer factually correct?
3. Completeness: Does the answer fully address the question?
4. Atomicity: Does it test exactly one concept?
5. Usefulness: Will this help the student learn?

Output format (JSON only):
{
    "approved": true/false,
    "score": 1-5,
    "issues": ["list of problems if any"],
    "suggestions": ["improvements if not approved"],
    "revised_card": null or {"question": "...", "answer": "...",
"topic": "..."}
}

Be strict but fair. Cards should be genuinely useful for learning.
If a card has minor issues, provide a revised version rather than
rejecting.
Only reject cards with fundamental problems (vague, incorrect, or
confusing)."""


def create_quality_checker_agent(model_name: str = "gpt-4o"):
    """Create the Quality Checker agent."""
    return ChatOpenAI(model=model_name, temperature=0.2)


def check_cards_batch(llm: ChatOpenAI, cards: list[dict]) -> list[dict]:
    """Check quality of multiple cards, returning only approved ones."""
    approved = []

    for card in cards:
        user_content = f"""Evaluate this flashcard:

Question: {card.get('question', '')}
Answer: {card.get('answer', '')}
Topic: {card.get('topic', 'Unknown')}

Provide your evaluation as JSON."""
```

```
            messages = [
                SystemMessage(content=QUALITY_CHECKER_PROMPT),
                HumanMessage(content=user_content),
            ]

            response = llm.invoke(messages)

            try:
                content = response.content.strip()
                if content.startswith("```"):
                    content = content.split("```")[1]
                    if content.startswith("json"):
                        content = content[4:]
                    content = content.strip()

                evaluation = json.loads(content)

                if evaluation.get("approved"):
                    approved.append(card)
                elif evaluation.get("revised_card"):
                    # Use the revised version
                    revised = evaluation["revised_card"]
                    revised["topic"] = card.get("topic", "Unknown")
                    approved.append(revised)

            except json.JSONDecodeError:
                # Default to approved if parsing fails (be lenient)
                approved.append(card)

    return approved
```

The Quality Checker does three things worth understanding in detail. First, it evaluates against five explicit criteria baked into the prompt—clarity, accuracy, completeness, atomicity, and usefulness—which gives the LLM a concrete rubric rather than letting it freelance. Second, it uses `gpt-4o` with a very low temperature (0.2) because this is evaluation work where consistency matters more than creativity. You want the model to apply the same standard repeatedly. Third, and most importantly, it has a repair mode. When a card fails, `check_cards_batch` checks for a `revised_card` in the response and uses that instead of discarding the original. That single design decision is what makes the Quality Checker a genuine quality improver rather than just a filter because it prevents good concepts from being lost due to minor wording issues.

Creating the Supervisor

The Supervisor decides which agent to route requests to:

```
# api/agents/supervisor.py
import json
```

```
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage, HumanMessage

SUPERVISOR_PROMPT = """You are StudyBuddy's Learning Coordinator,
orchestrating a team of specialized agents.

Your team:
- TUTOR: Explains concepts conversationally. Use for questions,
  confusion, or requests to learn about topics.
- CARD_GENERATOR: Creates flashcards. Use after tutoring explains
  something the student should remember.
- QUALITY_CHECKER: Validates flashcards. Always use after generating
  cards before showing to student.

Workflow patterns:
1. Learning mode: question → TUTOR explains → CARD_GENERATOR creates →
   QUALITY_CHECKER validates → respond with explanation and cards
2. Direct response: Simple greetings, clarifications, or meta-questions

Decision rules:
- Questions about concepts → TUTOR first
- "Create flashcards for X" → CARD_GENERATOR (then QUALITY_CHECKER)
- After TUTOR explains a key concept → consider CARD_GENERATOR
- Never show cards that haven't passed QUALITY_CHECKER
- Simple greetings/thanks → respond directly (no agent needed)

Output format (JSON only):
{
    "next_agent": "tutor|card_generator|quality_checker|respond",
    "reasoning": "Brief explanation of your decision",
    "task": "Specific instruction for the chosen agent (if not respond)"
}

For "respond", also include:
{
    "next_agent": "respond",
    "reasoning": "...",
    "response": "Your direct response to the student"
}"""


def create_supervisor_agent(model_name: str = "gpt-4o"):
    """Create the Supervisor agent."""
    return ChatOpenAI(model=model_name, temperature=0.3)


def route_request(
    llm: ChatOpenAI,
    query: str,
    context: dict | None = None,
```

```
) -> dict:
    """Decide which agent should handle the request."""
    context = context or {}

    user_content = f"Student message: {query}"

    if context.get("current_mode"):
        user_content += f"\nCurrent mode: {context['current_mode']}"

    if context.get("pending_cards"):
        user_content += f"\nPending cards awaiting quality check:
{len(context['pending_cards'])}"

    if context.get("recent_explanation"):
        user_content += "\nNote: Tutor just provided an explanation.
Consider generating flashcards."

    if context.get("card_context"):
        card = context["card_context"]
        user_content += f"\nStudent is studying flashcard: {card.
get('question', '')[:50]}..."

    user_content += "\n\nDecide which agent should handle this, or
respond directly."

    messages = [
        SystemMessage(content=SUPERVISOR_PROMPT),
        HumanMessage(content=user_content),
    ]

    response = llm.invoke(messages)

    try:
        content = response.content.strip()
        if content.startswith("```"):
            content = content.split("```")[1]
            if content.startswith("json"):
                content = content[4:]
            content = content.strip()

        decision = json.loads(content)

        # Validate next_agent
        valid_agents = ["tutor", "card_generator", "quality_checker",
"respond"]
        if decision.get("next_agent") not in valid_agents:
            decision["next_agent"] = "tutor"

        return decision
```

```
    except json.JSONDecodeError:
        return {
            "next_agent": "tutor",
            "reasoning": "Could not parse routing decision, defaulting
to tutor",
            "task": query,
        }
```

The Supervisor is StudyBuddy's Learning Coordinator, the agent that decides whether a student's message needs a Tutor response, card generation, quality checking, or a direct reply. What makes it interesting is where that routing intelligence lives: not in branching Python logic, as you might expect, but almost entirely in the prompt. The `SUPERVISOR_PROMPT` defines the agent roster, the workflow patterns (learning mode vs. direct response), and the decision rules. The LLM reads all of that, looks at the current conversation state, and returns a JSON object telling the system what to do next. The Python code is just scaffolding for invoking that decision and acting on the result. This is characteristic of the supervisor pattern: the coordination intelligence is in the prompt, not the control flow.

Shared State

All agents share state through a `TypedDict`:

```
# api/state.py
from typing import TypedDict, Annotated, Literal
from langgraph.graph.message import add_messages


class StudyBuddyState(TypedDict):
    """Shared state for the multi-agent graph."""

    # Conversation history
    messages: Annotated[list, add_messages]

    # Current mode
    current_mode: Literal["learning", "practice"]

    # Query from user
    query: str

    # Flashcard pipeline
    pending_cards: list[dict]   # Cards awaiting quality check
    approved_cards: list[dict]  # Cards that passed quality check

    # Context
    current_topic: str
```

```
    card_context: dict | None   # Current flashcard being studied

    # Response
    response: str

    # Supervisor routing
    next_agent: str | None      # Which agent to route to next
```

The state tracks everything agents need to coordinate: the conversation, flashcard pipeline, current topic, and routing decisions.

The Search Tool

Before building the graph, we need a way to search the knowledge base:

```
def search_materials(query: str, k: int = 4) -> str:
    """Search the knowledge base for relevant content."""
    results = vector_store.similarity_search(query, k=k)

    if not results:
        return ""

    formatted = []
    for doc in results:
        source = doc.metadata.get("source", "Unknown")
        formatted.append(f"[{source}]:\n{doc.page_content}")

    return "\n\n---\n\n".join(formatted)
```

This function returns formatted search results with source attribution, which agents can include in their context.

Building the Graph

Now we wire up the complete multi-agent graph:

```
# api/index.py
from langgraph.graph import StateGraph, END
from .state import StudyBuddyState
from .agents.tutor import create_tutor_agent, tutor_explain
from .agents.card_generator import create_card_generator_agent,
generate_cards
from .agents.quality_checker import create_quality_checker_agent,
check_cards_batch
from .agents.supervisor import create_supervisor_agent, route_request,
should_generate_cards

# Create the specialized agents
tutor_llm = create_tutor_agent("gpt-4o")
```

```
card_generator_llm = create_card_generator_agent("gpt-4o-mini")
quality_checker_llm = create_quality_checker_agent("gpt-4o")
supervisor_llm = create_supervisor_agent("gpt-4o")
```

Node Implementations

Each node wraps its agent and handles state updates:

```
def supervisor_node(state: StudyBuddyState) -> dict:
    """Route the request to the appropriate agent."""
    query = state.get("query", "")

    context = {
        "current_mode": state.get("current_mode", "learning"),
        "pending_cards": state.get("pending_cards", []),
        "card_context": state.get("card_context"),
    }

    # Check if we just got an explanation (for auto card generation)
    if state.get("response") and should_generate_cards(state.get
("response", "")):
        context["recent_explanation"] = True

    decision = route_request(supervisor_llm, query, context)

    return {"next_agent": decision.get("next_agent", "respond")}


def tutor_node(state: StudyBuddyState) -> dict:
    """Have the Tutor explain a concept."""
    query = state.get("query", "")
    card_context = state.get("card_context")

    # Search for relevant context
    context = search_materials(query)

    # Get the explanation
    explanation = tutor_explain(tutor_llm, query, context, card_context)

    return {"response": explanation, "current_topic": query[:50]}


def card_generator_node(state: StudyBuddyState) -> dict:
    """Generate flashcards."""
    topic = state.get("current_topic", "AI Engineering")
    explanation = state.get("response", "")

    # Search for additional context
    context = search_materials(topic)
```

```
    # Generate cards
    cards = generate_cards(card_generator_llm, topic, context,
explanation)

    return {"pending_cards": cards}


def quality_checker_node(state: StudyBuddyState) -> dict:
    """Check quality of pending cards."""
    pending = state.get("pending_cards", [])

    if not pending:
        return {"pending_cards": [], "approved_cards": []}

    approved = check_cards_batch(quality_checker_llm, pending)

    return {"pending_cards": [], "approved_cards": approved}


def respond_node(state: StudyBuddyState) -> dict:
    """Final response node - just passes through."""
    return {}
```

Routing Functions

Conditional edges determine the flow between nodes:

```
def route_after_supervisor(state: StudyBuddyState) -> str:
    """Route based on supervisor decision."""
    next_agent = state.get("next_agent", "respond")

    if next_agent in ["tutor", "card_generator", "quality_checker"]:
        return next_agent

    return "respond"


def route_after_tutor(state: StudyBuddyState) -> str:
    """After tutor, check if we should generate cards."""
    response = state.get("response", "")

    if should_generate_cards(response):
        return "card_generator"

    return "respond"


def route_after_card_generator(state: StudyBuddyState) -> str:
    """After generating cards, always check quality."""
```

```
    pending = state.get("pending_cards", [])

    if pending:
        return "quality_checker"

    return "respond"
```

The `should_generate_cards` function is a simple heuristic that checks if an explanation is substantive enough to warrant flashcard generation:

```
def should_generate_cards(explanation: str) -> bool:
    """Heuristic to decide if an explanation warrants flashcard
generation."""
    if len(explanation) < 200:
        return False

    educational_markers = [
        "is a", "are ", "means", "refers to", "defined as",
        "key concept", "important", "remember", "example",
        "first", "second", "third", "steps", "types",
    ]

    lower_explanation = explanation.lower()
    return any(marker in lower_explanation for marker in
educational_markers)
```

Assembling the Graph

Now we put it all together. The StudyBuddy v5 workflow follows this pattern:

1. Supervisor receives the user query and decides which specialist to engage.
2. Tutor explains concepts (can automatically trigger card generation for substantive explanations).
3. Card Generator creates flashcards from explanations.
4. Quality Checker validates cards before showing them to students.
5. Respond node returns the final output.

```
def build_studybuddy_graph():
    """Build the complete multi-agent graph."""
    graph = StateGraph(StudyBuddyState)

    # Add nodes
    graph.add_node("supervisor", supervisor_node)
    graph.add_node("tutor", tutor_node)
    graph.add_node("card_generator", card_generator_node)
    graph.add_node("quality_checker", quality_checker_node)
```

```
    graph.add_node("respond", respond_node)

    # Entry point
    graph.set_entry_point("supervisor")
```

The key difference from the simple research team is that the Tutor can automatically trigger card generation if the explanation seems substantive enough to warrant it. This creates a more fluid learning experience.

```
    # Supervisor routes to workers
    graph.add_conditional_edges(
        "supervisor",
        route_after_supervisor,
        {
            "tutor": "tutor",
            "card_generator": "card_generator",
            "quality_checker": "quality_checker",
            "respond": "respond",
        },
    )

    # Tutor can trigger card generation
    graph.add_conditional_edges(
        "tutor",
        route_after_tutor,
        {"card_generator": "card_generator", "respond": "respond"},
    )
```

The route_after_tutor function checks if the explanation was educational enough (using a heuristic like length and content markers) to warrant flashcard generation.

```
    # Card generator always goes to quality checker
    graph.add_conditional_edges(
        "card_generator",
        route_after_card_generator,
        {"quality_checker": "quality_checker", "respond": "respond"},
    )

    # Quality checker goes to respond
    graph.add_edge("quality_checker", "respond")

    # Respond ends the graph
    graph.add_edge("respond", END)

    return graph.compile()
```

The graph structure is: supervisor at the hub, workers around the edges, with conditional routing determining the flow. The Tutor can trigger card generation automatically for substantive explanations, and cards always pass through the Quality Checker before reaching the student.

The FastAPI Backend

Here's the FastAPI backend:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import Optional

app = FastAPI()

# Initialize the graph
studybuddy = build_studybuddy_graph()


class ChatRequest(BaseModel):
    message: str
    chapter_id: Optional[int] = None
    scope: Optional[str] = None
    card_context: Optional[dict] = None


class ChatResponse(BaseModel):
    reply: str


@app.post("/api/chat", response_model=ChatResponse)
def chat(request: ChatRequest):
    """Handle chat requests through the multi-agent system."""
    try:
        # Build the initial state
        initial_state = {
            "query": request.message,
            "messages": [],
            "current_mode": "learning",
            "pending_cards": [],
            "approved_cards": [],
            "current_topic": "",
            "card_context": request.card_context,
            "response": "",
            "next_agent": None,
        }
```

```
        result = studybuddy.invoke(initial_state)

        return ChatResponse(
            reply=result.get("response", "I couldn't generate a
response."),
        )

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

Testing Multi-Agent Coordination

Testing multi-agent systems requires two distinct approaches. Individual agent tests are fast and deterministic: you call each agent function directly with a known input and assert on the output, using whatever mocking strategy keeps LLM calls out of your unit test suite. Coordination tests are different because they run the full graph with real or stubbed agent calls and verify that routing decisions, state transitions, and the overall workflow produce the correct result. The following test suite covers both: the first three tests (`test_tutor_explains_concept`, `test_card_generator_produces_valid_json`, and `test_quality_checker_rejects_bad_cards`) target individual agents, while `test_full_learning_workflow` exercises the complete supervisor-coordinated pipeline end to end:

```
import pytest

class TestStudyBuddyV5:

    def test_tutor_explains_concept(self):
        """Tutor should search and explain."""
        explanation = tutor_explain(
            tutor_llm,
            query="Explain RAG",
            context=search_materials("RAG")
        )

        assert "retrieval" in explanation.lower()
        assert len(explanation) > 100

    def test_card_generator_produces_valid_json(self):
        """Card generator should output parseable cards."""
        cards = generate_cards(
            card_generator_llm,
            topic="embeddings",
            context=search_materials("embeddings")
        )
```

```
        assert len(cards) > 0
        assert all("question" in c for c in cards)
        assert all("answer" in c for c in cards)

    def test_quality_checker_rejects_bad_cards(self):
        """Quality checker should catch problems."""
        bad_card = {
            "question": "What?",  # Too vague
            "answer": "Yes",      # Not helpful
            "topic": "Unknown"
        }

        approved = check_cards_batch(quality_checker_llm, [bad_card])
        # Either rejected or revised
        assert len(approved) == 0 or approved[0]["question"] != "What?"

    def test_full_learning_workflow(self):
        """Complete workflow: question -> explanation -> cards."""
        state = {
            "query": "Teach me about vector databases",
            "messages": [],
            "current_mode": "learning",
            "pending_cards": [],
            "approved_cards": [],
            "current_topic": "",
            "card_context": None,
            "response": "",
            "next_agent": None,
        }

        result = studybuddy.invoke(state)

        # Should have a response
        assert result.get("response")
        assert len(result["response"]) > 100
```

Using LangSmith Studio

With multi-agent systems, LangSmith Studio becomes invaluable. Enable tracing:

```
import os
os.environ["LANGSMITH_TRACING"] = "true"
os.environ["LANGSMITH_PROJECT"] = "studybuddy-v5"
```

Now run some requests and open Studio. You'll see the supervisor receiving queries, routing to specialists, collecting results, and making decisions. Click any node to inspect its inputs and outputs. Watch the coordination unfold.

When something goes wrong—a card gets rejected, an explanation misses the point—you can trace exactly what happened. Studio shows you the full execution graph, letting you identify which agent misbehaved and why.

A Note on the UI

The preceding code examples focus on the core multi-agent architecture. The actual StudyBuddy v5 implementation extends this with a flashcard-first user interface, where students see flashcards as their primary study mode, with chat available as a secondary feature for deeper exploration.

This required a few additions to the API layer:

- `/api/status` endpoint to check indexing progress.
- `/api/chapters` endpoint that parses the `topic-list.md` file for chapter/topic navigation.
- `/api/flashcard` endpoint that generates scoped flashcards using the Card Generator and Quality Checker agents.
- `card_context` parameter in the chat endpoint so the Tutor knows what flashcard you're studying.

The chat feature focuses on tutoring—answering questions and explaining concepts—while flashcard generation is handled separately by the dedicated `/api/flashcard` endpoint. This separation keeps chat responses clean and focused, just like v4.

Running Locally

StudyBuddy v5 runs with two terminals, one for the backend, one for the frontend.

Terminal 1 runs the backend:

```
cd v5-multi-agent
uv sync
uv run uvicorn api.index:app --reload --port 8000
```

Terminal 2 runs the frontend:

```
cd v5-multi-agent/frontend
npm install
npm run dev
```

Visit `http://localhost:3000`. The Next.js dev server proxies `/api/*` requests to the FastAPI backend on port 8000 (configured in `next.config.ts`).

Deploying to Vercel

To deploy v5 to Vercel, you need to update two files in the repo root. First, update `vercel.json` to point to the `v5-multi-agent` directory:

```
{
    "version": 2,
    "builds": [
        {
            "src": "v5-multi-agent/api/index.py",
            "use": "@vercel/python"
        },
        {
            "src": "v5-multi-agent/frontend/package.json",
            "use": "@vercel/next"
        }
    ],
    "routes": [
        { "src": "/api/(.*)", "dest": "/v5-multi-agent/api/index.py" },
        { "src": "/(.*)", "dest": "/v5-multi-agent/frontend/$1" }
    ],
    "git": {
        "deploymentEnabled": {
            "main": true
        }
    },
    "ignoreCommand": "bash scripts/should-deploy.sh"
}
```

Second, update `scripts/should-deploy.sh` to trigger on v5 (and `vercel.json`) changes:

```
#!/bin/bash
# Deploy if changes are in v5-multi-agent or vercel config
# Exit 1 (true) = proceed with build, Exit 0 (false) = skip build
git diff HEAD^ HEAD --name-only | grep -qE "^v5-multi-agent/|^vercel\
.json" && exit 1 || exit 0
```

If you haven't done so already, set your OpenAI API key environment variable in Vercel by running the following command from the project root and pasting the key when prompted:

```
vercel env add OPENAI_API_KEY
```

Then deploy:

```
vercel --prod
```

Or simply push to GitHub. Vercel will automatically deploy when changes are detected in the `v5-multi-agent` directory.

What's Next

StudyBuddy v5 demonstrates the power of multi-agent coordination. You have specialists doing what they're good at, a supervisor keeping everything organized, and a system that's more maintainable than a monolithic single agent.

But there's something missing: memory. Right now, StudyBuddy forgets everything between sessions. It doesn't remember what you've studied, what you struggle with, or how you like to learn. And without memory, we can't implement spaced repetition—scheduling reviews based on your learning history requires remembering that history.

In Chapter 6, you'll add persistent memory that makes StudyBuddy truly personal. You'll also add a Scheduler Agent that uses the SM-2 spaced repetition algorithm to determine what cards to review and when. Spaced repetition only makes sense with persistent memory, because tracking review history and calculating optimal intervals requires knowing what happened in previous sessions.

You've leveled up from single agent to team coordination. That's a big deal. Take a moment to appreciate what you've built: a system where multiple AI specialists collaborate to help students learn. That's real AI engineering.

Let's keep building.

CHAPTER

6

Agent Memory

Here's the thing about LLMs that catches a lot of people off guard: they don't remember your previous conversations. Each request starts fresh, with no memory of what came before. For simple question answering, statelessness is fine. But the moment you want to build something more sophisticated—a personal tutor that adapts to your learning style, a customer support agent that remembers your history, a research assistant that builds on previous work—you need memory. Real, persistent memory that survives across sessions and grows smarter over time.

In this chapter, we're going to solve that problem. You'll learn how to architect memory systems that give your agents genuine continuity. We'll cover the different types of memory, explore when to form memories during conversations versus after them, and get hands-on with LangGraph's native memory capabilities. We'll also address something that's just as important as memory itself: performance optimization through smart caching. Once your agent starts remembering things and generating content, you'll want to avoid doing the same expensive work twice.

By the end of this chapter, you'll understand how to build agents that learn from experience, adapt to individual users, and do it all efficiently. These features are what separate a demo from a product people actually want to use.

The Memory Problem in AI Agents

Let's start with the fundamental challenge. Large language models are stateless by design. When you send a request to an LLM, it processes your input, generates a response, and immediately forgets everything. The next request arrives in a completely blank slate. There's no persistent state, no accumulated knowledge, no memory of previous interactions.

Now, you might be thinking: "Wait, I've had multi-turn conversations with ChatGPT. It definitely remembers what I said earlier." And you're right, but that's an illusion created by the context window. Each message in your conversation gets packed into that context window and sent along with your new message. The model isn't actually remembering anything; it's just being shown the full conversation history every single time. The "memory" is in the input, not in the model.

This approach works great for short conversations, but it has hard limits. Context windows typically range from 128,000 up to 1,000,000 tokens depending on the model. Once your conversation exceeds that limit, older content gets summarized or truncated. Your carefully built context evaporates. The model starts losing track of things you discussed earlier, and the quality of responses degrades.

For agents that need to maintain state across sessions—remembering user preferences, learning from past interactions, building on previous work—the context window isn't enough. You need explicit memory systems that persist beyond the current conversation.

Why Persistent Memory Matters

Think about what persistent memory enables. A customer support agent that remembers your previous tickets, knows what solutions have already been tried, and understands your technical environment. A personal tutor that tracks which concepts you've mastered and which ones need more work. A research assistant that builds on previous analysis instead of starting from scratch every time.

Without memory, every interaction is a fresh start. Users have to re-explain context. Agents can't learn from experience. There's no personalization, no adaptation, no continuity. The experience feels robotic and disconnected, well, because it is.

With memory, agents become genuinely useful over time. They adapt to individual users. They avoid repeating mistakes. They build on what's already been accomplished. The interaction feels more like working with a knowledgeable colleague who actually understands your situation.

Memory vs. RAG: Understanding the Difference

If you've been following along from earlier chapters, you might wonder how memory differs from RAG. After all, both involve storing information and retrieving it when relevant. The distinction is important.

RAG systems retrieve from a shared knowledge base, such as documents, articles, and product information. The content is the same for all users. When someone asks a question, you search this corpus for relevant passages and inject them into the context. RAG answers the question: "What information exists about this topic?"

Memory systems store user-specific information from interactions: preferences, goals, past conversations, and things the user has shared about themselves. This content is unique to each user. Memory answers a different question: "What do I know about *this particular user*?"

In practice, you often use both together. RAG provides the knowledge base, or the facts about your domain. Memory provides personalization, or the understanding of who you're talking to and what they need. A tutoring agent might use RAG to retrieve information about machine learning concepts and memory to know that this particular student prefers detailed explanations with code examples and is currently preparing for a job interview.

The retrieval mechanisms are similar, but the content and purpose are distinct. RAG is about domain knowledge. Memory is about relationship knowledge.

Memory System Architecture

Now that we understand why memory matters, let's look at how to build it. Before building our memory store, we need the database infrastructure. We'll use SQLAlchemy as our ORM, which gives us clean Python abstractions over SQL. If you haven't already, add it to your project dependencies:

```
uv add sqlalchemy
```

For the examples in this chapter, we use a SQLite in-memory database because it requires no setup and runs anywhere. In production, you'd swap in PostgreSQL with a single connection string change (we'll cover that later in the chapter). Here's the setup:

```
import os
import uuid
import json
import hashlib
from datetime import datetime, UTC
from dotenv import load_dotenv
```

```
from sqlalchemy import (
    Column, String, Integer, DateTime, JSON, Index,
    create_engine, cast
)
from sqlalchemy.orm import Session, sessionmaker, declarative_base
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI
load_dotenv()
if not os.getenv("OPENAI_API_KEY"):
    raise ValueError("OPENAI_API_KEY not found in environment
variables")
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
Base = declarative_base()
engine = create_engine("sqlite:///:memory:")
SessionLocal = sessionmaker(autocommit=False, autoflush=False,
bind=engine)
```

Note: You can find all of this chapter's non-StudyBuddy code in the Jupyter notebook `chapter06.ipynb`, available in this book's GitHub repo.

The Memory model uses a namespace/key/value structure. Namespaces categorize memories (preferences, facts, goals, episodes). Keys identify specific memories within a category. Values store the actual data as JSON, giving you flexibility without schema migrations:

```
class Memory(Base):
    """User memory storage organized by namespace.
    Namespaces let you categorize memories for efficient retrieval:
    - preferences: User preferences and settings
    - facts: Known facts about the user
    - episodes: Summaries of past interactions
    - goals: User's stated objectives
    """
    __tablename__ = "memories"
    id = Column(String(36), primary_key=True)
    user_id = Column(String(50), nullable=False, index=True)
    namespace = Column(String(100), nullable=False, index=True)
    key = Column(String(200), nullable=False)
    value = Column(JSON, nullable=False)
    created_at = Column(DateTime, default=lambda: datetime.now(UTC))
    updated_at = Column(DateTime, default=lambda: datetime.now(UTC),
onupdate= lambda: datetime.now(UTC))
    # Composite index for efficient lookups
    __table_args__ = (Index("ix_memory_user_namespace", "user_id",
"namespace"),)
```

The composite index on (`user_id`, `namespace`) makes lookups fast when retrieving all memories of a particular type for a user.

The Memory Store Pattern

At the core of any memory system is a store abstraction that handles basic operations such as `put` (add a memory to the store), `get` (retrieve a memory from the store), and `search` (find a memory in the store). Rather than using a third-party library, we'll build a lightweight store class that wraps SQLAlchemy queries. This keeps dependencies minimal and makes the code easy to understand and modify.

The API is straightforward. You put memories into the store with a namespace, key, and value. You get memories back by namespace and key. You search within a namespace to find relevant memories. Here's the pattern:

```
class MemoryStore:
    """Simple memory store backed by SQLAlchemy."""
    def __init__(self, db: Session):
        self.db = db
```

The `put` method handles both creating new memories and updating existing ones. When you call `put` with a namespace and key that already exists, it updates rather than duplicates:

```
    def put(self, user_id: str, namespace: str, key: str, value: dict)
-> Memory:
        """Store or update a memory."""
        existing = (
            self.db.query(Memory)
            .filter_by(user_id=user_id, namespace=namespace, key=key)
            .first()
        )
        if existing:
            existing.value = value
            existing.updated_at = datetime.now(UTC)
            self.db.commit()
            return existing
        memory = Memory(
            id=str(uuid.uuid4()),
            user_id=user_id,
            namespace=namespace,
            key=key,
            value=value,
        )
        self.db.add(memory)
        self.db.commit()
        return memory
```

The `get` method retrieves a single specific memory when you know exactly which one you want:

```
    def get(self, user_id: str, namespace: str, key: str) -> dict |
None:
        """Get a specific memory by key."""
        memory = (
            self.db.query(Memory)
            .filter_by(user_id=user_id, namespace=namespace, key=key)
            .first()
        )
        return memory.value if memory else None
```

The `search` method lets you explore memories within a namespace or across all namespaces. Results are ordered by recency with the most recently updated memories appearing first:

```
    def search(self, user_id: str, namespace: str | None = None, limit:
int = 10) -> list[dict]:
        """Search memories, ordered by recency."""
        query = self.db.query(Memory).filter_by(user_id=user_id)
        if namespace:
            query = query.filter_by(namespace=namespace)
        query = query.order_by(Memory.updated_at.desc()).limit(limit)
        return [
            {
                "namespace": m.namespace,
                "key": m.key,
                "value": m.value,
                "updated_at": m.updated_at.isoformat() if m.updated_at
else None,
            }
            for m in query.all()
        ]
```

This pattern—`namespace` plus `key` equals `value`—is simple but surprisingly powerful. The `namespace` organizes memories into categories. The `key` identifies a specific memory within that category. The `value` stores the actual data as JSON.

Giving Agents Memory Tools

Your agent needs a way to interact with the memory store. The cleanest approach is to wrap store operations as tools that the agent can call. This lets the agent decide when to remember something and when to recall past context, so we give it the capability to store new information and retrieve existing memories.

First, we'll create a single instance of the store using our database session:

```
# Create store with database session
store = MemoryStore(db_session)
```

The `remember` tool lets the agent capture important information during a conversation. When should it be called? Whenever the user mentions preferences ("I need the examples to be more detailed"), shares facts about themselves ("I am studying for my AWS certification"), or expresses goals ("I want to understand machine learning basics"):

```
@tool
def remember(namespace: str, key: str, fact: str) -> str:
    """Store an important fact about the user or conversation.
    Args:
        namespace: Category like 'preferences', 'goals', or 'struggles'
        key: Identifier for this specific memory
        fact: The information to remember
    """
    store.put(
        user_id=current_user_id,
        namespace=namespace,
        key=key,
        value={"content": fact, "timestamp": datetime.now(UTC).isoformat()}
    )
    return f"I'll remember that: {fact}"
```

The `recall` tool searches for memories in a specific category. The agent might call this at the start of a session to refresh its context about the user:

```
@tool
def recall(namespace: str) -> str:
    """Recall memories from a specific category.
    Args:
        namespace: Category to search, like 'preferences' or 'goals'
    """
    memories = store.search(current_user_id, namespace=namespace, limit=5)
    if not memories:
        return f"No memories found in {namespace}."
    return "\n".join([
        f"- {m['key']}: {m['value'].get('content', m['value'])}"
        for m in memories
    ])
```

With these tools, your agent can decide on its own when to store information and when to search for it. The agent might store a preference when a user mentions one, or search for past context when answering a follow-up question.

Namespace Organization

Namespaces are how you organize memories in the store. Think of them as a hierarchical folder structure for user data. Each namespace groups related memories together, making retrieval efficient and logical.

For multiuser applications, the user ID is your first level of organization. Within each user's memories, namespaces categorize the content:

```
# Namespace conventions vary by agent type

# --- Personal tutor agent ---
"preferences"   # Learning style, communication preferences, settings
"facts"         # Known facts about the user (profession, background,
etc.)
"goals"         # What they're trying to achieve
"struggles"     # Topics or concepts they find difficult
"sessions"      # Summaries of past interactions

# --- Custom support agent ---
"account"       # Account details, plan, usage history
"tickets"       # Past issues and resolutions
"preferences"   # Contact preferences, notification settings
"environment"   # Tech stack, OS, integration details

# --- Research assistant ---
"interests"     # Research areas and standing questions
"sources"       # Trusted publications, preferred citation style
"sessions"      # Prior research threads and findings
```

When searching, you scope your query to a specific namespace. This keeps searches focused and fast. For example, a tutoring agent checking known weaknesses searches the "struggles" namespace rather than scanning everything the system knows about that user. Similarly, a customer support agent looking for prior tickets searches the "tickets" namespace. This organization also helps with data management because you can easily clear all session summaries without touching preferences or facts you want to keep long-term.

Advanced Retrieval Pattern

The most common retrieval pattern is searching within a namespace, ordered by recency because more recent memories are usually more relevant than old ones. That's how the `search` method added earlier to the `MemoryStore` class works.

For more sophisticated retrieval, you can add filtering by key patterns, date ranges, or even full-text search on the JSON content. SQLAlchemy's text matching makes this straightforward. Add this code to the `MemoryStore` class:

```
def search_by_content(self, user_id: str, search_term: str, limit: int =
10) -> list[dict]:
    """Search memories by content within the JSON value."""
    from sqlalchemy import cast, String
    query = (
        self.db.query(Memory)
```

```
        .filter_by(user_id=user_id)
        .filter(cast(Memory.value, String).ilike(f"%{search_term}%"))
        .order_by(Memory.updated_at.desc())
        .limit(limit)
    )
    return [{"key": m.key, "value": m.value} for m in query.all()]
```

This approach uses SQL pattern matching on the JSON content. It's not as sophisticated as vector similarity search, but it's fast, requires no additional infrastructure, and works well for keyword-based recall. For applications that need semantic search, you can add vector embeddings later; the namespace/key/value structure remains the same.

Types of Memory

Not all memories are created equal. Just as human memory has different types—facts you know, experiences you've had, skills you've learned—AI memory systems benefit from similar distinctions. Understanding these types helps you design systems that remember the right things in the right ways.

Semantic Memory: Facts and Knowledge

Semantic memory stores facts and knowledge about the user or domain. These are declarative statements: "The user works at a fintech startup." "They prefer Python over JavaScript." "Their company uses PostgreSQL." "They're preparing for AWS certification."

This type of memory drives personalization. When you know facts about a user, you can tailor your responses accordingly: technical explanations for the engineer, business-focused summaries for the executive, and examples using their preferred programming language.

Semantic memories tend to be relatively stable. Once you learn that a user prefers detailed explanations, that preference probably won't change next week. This stability makes semantic memory efficient: you store it once and reference it many times.

```
# Examples of semantic memory
store.put(
    user_id=current_user_id,
    namespace="facts",
    key="profession",
    value={"content": "Senior software engineer at a healthcare
company"}
)
store.put(
```

```
    user_id=current_user_id,
    namespace="preferences",
    key="explanation_style",
    value={"content": "Prefers code examples over abstract
descriptions"}
)
```

Episodic Memory: Past Experiences

Episodic memory records specific interactions and experiences. "Last Tuesday, we discussed transformer architectures and the user struggled with attention mechanisms." "Three weeks ago, they asked for help debugging a memory leak." "Yesterday, we created a study plan for their certification exam."

While semantic memory captures what you know about someone, episodic memory captures what you've done together. This enables learning from experience. If a particular explanation didn't land well last time, you can try a different approach. If a user asked about a topic before, you can build on that foundation rather than starting over.

Episodic memories are timestamped and contextual. They capture not just the content but the circumstances: when it happened, what was being discussed, how it went. This temporal aspect lets you weight recent experiences more heavily than distant ones.

```
# Examples of episodic memory
store.put(
    user_id=current_user_id,
    namespace="episodes",
    key=f"session_{session_id}",
    value={
        "content": "Discussed RAG implementation patterns",
        "timestamp": datetime.now(UTC).isoformat(),
        "outcome": "User understood chunking but needed more help with
retrieval",
        "topics": ["RAG", "chunking", "vector search"]
    }
)
```

Procedural Memory: Learned Behaviors

Procedural memory encodes how the agent should behave. These aren't facts about the user; they're patterns and rules the agent has learned to follow. "Always cite sources when answering technical questions." "Ask clarifying questions when queries are ambiguous." "Break down complex explanations into steps."

This type of memory shapes agent behavior rather than agent knowledge. Over time, an agent can learn that certain approaches work better with certain

users. Maybe this user responds well to analogies. Maybe that user gets frustrated by too many clarifying questions. Procedural memory captures these behavioral adaptations.

In practice, procedural memory often emerges from analyzing episodic memories. You notice patterns across many interactions and extract rules that should guide future behavior.

When to Use Each Type

The three memory types serve complementary purposes, and most production systems use all three. Semantic memory powers personalization with stable user facts. Episodic memory enables learning from experience with timestamped interaction records. Procedural memory shapes behavior with learned patterns and rules. For example:

- For a customer support agent, semantic memory might store the customer's product tier and technical environment. Episodic memory would track previous support tickets and their resolutions. Procedural memory might capture that this customer prefers brief, action-oriented responses rather than lengthy explanations.
- For a research assistant, semantic memory stores the researcher's domain expertise and current projects. Episodic memory records past research sessions and findings. Procedural memory encodes preferences such as "always include citations" or "summarize before diving into details."

The key is matching memory type to information type. Facts go in semantic memory. Events go in episodic memory. Behaviors go in procedural memory. When you keep these distinct, your memory system stays organized and your retrievals stay relevant.

Memory Formation Strategies

Knowing *what* to remember is only half the challenge. The other half is knowing *when* to create memories. There are two fundamental approaches: forming memories during the conversation (hot path) or extracting memories afterward (background). Each has distinct trade-offs, and many production systems use both.

Hot Path: Active Memory Formation

In the hot path approach, memory formation happens while the agent is actively responding. The agent decides in real time that something is worth remembering and stores it immediately. This might be triggered by explicit user statements

("Remember that I prefer Python") or by the agent recognizing important information during the conversation.

The implementation typically involves giving your agent a memory tool that it can invoke during conversations:

```
@tool
def remember_hotpath(fact: str) -> str:
    """
    Store an important fact about the user or conversation.
    Use when the user shares preferences, goals, or important context.
    """
    store.put(
        user_id=current_user_id,
        namespace="facts",
        key=str(uuid.uuid4()),
        value={
            "content": fact,
            "type": "semantic",
            "timestamp": datetime.now().isoformat()
        }
    )
    return f"I'll remember that: {fact}"
```

The advantage of hot path memory is immediacy. The agent has full context when deciding what to remember. It understands the nuance of the conversation, what seems important, what the user emphasized. The memory is stored with that rich contextual understanding.

The downside is latency. Every memory operation adds time to the response. The agent needs to spend tokens reasoning about what to remember, then execute the tool call to store it. For real-time conversational applications, this overhead can be noticeable.

Background: Passive Memory Extraction

Background extraction runs after a conversation ends, analyzing the full transcript to identify things worth remembering. This approach has zero latency impact on the conversation itself. After a conversation ends, a separate process analyzes the transcript and extracts memories. This process is essentially an offline summarization job: it receives the full conversation as input, an LLM reads the exchange to identify preferences, facts, commitments, or patterns worth keeping, and the process writes the resulting structured memories back into the user's store. Because it runs outside the request-response cycle, it can take as long as it needs and use a model as capable as necessary. The following function builds a prompt over the transcript, invokes the LLM, parses the structured extraction, and persists each extracted memory.

First, we need a helper to parse whatever the LLM returns. LLMs sometimes return malformed JSON, so we handle that gracefully:

```
def parse_extraction(extraction) -> list[dict]:
    """Parse LLM extraction into memory dicts."""
    try:
        return json.loads(extraction.content)
    except (json.JSONDecodeError, AttributeError):
        return []
```

The main `extract_memories_background` function sends the full conversation to an LLM with instructions on what to extract. The prompt tells it exactly what categories we care about: preferences, facts, topics, and any commitments:

```
def extract_memories_background(conversation: list[dict]):
    """Run after conversation ends to extract memories."""
    prompt = """Analyze this conversation and extract:
    1. User preferences mentioned
    2. Important facts shared
    3. Topics discussed
    4. Any commitments or follow-ups needed

    Return as a JSON array of objects with "type" and "content" fields.

    Conversation:
    {conversation}
    """

    extraction = llm.invoke(prompt.format(conversation=json.dumps
(conversation)))
    memories = parse_extraction(extraction)

    for memory in memories:
        store.put(
            user_id=current_user_id,
            namespace=memory.get("type", "facts"),
            key=str(uuid.uuid4()),
            value=memory
        )
```

Background extraction has clear advantages. It adds zero latency to conversations. It can analyze the full conversation holistically, identifying patterns that might not be obvious in any single message. It can use larger, more capable models for extraction without impacting response time.

The trade-off is delayed memory. Information from a conversation won't be available until the extraction process runs. If a user mentions their preference early in a session, the agent can't use that memory later in the same session because it hasn't been extracted yet.

Trade-Offs and Decision Framework

When should you use each approach? Hot path memory excels when information needs to be available immediately within the same conversation. If a user says, "I'm a visual learner," you want that preference to influence responses right away, not in the next session.

Background extraction excels at comprehensive analysis and long-term memory. After a tutoring session, you might want to analyze: What concepts did we cover? What did the student struggle with? How did the session go overall? These summary-level insights benefit from holistic analysis after the conversation ends.

Latency sensitivity matters too. Customer support chat that needs sub-second responses? Background extraction keeps things responsive. Research assistant where users expect thoughtful pauses? Hot path memory is more acceptable.

Hybrid Strategies

Most production systems use both approaches. Hot path for explicit memory requests and high-priority information. Background for comprehensive extraction and analysis. The combination gives you both benefits: immediate memory when it matters, plus thorough extraction for long-term learning.

A practical hybrid might work like this:

- During the conversation, the agent stores memories only when the user explicitly asks ("Remember that...") or when critical information surfaces ("My company deadline is Friday").
- After the conversation, a background job extracts additional insights, preferences, and patterns that weren't explicitly stored.

This approach keeps conversations fast while still building a comprehensive understanding of the user over time. The hot path handles urgent memory needs; the background handles everything else.

Memory Storage and Retrieval

Getting memories into the store is only half the equation. Getting them back out—at the right time, in the right context—is equally important. A memory system that stores everything but retrieves nothing useful is just a database with extra steps.

Storing Memories with Metadata

When you store a memory, include rich metadata beyond just the content. Timestamps let you weight by recency. Type labels distinguish semantic from

episodic. Source information helps with debugging. Topic tags enable efficient filtering.

```
# Rich memory storage with metadata
store.put(
    user_id=current_user_id,
    namespace="semantic",
    key=str(uuid.uuid4()),
    value={
        "content": "User is preparing for AWS Solutions Architect exam",
        "type": "semantic",
        "source": "explicit_mention",
        "timestamp": datetime.now().isoformat(),
        "topics": ["AWS", "certification", "career"],
        "confidence": 1.0
    }
)
```

This metadata becomes essential during retrieval. You might want only recent memories, or only memories about certain topics, or only high-confidence facts. The richer your metadata, the more precise your retrieval can be.

Semantic Search for Relevant Memories

The most common retrieval pattern is content-based search: given the current conversation context, find memories that might be relevant. In production, you'd use vector similarity search for this; here, we use keyword matching via `search_by_content` as a simpler stand-in that requires no additional infrastructure.

```
def get_relevant_memories(query: str, user_id: str, k: int = 5):
    """Retrieve memories semantically related to the query."""
    # In production, use vector similarity search
    # Here we use keyword matching as a simplified stand-in
    results = store.search_by_content(user_id, query, limit=k)
    return [result["value"] for result in results]
```

Text matching handles exact keyword overlap well. For variations in phrasing—for example, where the user asks about "machine learning" and you want to retrieve memories stored under "ML" or "neural networks"—you'd add vector embeddings. The namespace/key/value structure remains the same; only the search method changes.

Recency and Relevance Weighting

Pure semantic similarity isn't always optimal. A memory from yesterday is often more relevant than a memory from six months ago, even if the older one scores slightly higher on similarity. Combining recency with relevance produces better results.

First, we fetch a larger pool of candidates (20 instead of 5) so the reranking has more to choose from:

```
def get_memories_with_recency(query: str, user_id: str):
    """Get memories weighted by both relevance and recency."""
    results = store.search(user_id, limit=20)  # Get more candidates for
reranking
```

Next we calculate a decay score for each memory. The formula `1 / (1 + age_hours / 24)` gives us full weight for very recent memories, with gradual decay as they get older by the day:

```
    scored = []
    now = datetime.now(UTC)

    for r in results:
        updated = r.get("updated_at")
        if updated:
            timestamp = datetime.fromisoformat(updated)
        else:
            timestamp = now
        age_hours = (now - timestamp).total_seconds() / 3600
        recency_weight = 1 / (1 + age_hours / 24)  # Decay over days
        scored.append((recency_weight, r))
```

Finally, sort by the recency score and return only the top 5 results:

```
    scored.sort(reverse=True, key=lambda x: x[0])
    return [r["value"] for _, r in scored[:5]]
```

This decay function can be tuned to your use case. A customer support system might decay memories more slowly; issues from a month ago might still be relevant. A news-focused assistant might decay aggressively; yesterday's context matters more than last week's.

Memory Consolidation over Time

As memory accumulates, you'll want strategies for consolidation. Rather than keeping every individual memory forever, you can summarize patterns, merge similar memories, and archive or delete stale information.

Think of it like human memory. You don't remember every individual conversation from last year, but you remember the general patterns and important events. Memory consolidation transforms many specific memories into fewer, more useful summary memories.

A practical consolidation might run weekly: analyze all episodic memories from the past week, extract patterns ("User tends to ask follow-up questions

about implementation details"), store those patterns as new semantic memories, and archive the original episodes. The raw detail is preserved if needed, but the working memory stays focused on high-value insights.

Memory Integration Patterns

You've got memories in your store. Now the question becomes: How do you get them into your agent's context at the right time? There are several integration patterns, each with its own strengths.

Injection at Start

The simplest pattern is to load relevant memories at the start of every conversation and inject them into the system prompt. Before the agent sees the user's message, it already has context about who it's talking to.

```
def build_system_prompt(user_id: str, current_query: str):
    """Build a personalized system prompt with relevant memories."""

    memories = get_relevant_memories(current_query, user_id)

    memory_context = "\n".join([
        f"- {m.get('content', m)}" for m in memories
    ])

    return f"""You are a helpful assistant.

What you know about this user:
{memory_context}

Use this context to personalize your responses."""
```

This approach is reliable and predictable. The agent always has memory context available. There's no need for the agent to decide when to search because the relevant information is already there.

The trade-off is token usage. You're including memory context in every request, even when it's not needed. For users with extensive memory, this can eat into your context budget. You'll want to limit how many memories you inject and be thoughtful about relevance thresholds.

On-Demand Retrieval

An alternative is giving the agent a memory search tool and letting it decide when to retrieve. The agent might search memory at the start of a conversation to load user context; when the user's message contains words or phrases—such

as "like we discussed" or "last time I asked about this"—that are clear clues that relevant memories exist; or when the user references something from the past.

```
@tool
def recall_on_demand(topic: str) -> str:
    """
    Search memory for information about a topic.
    Use when you need to remember something about the user
    or from past conversations.
    """
    memories = store.search_by_content(current_user_id, topic, limit=5)

    if not memories:
        return "No relevant memories found."

    return "\n".join([f"- {m['value'].get('content', m['value'])}" for m
in memories])
```

On-demand retrieval is more token-efficient. The agent only loads memory when it actually needs it. That does require guiding the agent to recognize when memory would help. You handle this in the system prompt with explicit instructions about the signals to watch for, such as user references to past conversations, preference statements, or questions about prior work. If those instructions are vague, the agent won't search when it should, and relevant memories will go unused.

Proactive Memory

A more sophisticated pattern is proactive memory, where the system surfaces relevant memories without being asked. Based on the current conversation context, the system identifies memories that might be useful and injects them automatically.

```
def get_proactive_memories(conversation_context: str, user_id: str):
    """Find memories relevant to the current context that the user
hasn't explicitly requested."""

    related = store.search_by_content(user_id, conversation_context,
limit=5)

    # In production, filter by relevance score (e.g., score > 0.75)
    return [r["value"] for r in related]
```

Proactive memory can create delightful experiences. The agent "remembers" relevant context without being prompted. "Last time we discussed this topic, you mentioned being interested in the production deployment aspects. Want me to focus there again?"

The risk is over-triggering. If you surface too many memories, or memories that aren't actually relevant, the experience feels surveillance-like rather than helpful. High relevance thresholds help, but you'll need to tune based on user feedback.

Adapting Behavior Based on History

Beyond surfacing content, memory can also adapt how the agent behaves. If you know this user struggled with a concept before, spend more time on it. If you know they prefer brief responses, don't write essays. Memory enables dynamic personalization at the behavior level, not just the content level.

This kind of behavioral adaptation typically comes from procedural memories. "This user responds well to analogies." "This user gets frustrated when asked too many clarifying questions." "This user is an expert, so skip the basics." These patterns shape how the agent approaches interactions, making each conversation feel tailored.

Moving to PostgreSQL for Production

The SQLite in-memory database we've been using is perfect for learning and prototyping, but your carefully curated memories vanish the moment your process exits. For production applications, you need persistent storage that survives beyond a single session. PostgreSQL is the standard choice, and since we're using SQLAlchemy, switching requires only a connection string change. The next few sections show you how to set up PostgreSQL, but don't implement it. For a complete working example, refer to the "Building StudyBuddy v6" section, which uses PostgreSQL for all memory persistence.

Why PostgreSQL Everywhere

There's a philosophy in software development called "dev-prod parity," which is the idea that your development environment should match production as closely as possible. When you use one database locally and a different one in production, you're inviting subtle bugs. Queries that work fine in your local SQLite setup might behave differently in your production PostgreSQL environment. Data types don't map perfectly. You're essentially testing against a different system than you're deploying to.

By running PostgreSQL both locally and in a production environment such as Vercel, you eliminate this entire class of problems. Your local database is a real PostgreSQL instance. Your production database is a real PostgreSQL instance. The only difference is the connection string.

This approach also teaches you skills that transfer directly to professional work. PostgreSQL is the most popular database for production applications. Learning it now prepares you for real-world development environments.

Setting Up Local PostgreSQL

On macOS, Homebrew makes PostgreSQL installation simple. Install it and start the service:

```
# Install PostgreSQL
brew install postgresql@16

# Add to PATH (add this to your ~/.zshrc for persistence)
export PATH="/opt/homebrew/opt/postgresql@16/bin:$PATH"

# If you add the path to ~/.zshrc, run the following or open a new
terminal
source ~/.zshrc

# Start PostgreSQL as a background service
brew services start postgresql@16
```

This installs PostgreSQL 16 and runs it as a background service that starts automatically when you log in. Your data persists in Homebrew's data directory.

Once PostgreSQL is running, create a database for your application:

```
# Create a database (use your app's name)
createdb myapp
```

That's it. PostgreSQL on macOS uses *peer authentication* by default, meaning your macOS username connects without a password. This is perfect for local development.

On Windows, download the official installer from `PostgreSQL.org`. On Linux, your distribution's package manager provides PostgreSQL: `apt install postgresql` on Ubuntu/Debian; `dnf install postgresql-server` on Fedora.

To verify that your database is running, connect with `psql`:

```
# Connect to local PostgreSQL
psql myapp
```

If the connection is successful, you will see the `psql` prompt:

```
myapp=#
```

Database Schema Design

For agent applications with memory, you typically need tables for users, memories, and any domain-specific data. The following is a schema that covers the

common patterns. Paste the following into the `psql` prompt and then press Enter/Return:

```
-- Users table with preferences
CREATE TABLE users (
    id VARCHAR(50) PRIMARY KEY,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    preferences JSON DEFAULT '{}'
);
-- Memories table for agent context (organized by namespace)
CREATE TABLE memories (
    id VARCHAR(36) PRIMARY KEY,
    user_id VARCHAR(50) REFERENCES users(id) NOT NULL,
    namespace VARCHAR(100) NOT NULL,
    key VARCHAR(200) NOT NULL,
    value JSON NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX ix_memory_user_namespace ON memories(user_id, namespace);
-- Sessions table for conversation history
CREATE TABLE sessions (
    id VARCHAR(36) PRIMARY KEY,
    user_id VARCHAR(50) REFERENCES users(id) NOT NULL,
    started_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    ended_at TIMESTAMP,
    summary TEXT,
    metadata JSON DEFAULT '{}'
);
CREATE INDEX ix_session_user ON sessions(user_id);
-- Generated content cache (content-addressed)
CREATE TABLE content_cache (
    id VARCHAR(36) PRIMARY KEY,
    content_hash VARCHAR(64) UNIQUE NOT NULL,
    content_type VARCHAR(50) NOT NULL,
    content JSON NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    access_count INTEGER DEFAULT 0,
    last_accessed TIMESTAMP
);
CREATE INDEX ix_cache_hash ON content_cache(content_hash);
```

The `memories` table uses a namespace + key structure instead of a single content field. This lets you organize memories by category—preferences, facts, goals, and episodes—and look them up efficiently. The `value` column stores the actual memory data as JSON, giving you flexibility without schema migrations.

The `content_cache` table uses content-addressed storage: hash the inputs and use the hash as a lookup key. If the hash exists, return the cached content.

This pattern works for any expensive generation, including explanations, summaries, recommendations, or any LLM output you don't want to regenerate unnecessarily.

Connecting to PostgreSQL

The SQLAlchemy setup we defined at the start of this chapter used SQLite in-memory. To switch to PostgreSQL, you only need to change the engine creation and add connection pool settings. Everything else—the `Memory` model, `MemoryStore` class, and all memory tools—stays exactly the same:

```
POSTGRES_URL = os.environ.get("POSTGRES_URL")
if not POSTGRES_URL:
    raise RuntimeError(
        "POSTGRES_URL environment variable is required. "
        "Set it in your .env file, e.g.: POSTGRES_URL=postgresql://local
host/myapp"
    )
# Vercel uses 'postgres://' but SQLAlchemy requires 'postgresql://'
DATABASE_URL = POSTGRES_URL.replace("postgres://", "postgresql://", 1)
engine = create_engine(
    DATABASE_URL,
    pool_pre_ping=True,
    pool_size=5,
    max_overflow=10,
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=
engine)
```

For local development, add the following to your `.env` file:

```
POSTGRES_URL=postgresql://localhost/myapp
```

The `pool_pre_ping=True` setting verifies connections before using them, which is important for serverless environments where functions can be idle for extended periods. The URL translation from `postgres://` to `postgresql://` handles a Vercel quirk: they use the older URL scheme, but SQLAlchemy 2.0 requires the explicit `postgresql://` prefix.

Setting Up Vercel Postgres

For production, Vercel Postgres provides a managed PostgreSQL database with a generous free tier. Setup takes about a minute:

1. Go to your Vercel project dashboard.
2. Navigate to the Storage tab.
3. Click Create Database.

4. Select Neon: Serverless Postgres and click Continue.
5. Set up your database settings as needed (for example, by selecting a region close to you), and then click Continue.
6. Give your database a name (or leave the default as is) and then click Create. Vercel creates the database and automatically populates the `POSTGRES_URL` environment variable.
7. Redeploy your application.

That's it. Both environments use `POSTGRES_URL`. Locally it points to your development database, and on Vercel it contains the full connection string with credentials. Same code, same database engine, different instances.

Optimizing Performance Through Caching

Here's a reality check: LLM calls are expensive. Every token costs money, and every request takes time. So, if you're regenerating the same content repeatedly, you're burning money and making users wait for no good reason.

Caching solves this. Store the results of expensive operations and serve them instantly when the same request comes in again. This isn't just an optimization. For many applications, it's the difference between viable and unaffordable.

The Cost of Redundant Generation

Consider the background memory extraction we discussed earlier. Every time a conversation ends, you're calling an LLM to analyze the transcript and extract memories. But what if multiple users have similar conversations? What if your research assistant handles dozens of requests about the same topic? Without caching, you're paying for duplicate analysis every single time.

Let's say memory extraction costs $0.02 per conversation in LLM tokens. You have 10,000 users, and 30% of their conversations cover similar ground—common questions, standard onboarding flows, frequently discussed topics. Without caching, you're paying full price for every extraction. Those 3,000 redundant extractions at $0.02 each add up to $60 per cycle, and the costs scale linearly as your user base grows. With caching, you extract once and reuse the results when similar patterns appear. Those 3,000 extractions become database lookups costing essentially nothing.

Content-Addressed Caching Strategies

Content-addressed caching uses the input content itself as the cache key. Hash the input and use that hash to look up cached results. If the hash matches, you

have a cache hit, so return the stored result. If not, generate fresh content and store it under that hash.

```
def get_content_hash(content: str, params: dict) -> str:
    """Generate a deterministic hash for cache lookup."""
    cache_key_data = {
        "content": content,
        "params": params
    }
    key_string = json.dumps(cache_key_data, sort_keys=True)
    return hashlib.sha256(key_string.encode()).hexdigest()
```

The advantage of content-addressed caching is that identical inputs always produce the same hash. It doesn't matter which user requests it or when. If the input matches, the cached result applies.

Database-Backed Performance Wins

Your database doubles as a cache store. Store generated content with its content hash, then check the database before calling the LLM. Cache hits are simple `SELECT` queries, which are orders of magnitude faster than LLM generation.

The logic is straightforward: whenever the agent needs to generate content, the system first builds a deterministic fingerprint of the inputs (topic, source material, parameters like the user's expertise level) and looks up that fingerprint in the cache table. If the fingerprint is present, the stored result is returned without calling the LLM. If it is missing, the system generates the content, stores it under its fingerprint, and returns it. In effect, the database becomes a content-addressed store where identical inputs always map to the same key, so repeat requests cost only a single indexed row lookup instead of a full generation.

```
class ExplanationCache(Base):
    __tablename__ = "explanation_cache"
    id = Column(String(36), primary_key=True, default=lambda: str(uuid
.uuid4()))
    content_hash = Column(String(64), unique=True, index=True)
    topic = Column(String(200))
    user_level = Column(String(50))
    explanation = Column(String)
    access_count = Column(Integer, default=0)
    last_accessed = Column(DateTime)

ExplanationCache.__table__.create(engine, checkfirst=True)

def generate_explanation_with_llm(topic: str, context: str, user_level:
str) -> str:
    """Stub: in production, this calls an LLM to generate an
explanation."""
    return f"{topic} ({user_level} level): {context}"
```

To see how this works in practice, here is a typical content-cached generator. The function has three distinct phases: first, it builds the cache key from the request inputs, hashes the result, and queries the cache. Second, if a cached record exists, it updates the access counters and returns the stored explanation immediately. Third, if no record exists, it calls the LLM, stores the fresh result under the new hash, and returns it. The same inputs always follow the same code path, which means stability and predictability.

```
def get_or_generate_explanation(topic: str, context: str, user_level:
str) -> str:
    """Return cached explanation if available, otherwise generate new
one."""

    # Check cache first
    content_hash = get_content_hash(topic + context, {"level": user_
level})
    cached = db.query(ExplanationCache).filter_by(content_hash=content_
hash).first()

    if cached:
        cached.access_count += 1
        cached.last_accessed = datetime.now(UTC)
        db.commit()
        return cached.explanation

    # Cache miss - generate new explanation
    explanation = generate_explanation_with_llm(topic, context, user_level)

    # Store for future requests
    db.add(ExplanationCache(
        id=str(uuid.uuid4()),
        content_hash=content_hash,
        topic=topic,
        user_level=user_level,
        explanation=explanation
    ))
    db.commit()

    return explanation
```

This pattern applies to any LLM generation where the same inputs reliably produce the same outputs: explanations, summaries, memory extractions, analysis reports, and more.

Caching Generated Content

Let's go deeper into the mechanics of caching generated content. The pattern is straightforward, but the details matter for reliability and performance.

Hashing Inputs for Cache Keys

Your cache key needs to capture every stable input that affects the output. This means the content itself and the parameters that steer generation, such as the explanation style or target difficulty, but not mutable or nondeterministic values like timestamps or session IDs, because including those would invalidate the cache on every request even when the real workload has not changed.

```
def build_cache_key(topic: str, source_docs: list[str], params: dict)
-> str:
    """Build a comprehensive cache key."""

    # Sort everything for deterministic ordering
    key_components = {
        "topic": topic,
        "sources": sorted(source_docs),
        "params": dict(sorted(params.items()))
    }

    key_string = json.dumps(key_components, sort_keys=True)
    return hashlib.sha256(key_string.encode()).hexdigest()
```

A common mistake is including nondeterministic elements in your cache key, such as timestamps, random IDs, or mutable objects. These cause constant cache misses even when the meaningful input hasn't changed. Be deliberate about what goes into the hash.

Storing Generated Outputs

Once you've generated content, store it with enough metadata to be useful later: the content itself, obviously, but also when it was created, what inputs produced it, and maybe statistics like token usage.

```
class GeneratedContent(Base):
    __tablename__ = "generated_content"

    id = Column(String, primary_key=True)
    content_hash = Column(String, unique=True, index=True)
    content_type = Column(String)  # "explanation", "summary",
"extraction", etc.
    content = Column(JSON)
    input_summary = Column(String)  # For debugging
    created_at = Column(DateTime, default=datetime.utcnow)
    access_count = Column(Integer, default=0)
    last_accessed = Column(DateTime)
```

The access tracking (count and last accessed time) helps with cache management. You can identify popular content worth keeping and stale content worth evicting.

Even with generous storage limits, tracking usage patterns helps optimize your cache strategy.

Measuring Performance Improvements

How do you know caching is actually helping? Instrument your cache with metrics. Track hits versus misses. Measure latency with and without cache. To calculate cost savings, multiply the number of cache hits by the estimated cost per generation: the sum of LLM token fees or inferred compute cost for the operation you intercepted. For example, if a memory extraction costs $0.02 and your cache recorded 3k hits, the avoided cost is $60. This figure gives you a direct monetary argument for maintaining the cache layer.

```
class CacheMetrics:
    def __init__(self):
        self.hits = 0
        self.misses = 0
        self.total_latency_saved_ms = 0
        self.estimated_cost_saved = 0.0

    def record_hit(self, latency_saved_ms: int, cost_saved: float):
        self.hits += 1
        self.total_latency_saved_ms += latency_saved_ms
        self.estimated_cost_saved += cost_saved

    def record_miss(self):
        self.misses += 1

    @property
    def hit_rate(self) -> float:
        total = self.hits + self.misses
        return self.hits / total if total > 0 else 0.0
```

Good caching can yield dramatic improvements. Hit rates of 60–80% are common for applications with repeat queries. At those rates, you're cutting LLM costs by more than half while serving most responses instantly.

When to Cache

Not everything benefits from caching. Cache when the operation is expensive (LLM calls), when inputs repeat (common topics, standard questions), and when outputs are deterministic or semi-deterministic (same input produces same or similar output).

Don't cache when freshness matters (current events, time-sensitive data), when personalization makes outputs unique (heavily user-specific responses), or when storage costs exceed generation costs (tiny operations, huge outputs).

The sweet spot for caching is expensive operations with moderate output size and repeatable inputs. Concept explanations hit all three: LLM calls are expensive, explanations are moderate in size, and many users ask about the same topics. Memory extraction is another good candidate because conversation patterns repeat and the extracted insights can be reused when similar conversations occur.

Building StudyBuddy v6

All right, here's where it all comes together. We've covered the theory of memory systems, explored formation strategies, and discussed caching for performance. Now let's put it into practice by adding memory and persistence to StudyBuddy.

Where We Left Off

In Chapter 5, we built a multi-agent system with a Tutor Agent, Card Generator Agent, Quality Checker Agent, and a Learning Coordinator (supervisor) that orchestrates them. The system can explain concepts and generate flashcards, but it has a significant limitation: it forgets everything the moment you restart the server. Ask it what you studied yesterday? It has no idea. Start a new session? It starts with a complete blank slate.

Version 6 fixes that. We're adding persistent memory that survives across sessions, flashcard caching that avoids redundant generation, and a new Scheduler Agent that implements spaced repetition. When we're done, StudyBuddy will remember what you've studied, how well you've performed, and when you should review each topic next.

What We're Adding

This version introduces several new capabilities that work together to create a personalized learning experience:

- **Persistence with PostgreSQL:** All data—memories, flashcards, learning history—persists in a real database. Restart the server, come back tomorrow, and everything is still there.
- **Content-addressed flashcard caching:** Generated flashcards are cached by content hash. The same material produces instant results with no regeneration needed.
- **Scheduler Agent:** A new agent that tracks review performance and implements the SM-2 spaced repetition algorithm to optimize learning.
- **Learning history tracking:** A complete record of what concepts you've studied, how you performed, and what needs attention.

Persistence Strategy

We're using PostgreSQL for all persistent storage, with the same database engine in development and production. Locally, you'll run PostgreSQL via Homebrew. On Vercel, you'll use their managed PostgreSQL service. Your code doesn't change between environments; only the connection string differs.

This approach eliminates the "works on my machine" problem. When something works locally, it works in production. When you debug an issue locally, you're debugging against the same database engine you're running in production. No surprises.

Your database contains everything: user profiles, memories, flashcard cache, and learning history. Locally, it's your Homebrew PostgreSQL instance. In production, it's your Vercel Postgres database. The application code is identical.

Database Configuration

Our database setup requires the `POSTGRES_URL` environment variable, failing fast if it's missing rather than silently using a wrong database:

```
# api/database/connection.py
import os
from contextlib import contextmanager
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

from .models import Base, User

# PostgreSQL connection (required)
POSTGRES_URL = os.environ.get("POSTGRES_URL")

if not POSTGRES_URL:
    raise RuntimeError(
        "POSTGRES_URL environment variable is required. "
        "Set it in your .env file, e.g.: POSTGRES_URL=postgresql://
localhost/studybuddy"
    )

# Fix URL scheme for SQLAlchemy 2.0 (Vercel uses postgres://)
DATABASE_URL = POSTGRES_URL.replace("postgres://", "postgresql://", 1)
engine = create_engine(
    DATABASE_URL,
    pool_pre_ping=True,  # Verify connections before using
    pool_size=5,
    max_overflow=10,
)
print(f"Using PostgreSQL: {DATABASE_URL.split('@')[-1] if '@' in
DATABASE_URL else 'localhost'}")
```

```
# Session factory
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=
engine)

def init_database():
    """Create all tables if they don't exist and ensure default user
exists."""
    # Create tables
    Base.metadata.create_all(bind=engine)
    print("Database initialized")

    # Ensure default user exists
    db = SessionLocal()
    try:
        default_user = db.query(User).filter_by(id="default").first()
        if not default_user:
            db.add(User(id="default", preferences={}))
            db.commit()
            print("Created default user")
    finally:
        db.close()
```

Custom Memory Store

For user-specific memories about learning preferences and progress, we use a custom SQLAlchemy-based memory store. This gives us full control without external dependencies:

```
# api/services/memory_store.py
import uuid
from datetime import datetime
from sqlalchemy.orm import Session

from ..database.models import Memory


class MemoryStore:
    """Simple memory store backed by SQLAlchemy.

    Usage:
        store = MemoryStore(db_session)
        store.put("default", "preferences", "learning_style", {"type":
"visual"})
        memories = store.search("default", "preferences")
    """

    def __init__(self, db: Session):
        """Initialize with database session.
```

```
    Args:
        db: SQLAlchemy database session
    """
    self.db = db

def put(
    self,
    user_id: str,
    namespace: str,
    key: str,
    value: dict,
) -> Memory:
    """Store or update a memory.

    Args:
        user_id: User identifier
        namespace: Category (e.g., "preferences", "struggles")
        key: Unique key within namespace
        value: Dictionary of data to store

    Returns:
        The created or updated Memory object
    """
    # Check for existing memory
    existing = (
        self.db.query(Memory)
        .filter_by(user_id=user_id, namespace=namespace, key=key)
        .first()
    )

    if existing:
        existing.value = value
        existing.updated_at = datetime.utcnow()
        self.db.commit()
        return existing
    else:
        memory = Memory(
            id=str(uuid.uuid4()),
            user_id=user_id,
            namespace=namespace,
            key=key,
            value=value,
        )
        self.db.add(memory)
        self.db.commit()
        return memory

def get(
    self,
    user_id: str,
```

```
            namespace: str,
            key: str,
        ) -> dict | None:
            """Get a specific memory by key.

            Args:
                user_id: User identifier
                namespace: Category
                key: Memory key

            Returns:
                The memory value dict, or None if not found
            """
            memory = (
                self.db.query(Memory)
                .filter_by(user_id=user_id, namespace=namespace, key=key)
                .first()
            )
            return memory.value if memory else None

        def search(
            self,
            user_id: str,
            namespace: str | None = None,
            limit: int = 10,
        ) -> list[dict]:
            """Search memories in a namespace.

            Args:
                user_id: User identifier
                namespace: Optional namespace to filter by
                limit: Maximum results to return

            Returns:
                List of memory dicts with key and value
            """
            query = self.db.query(Memory).filter_by(user_id=user_id)

            if namespace:
                query = query.filter_by(namespace=namespace)

            query = query.order_by(Memory.updated_at.desc()).limit(limit)

            return [
                {
                    "namespace": m.namespace,
                    "key": m.key,
                    "value": m.value,
                    "updated_at": m.updated_at.isoformat() if m.updated_at
else None,
                }
```

```
            for m in query.all()
        ]

    def delete(
        self,
        user_id: str,
        namespace: str,
        key: str,
    ) -> bool:
        """Delete a specific memory.

        Args:
            user_id: User identifier
            namespace: Category
            key: Memory key

        Returns:
            True if deleted, False if not found
        """
        memory = (
            self.db.query(Memory)
            .filter_by(user_id=user_id, namespace=namespace, key=key)
            .first()
        )

        if memory:
            self.db.delete(memory)
            self.db.commit()
            return True
        return False
```

The namespace conventions organize memories into categories:

- `preferences`: Learning style, favorite topics, preferred explanation depth
- `struggles`: Topics the user finds difficult
- `goals`: What the user is studying for (certification, job, etc.)
- `sessions`: Summaries of past study sessions

When the Tutor Agent learns something important about a user, it stores that information. In subsequent sessions, whether running locally or in production, it retrieves relevant context to personalize the experience.

Content-Addressed Flashcard Caching

Here's where we get serious performance wins. Instead of regenerating flashcards every time someone studies a topic, we cache generated cards by content hash. Same topic plus same source material equals same hash, leading to instant retrieval instead of expensive LLM generation.

Let's start with the imports:

```
# api/services/flashcard_cache.py
import hashlib
import uuid
import logging
from datetime import datetime
from sqlalchemy.orm import Session

from ..database.models import Flashcard

logger = logging.getLogger(__name__)
```

The first building block is generating a deterministic hash from the topic and source material. We normalize by lowercasing and stripping whitespace so "Machine Learning" and "machine learning" produce the same hash:

```
def compute_content_hash(topic: str, source_context: str) -> str:
    """Generate SHA-256 hash of topic + source context."""

    # Normalize inputs for consistent hashing
    content = f"{topic.strip().lower()}|{source_context.strip()}"
    return hashlib.sha256(content.encode()).hexdigest()
```

The lookup function checks whether we already have flashcards for this exact content. If so, it's a cache hit, and we can skip the expensive LLM call:

```
def get_cached_flashcards(
    topic: str,
    source_context: str,
    db: Session
) -> list[Flashcard] | None:
    """Check cache for existing flashcards."""
    content_hash = compute_content_hash(topic, source_context)

    cards = db.query(Flashcard).filter_by(content_hash=content_hash)
.all()

    if cards:
        logger.info(f"Cache HIT for topic '{topic}' - {len(cards)}
cards")
        return cards

    logger.info(f"Cache MISS for topic '{topic}'")
    return None
```

When we have a cache miss, we need to store the newly generated flashcards. Each card gets its own UUID and links back to the content hash so we can find it again:

```
def cache_flashcards(
    topic: str,
    source_context: str,
    cards_data: list[dict],
    db: Session,
    chapter_id: int | None = None,
) -> list[Flashcard]:
    """Store generated flashcards in cache.

    Args:
        topic: The topic of the cards
        source_context: The RAG context used
        cards_data: List of card dictionaries with question/answer
        db: Database session
        chapter_id: Optional chapter identifier

    Returns:
        List of created Flashcard objects
    """
    content_hash = compute_content_hash(topic, source_context)

    flashcards = []
    for card in cards_data:
        flashcard = Flashcard(
            id=str(uuid.uuid4()),
            content_hash=content_hash,
            topic=topic,
            question=card.get("question", ""),
            answer=card.get("answer", ""),
            difficulty=card.get("difficulty", "intermediate"),
            chapter_id=chapter_id,
            quality_score=card.get("quality_score", 3),
            was_revised=card.get("was_revised", False),
            source_context_hash=hashlib.sha256(
                source_context[:500].encode()
            ).hexdigest(),
        )
        db.add(flashcard)
        flashcards.append(flashcard)

    db.commit()
    logger.info(f"Cached {len(flashcards)} cards for topic '{topic}'")

    return flashcards
```

The main entry point handles the full flow: checking the cache, generating if needed, and storing the results. Callers get back a tuple of `(flashcards, was_cache_hit)`:

```
def get_or_generate_flashcards(
    topic: str,
    source_context: str,
    generator_func,
    db: Session,
    chapter_id: int | None = None,
) -> tuple[list[dict], bool]:
    """Generate flashcards with caching - main entry point.

    Checks cache first, generates only on miss.

    Args:
        topic: Topic to generate cards for
        source_context: RAG context
        generator_func: Function that generates cards (LLM call)
        db: Database session
        chapter_id: Optional chapter ID

    Returns:
        Tuple of (list of flashcard dicts, cache_hit boolean)
    """
    # Check cache first
    cached = get_cached_flashcards(topic, source_context, db)
    if cached:
        # Convert Flashcard objects to dicts
        return (
            [
                {
                    "id": c.id,
                    "question": c.question,
                    "answer": c.answer,
                    "topic": c.topic,
                    "difficulty": c.difficulty,
                }
                for c in cached
            ],
            True,  # cache_hit
        )

    # Cache miss - generate with LLM
    cards_data = generator_func(topic, source_context)

    # Store in cache
    if cards_data:
        cached_cards = cache_flashcards(
```

```
            topic=topic,
            source_context=source_context,
            cards_data=cards_data,
            db=db,
            chapter_id=chapter_id,
        )
        # Return with IDs from cached objects
        return (
            [
                {
                    "id": c.id,
                    "question": c.question,
                    "answer": c.answer,
                    "topic": c.topic,
                    "difficulty": c.difficulty,
                }
                for c in cached_cards
            ],
            False,  # cache_hit
        )

    return (cards_data, False)
```

The first user to study a topic pays the generation cost. Every subsequent user gets instant results. For a curriculum with well-defined topics, cache hit rates can exceed 80%, resulting in a massive savings in both cost and latency.

Running Locally

StudyBuddy v6 runs with three terminals.

Terminal 1 runs the backend:

```
cd v6-agent-memory
uv sync
uv run uvicorn api.index:app --reload --port 8000
```

Terminal 2 runs the frontend:

```
cd v6-agent-memory/frontend
npm install
npm run dev
```

Terminal 3 runs the database:

```
# Check PostgreSQL service status
brew services list | grep postgresql

# If not running, start it
brew services start postgresql@16
```

Visit `http://localhost:3000`. The Next.js dev server proxies `/api/*` requests to the FastAPI backend on port 8000 (configured in `next.config.ts`).

Deploying to Vercel

To deploy v6 to Vercel, you need to update two files in the repo root. First, update `vercel.json` to point to the `v6-agent-memory` directory:

```
{
    "version": 2,
    "builds": [
        {
            "src": "v6-agent-memory/api/index.py",
            "use": "@vercel/python"
        },
        {
            "src": "v6-agent-memory/frontend/package.json",
            "use": "@vercel/next"
        }
    ],
    "routes": [
        { "src": "/api/(.*)", "dest": "/v6-agent-memory/api/index.py" },
        { "src": "/(.*)", "dest": "/v6-agent-memory/frontend/$1" }
    ],
    "git": {
        "deploymentEnabled": {
            "main": true
        }
    },
    "ignoreCommand": "bash scripts/should-deploy.sh"
}
```

Second, update `scripts/should-deploy.sh` to trigger on v6 (and `vercel.json`) changes:

```
#!/bin/bash
# Deploy if changes are in v6-agent-memory or vercel config
# Exit 1 (true) = proceed with build, Exit 0 (false) = skip build
git diff HEAD^ HEAD --name-only | grep -qE "^v6-agent-memory/|^vercel\
.json" && exit 1 || exit 0
```

If you haven't done so already, set your OpenAI API key environment variable in Vercel by running the following command from the project root and pasting the key when prompted:

```
vercel env add OPENAI_API_KEY
```

Be sure to go through the steps we outlined earlier in the "Setting Up Vercel Postgres" section to get your Vercel database running and the appropriate environment variables set. Then deploy:

```
vercel --prod
```

Or simply push to GitHub. Vercel will automatically deploy when changes are detected in the `v6-agent-memory` directory.

Testing Persistence

With PostgreSQL running locally, testing persistence is straightforward. Start the server, have a tutoring session, generate some flashcards, review a few cards. Then stop the server completely and restart it. Everything should still be there.

Here's your testing checklist:

1. Check that memories survive a restart: the memories table should retain data across server restarts.
2. Verify flashcard cache hits: studying the same topic should return instant results.
3. Confirm learning history: the stats endpoint should show previous review performance.
4. Test spaced repetition: cards reviewed yesterday shouldn't be due again yet.

You can inspect your local database directly using `psql` or a GUI tool like pgAdmin, DBeaver, or TablePlus:

```
# Connect and explore
psql studybuddy

# List tables
\dt
# Check flashcard cache
SELECT topic, COUNT(*) FROM flashcards GROUP BY topic;
```

For production testing on Vercel, the process is identical: deploy, test the same checklist, and verify data persists. Vercel's dashboard includes a Data Browser for exploring your production tables, or you can connect with any PostgreSQL client using the credentials from your project settings.

If you need to, reset your local database and start fresh:

```
# Drop and recreate the database
dropdb studybuddy
createdb studybuddy
```

This clean-slate approach takes seconds. When your local data gets messy from testing, just drop and re-create.

What's Next

StudyBuddy v6 is a major upgrade. We've added persistence that survives restarts, intelligent caching that cuts costs, and spaced repetition that optimizes learning. The system now genuinely learns about users and adapts to their needs.

In Chapter 7, we'll go even deeper with deep agents that plan over long time horizons, spawn subagents for complex tasks, and manage context across extended operations. The memory foundation we've built here will be essential for agents that need to maintain state across days or weeks of work.

Let's keep building.

CHAPTER

7

Deep Agents

The agents we've built so far are reactive. A user asks a question, the agent thinks for a moment, maybe calls a tool or two, and responds. This approach is useful and handles a wide variety of tasks. But some problems don't fit neatly into a single conversation turn.

Think about planning a two-week trip through Europe. You need to research destinations, check flight availability, compare hotel prices, build an itinerary that makes geographic sense, book everything in the right sequence, and handle the inevitable conflicts when your preferred hotel is full on the dates you need. A simple agent might help you search for flights. A *deep* agent could orchestrate the entire planning process over multiple sessions, remembering your preferences, adjusting plans when constraints change, and coordinating all the moving pieces.

That's what this chapter is about: building agents that can handle complex, multistep projects that unfold over time. We're moving from agents that respond to agents that *plan*, *delegate*, *reflect*, and *adapt*. This is where AI engineering gets seriously interesting.

What Makes an Agent "Deep"?

The term "deep agent" comes from recent work in the AI engineering community, particularly from teams at Anthropic and LangChain who've been pushing the

boundaries of what agents can accomplish. But what actually makes an agent "deep" versus "shallow" (see Figure 7.1)?

A *shallow* agent operates in what we might call stimulus-response mode. The user provides input, and the agent generates output. The agent might use tools and reason through problems, but each interaction is essentially self-contained. The ReAct pattern we covered in Chapter 3 is a perfect example. The agent thinks, acts, observes, and repeats until it has an answer. Powerful, but bounded.

A *deep agent* breaks out of this single-turn paradigm. It maintains goals across multiple interactions. It decomposes big objectives into smaller, manageable subtasks. It spawns helper agents to handle specific pieces of work. It reflects on its own progress and adjusts strategies when things aren't working. And, crucially, the deep agent manages context intelligently by knowing what to remember, what to summarize, and what to let go. Figure 7.1 captures this contrast visually: on the left, the shallow agent's linear input-process-output loop; on the right, the deep agent as a hub connected to its four core capabilities across multiple interactions, also known as *sessions*. The bidirectional arrows between the Deep Agent hub and each capability node—Planning, Context Management, Delegation, and Reflection—indicate that the agent both drives and is updated by each capability. The Session nodes represent distinct interactions over time; unlike a shallow agent, the deep agent maintains continuity and pursues its goals across all of them.

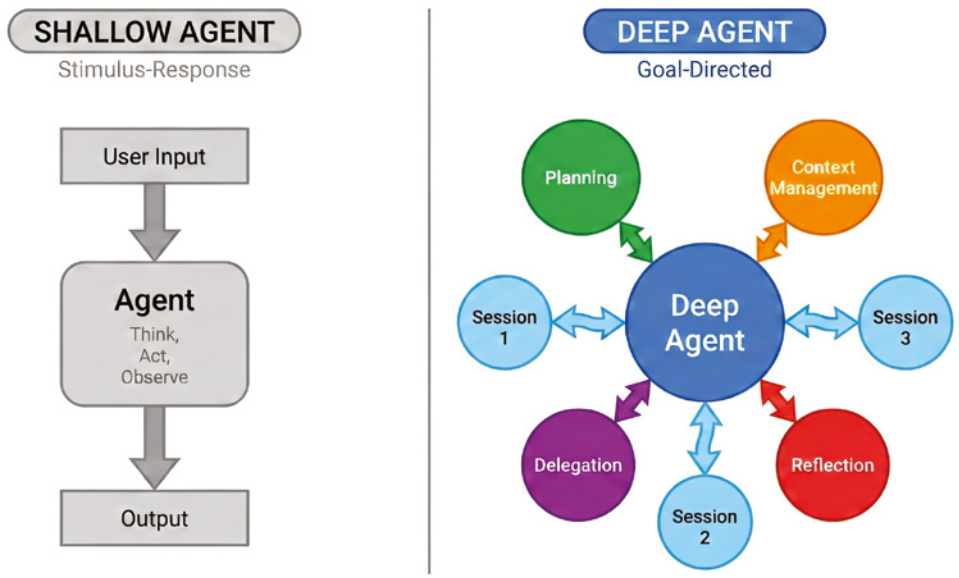

Figure 7.1: A shallow agent (left) follows a simple stimulus-response loop: the user input flows in, the agent thinks and acts, and output flows out. A deep agent (right) is goal-directed rather than reactive.

Here's a concrete way to think about the distinction. Imagine you're building a research assistant. A shallow agent helps you search for papers and summarize what it finds. You ask, and it delivers. A deep agent, on the other hand, could take a research question such as "What are the emerging best practices for fine-tuning large language models?" and autonomously create a research plan, identify the key sub-questions to investigate, search multiple sources for each sub-question, synthesize findings across sources, identify gaps and contradictions, and produce a comprehensive report, all while keeping you informed of progress and asking for guidance only when genuinely needed.

So, when do you actually need deep agent capabilities? Here's a quick decision framework:

Consider deep agents when:

- The task naturally spans multiple sessions or extended time periods.
- Success requires coordinating multiple distinct subtasks.
- The problem benefits from planning before acting.
- You need the agent to recover from dead ends autonomously.
- Context management becomes a bottleneck with simpler approaches.

Stick with simpler agents when:

- Tasks complete in a single conversation turn.
- The workflow is predictable and can be hard-coded.
- Latency is critical (deep agents add overhead).
- You're still figuring out the problem space.

Deep agents aren't always better. They're more complex to build, harder to debug, and slower to execute. But for the right problems, they're absolute game-changers.

The Four Key Elements of Deep Agents

The deep agents framework rests on four interconnected capabilities, as shown in Figure 7.2. Each element addresses a specific challenge that emerges when agents tackle complex, long-running tasks:

- **Planning and task decomposition:** Breaking big goals into achievable subtasks with clear dependencies. Without this, agents struggle. They might work hard but not make meaningful progress toward the actual objective.

- **Context management:** Strategically handling information over long time horizons. Context windows are finite, and deep agents need to decide what's worth keeping, what to summarize, and what to forget.
- **Subagent spawning and delegation:** Creating specialized helper agents for specific subtasks. Just like a manager delegates to team members, a deep agent orchestrates a workforce of focused specialists.
- **Reflection and backtracking:** Evaluating whether the agent is making progress, critiquing its own outputs, and adjusting strategies when something isn't working.

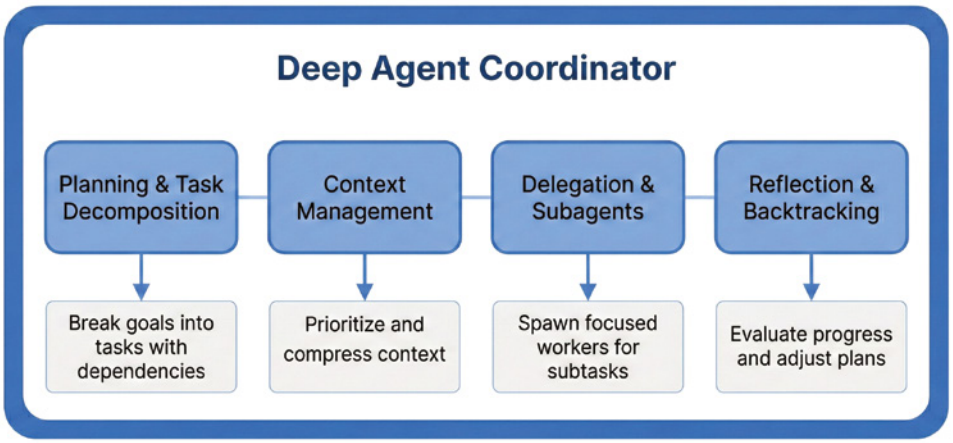

Figure 7.2: The four interconnected capabilities that every deep agent coordinator must implement: planning, context management, delegation, and reflection.

These four elements work together as a system. Planning generates the subtasks that get delegated to subagents. Context management ensures the agent doesn't lose track of its goals or progress. Spawning brings specialists in to handle specialized tasks. Reflection ensures that the deep agent stays focused on its objectives. When all four work together, you get agents that feel genuinely intelligent. Not just responsive, but purposeful.

Let's dig into each element.

Planning and Task Decomposition

Every complex project starts with a plan. This is true for humans, and it's equally true for deep agents. Without explicit planning, agents tend to tackle whatever seems most immediate, which often isn't what's most important. Planning gives agents direction.

The core idea is straightforward: take a high-level goal and break it down into a sequence of smaller, achievable tasks. But the execution has real nuance. How do you decide what tasks are needed? How do you handle dependencies between tasks? How detailed should each task be?

Breaking Down Complex Problems

Let's say you're building an agent to help organize a company offsite. The high-level goal is "Plan and execute a successful team offsite for 30 people." That's too vague for an agent to act on directly. We need decomposition.

A good decomposition represents the overall goal and breaks it into discrete, individual tasks. Each task carries a unique ID, a plain-language description, a list of the task IDs it depends on, and an estimated duration. The dependency list is crucial because it makes the structure of the problem explicit. For example, it would prevent the agent from trying to book a venue before it knows the budget. The following code defines this structure using Pydantic models:

```
from pydantic import BaseModel

class Task(BaseModel):
    id: str
    description: str
    depends_on: list[str] = []  # Task IDs this depends on
    estimated_hours: float
    status: str = "pending"

class Plan(BaseModel):
    goal: str
    tasks: list[Task]

offsite_plan = Plan(
    goal="Plan team offsite for 30 people",
    tasks=[
        Task(id="t1", description="Determine budget and dates", depends_
on=[], estimated_hours=1),
        Task(id="t2", description="Survey team for location preferences",
depends_on=["t1"], estimated_hours=0.5),
        Task(id="t3", description="Research venue options matching budget",
depends_on=["t1", "t2"], estimated_hours=3),
        Task(id="t4", description="Get quotes from top 3 venues", depends_
on=["t3"], estimated_hours=2),
        Task(id="t5", description="Book selected venue", depends_on=["t4"],
estimated_hours=0.5),
        Task(id="t6", description="Plan agenda and activities", depends_
on=["t5"], estimated_hours=4),
        Task(id="t7", description="Arrange transportation and logistics",
depends_on=["t5"], estimated_hours=2),
```

```
        Task(id="t8", description="Send invitations with details", depends_
on=["t6", "t7"], estimated_hours=1),
    ]
)
```

NOTE **You can find all of this chapter's non-StudyBuddy code in the Jupyter notebook `chapter07.ipynb`, available in this book's GitHub repo.**

Notice what's happening here. Each task is specific enough to act on. Dependencies are explicit so you can't, for example, book a venue before you know the budget. And the decomposition reveals the natural structure of the problem.

Forward Planning vs. Backward Chaining

There are two main strategies for generating plans: forward planning and backward chaining.

Forward planning starts from the current state and asks, "What's the first thing I need to do? Then what? Then what?" It's intuitive and works well when the path forward is reasonably clear.

Backward chaining starts from the goal and works backward: "To achieve X, what needs to be true? To make that true, what needs to happen first?" This approach excels when you have a clear end state but the path to get there is murky.

For learning-related goals, backward chaining often works well. Say someone wants to pass the AWS Solutions Architect certification exam. Working backward:

- To pass the exam, I need to understand all domains covered.
- To understand those domains, I need to study each one systematically.
- To study effectively, I need quality learning materials and practice questions.
- To get practice questions, I need to either find them or generate them.

Each step backward reveals prerequisites, eventually reaching tasks the agent can execute immediately.

The following simplified backward chaining implementation shows the core technique. The `backward_chain` function takes a goal, asks the LLM what must be true immediately before that goal can be achieved, then recursively applies the same question to each prerequisite. The `max_depth` parameter prevents infinite recursion by capping how many levels deep the function will go. The result is a flat list of tasks ordered from earliest prerequisite to the final goal:

```
def backward_chain(goal: str, llm, max_depth: int = 3, depth: int = 0)
-> list[str]:
    """Generate prerequisites by working backward from goal."""
```

```
    if depth >= max_depth:
        return [goal]

    prompt = f"""Given this goal: {goal}

What must be true or completed IMMEDIATELY BEFORE this goal can be
achieved?
List only the direct prerequisites, not earlier steps.
Be specific and actionable.

Format: One prerequisite per line, no numbering."""

    prerequisites = llm.invoke(prompt).content.strip().split("\n")

    all_tasks = []
    for prereq in prerequisites:
        prereq = prereq.strip()
        if prereq:
            # Recursively find prerequisites of prerequisites
            sub_prereqs = backward_chain(prereq, llm, max_depth=max_
depth, depth=depth + 1)
            all_tasks.extend(sub_prereqs)

    all_tasks.append(goal)
    return all_tasks
```

In practice, you'd add depth limits to prevent infinite recursion and caching to avoid recomputing the same prerequisites. But the core pattern—start at the goal, work backward—is incredibly powerful for educational and project planning agents.

Context Management in Deep Agents

Here's a problem that doesn't exist for simple agents but becomes critical for deep ones: context window exhaustion.

When an agent operates over extended periods, information accumulates. Conversation history, tool outputs, intermediate results, plans, progress updates—it all adds up. Eventually, you hit the context window limit. But even before that, performance degrades. Models get confused when context is cluttered with irrelevant details. The important stuff gets lost in the noise.

Deep agents need explicit strategies for managing context over time, as shown in Figure 7.3.

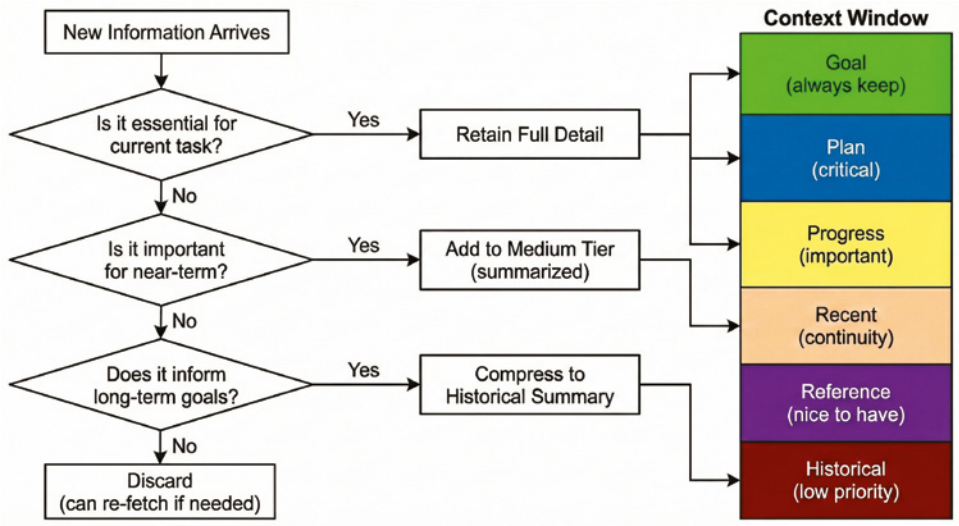

Figure 7.3: How a deep agent decides what to keep, summarize, or discard as new information arrives, mapping each decision to a priority tier in the context window.

Selective Information Retention

Not all information is equally important. A deep agent working on a research project probably needs to retain:

- The original research question (always relevant)
- Current plan and progress status (where the project stands)
- Key findings discovered so far (the actual results)
- Recent conversation context (immediate continuity)

But it probably doesn't need:

- The full text of every document searched (it can re-fetch if needed)
- Detailed reasoning traces from completed tasks (the conclusion matters, not the journey)
- Failed attempts that led nowhere (unless they inform current strategy)

The following `prioritize_context` function implements a simple priority scoring system. Each context has a `type` field that maps to a base score: the original goal scores the highest value of 10 (which means this is always kept), the current plan scores 9, progress updates score 8, and so on down to historical interactions, which score 1. Items that are directly relevant to the current task get a boost of 1.5 times on top of their base score. The function sorts all items by final score in descending order and returns them ranked so that the

coordinator can fill the context window from the top of the list and stop when it's full, thereby naturally keeping the most relevant information:

```
def prioritize_context(context_items: list[dict], current_task: str) ->
list[dict]:
    """Rank context items by relevance to current task."""

    categories = {
        "goal": 10,          # Original objective - always keep
        "plan": 9,           # Current plan - critical
        "progress": 8,       # What's done/pending - important
        "findings": 7,       # Key results - valuable
        "recent": 6,         # Last few messages - continuity
        "reference": 3,      # Supporting details - nice to have
        "historical": 1,     # Old interactions - low priority
    }

    scored = []
    for item in context_items:
        base_score = categories.get(item.get("type"), 2)

        # Boost if relevant to current task
        if is_relevant(item, current_task):
            base_score *= 1.5

        scored.append((base_score, item))

    # Sort by score descending, return items
    scored.sort(key=lambda x: x[0], reverse=True)
    return [item for _, item in scored]
```

NOTE **To save space, we're not showing helper functions such as the `is_relevant()` function in the preceding code. See this chapter's Jupyter notebook for the full code.**

Context Compression and Summarization

When you can't keep everything, summarize. The trick is summarizing at the right level of detail: enough to be useful, concise enough to save space.

```
async def compress_context(full_context: str, max_tokens: int, llm) ->
str:
    """Compress context while preserving essential information."""

    prompt = f"""Summarize the following context, preserving:
1. The main goal or objective
```

```
2. Key decisions made and their rationale
3. Important findings or results
4. Current status and next steps
5. Any critical constraints or requirements

Remove:
- Redundant information
- Detailed intermediate reasoning
- Verbose explanations where a summary suffices

Context to compress:
{full_context}

Provide a compressed summary in under {max_tokens} tokens:"""

    summary = await llm.ainvoke(prompt)
    return summary.content
```

For long-running agents, periodic compression keeps context manageable. You might compress after completing each major task, or when context size exceeds a threshold, or at natural breakpoints in the workflow.

When to Summarize vs. Retain Details

This is a judgment call, but here are some guidelines.

Retain full details when:

- Information is needed for the immediate next steps.
- Exact wording matters (legal text, specific quotes).
- You're in active problem-solving mode.
- The information hasn't been acted on yet.

Summarize when:

- A task is complete and you only need the conclusion.
- Information is supporting context, not primary content.
- You're transitioning between major phases of work.
- Multiple items cover similar ground and can be consolidated.

The best deep agents develop "intuitions" about this through reflection: noticing when they've summarized too aggressively and lost needed detail or kept too much and gotten confused.

Subagent Spawning and Delegation

A single agent trying to do everything is like a one-person company trying to scale. At some point, you need to delegate.

Subagent spawning is the pattern in which a coordinating agent creates specialized helper agents to handle specific subtasks. The coordinator is the top-level agent. It's instantiated when the task begins, persists for the full duration of the work, and is responsible for the overall goal. Subagents, by contrast, are created on demand by the coordinator, scoped to a single focused task, and terminated when that task is done. The coordinator never becomes a subagent itself. Each subagent has a focused role, the tools relevant to its job, and only the context it needs to complete its assignment. The coordinator orchestrates the overall workflow, delegates tasks, collects results, and synthesizes the final output.

Figure 7.4 shows an example. The Coordinator Agent receives the complex task, performs task analysis, and spawns three specialized subagents—Research, Writing, and Analysis—that run in parallel inside the Parallel Execution Zone. Each subagent follows its own initialize-execute-report lifecycle before returning results to the coordinator, which then synthesizes the final output. Real delegation scenarios will call for different subagents depending on the task; we use the research scenario here to make the pattern concrete.

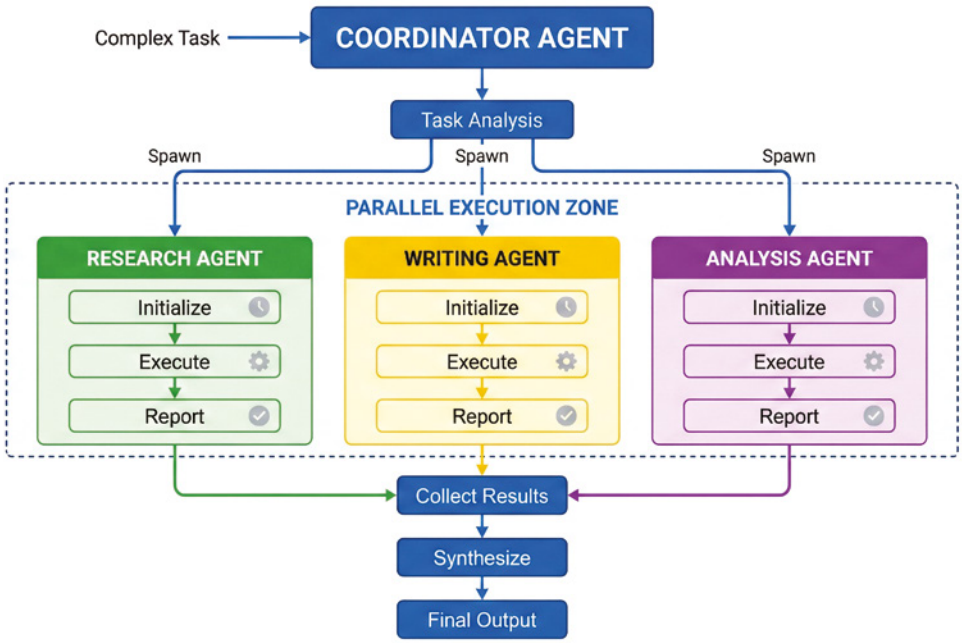

Figure 7.4: A delegation example using a research scenario.

Not every task needs delegation. Here's when subagents make sense:

- Tasks requiring specialized tools or knowledge
- Work that can proceed independently of other tasks
- Subtasks with clearly defined inputs and outputs
- Tasks where context isolation helps (the full project context isn't needed)

And here's when delegation is not necessary:

- Tasks requiring holistic project understanding
- Decision points that affect overall strategy
- Synthesis and integration work
- Tasks where context-switching overhead exceeds the benefit

Managing Subagent Lifecycles

Spawning a subagent is just the beginning. You also need to:

1. *Initialize* with appropriate context and instructions.
2. *Monitor* progress and handle failures.
3. *Collect* results when the task is complete.
4. *Terminate* cleanly, releasing resources.

Before we dive into delegation strategies, we need infrastructure to track subagents. Think of `SubagentManager` as a project manager that spawns workers, checks on their progress, and collects results when they're done.

We start with basic data structures, a status `enum` tracking lifecycle stages (pending → running → completed/failed) and a dataclass holding each subagent's identity, role, task, and state:

```
from dataclasses import dataclass
from typing import Any
from enum import Enum

class SubagentStatus(Enum):
    PENDING = "pending"
    RUNNING = "running"
    COMPLETED = "completed"
    FAILED = "failed"

@dataclass
class Subagent:
    id: str
    role: str
```

```
    task: str
    status: SubagentStatus
    result: Any = None
    error: str = None
```

The `SubagentManager` class orchestrates everything. When you call `spawn()`, it creates a new subagent, stores it in the registry, and kicks off execution:

```
class SubagentManager:
    def __init__(self):
        self.subagents: dict[str, Subagent] = {}

    def spawn(self, role: str, task: str, context: dict) -> str:
        """Create and start a new subagent."""
        agent_id = f"{role}_{len(self.subagents)}"

        subagent = Subagent(
            id=agent_id,
            role=role,
            task=task,
            status=SubagentStatus.PENDING
        )
        self.subagents[agent_id] = subagent

        # In practice, you'd actually create and run the agent here
        self._execute_subagent(subagent, context)

        return agent_id
```

Once tasks are running, the coordinator needs to check on them. The `get_status()` method returns current state, while `get_result()` retrieves outputs–but only for completed agents:

```
    def get_status(self, agent_id: str) -> SubagentStatus:
        return self.subagents[agent_id].status

    def get_result(self, agent_id: str) -> Any:
        subagent = self.subagents[agent_id]
        if subagent.status != SubagentStatus.COMPLETED:
            raise ValueError(f"Subagent {agent_id} not yet complete")
        return subagent.result
```

When multiple subagents finish, `collect_all()` gathers their outputs into a single dictionary:

```
    def collect_all(self) -> dict[str, Any]:
        """Collect results from all completed subagents."""
        return {
            agent_id: sa.result
```

```
            for agent_id, sa in self.subagents.items()
            if sa.status == SubagentStatus.COMPLETED
        }
```

Parallel vs. Sequential Delegation

Sometimes subtasks can run in parallel because they don't depend on each other's outputs. Other times, they must run sequentially because each task builds on the previous one. Deep agents need to handle both patterns and can use Python's `asyncio.gather()` to run all subagents at once and wait for all of them to complete:

```
import asyncio

async def parallel_delegation(tasks: list[dict], manager: Subagent
Manager) -> list[Any]:
    """Execute independent tasks in parallel."""

    async def run_task(task):
        agent_id = manager.spawn(
            role=task["role"],
            task=task["description"],
            context=task.get("context", {})
        )
        # Wait for completion (simplified)
        while manager.get_status(agent_id) == SubagentStatus.RUNNING:
            await asyncio.sleep(0.1)
        return manager.get_result(agent_id)

    results = await asyncio.gather(*[run_task(t) for t in tasks])
    return results
```

Some tasks must complete before others start. Each one may depend on previous outputs. The sequential approach builds up accumulated context, feeding each task's result into the next:

```
def sequential_delegation(tasks: list[dict], manager: SubagentManager)
-> list[Any]:
    """Execute dependent tasks sequentially, passing results forward."""

    results = []
    accumulated_context = {}

    for task in tasks:
        # Include results from previous tasks in context
        task_context = {**task.get("context", {}), **accumulated_
context}
```

```
        agent_id = manager.spawn(
            role=task["role"],
            task=task["description"],
            context=task_context
        )

        # Wait and get result
        result = manager.get_result(agent_id)
        results.append(result)

        # Add to accumulated context for next task
        accumulated_context[f"{task['role']}_result"] = result

    return results
```

Hybrid Delegation Patterns

Real workflows often mix parallel and sequential execution. Maybe your research phase can run three searches in parallel, but the writing phase must wait for all research to complete, and editing must wait for writing. A smart coordinator handles this complexity:

```
async def hybrid_delegation(workflow: dict, manager: SubagentManager)
-> dict:
    """Execute a workflow with both parallel and sequential phases."""

    results = {}

    for phase in workflow["phases"]:
        phase_tasks = phase["tasks"]

        if phase.get("parallel", False):
            # Run all tasks in this phase concurrently
            phase_results = await parallel_delegation(phase_tasks,
manager)
        else:
            # Run tasks sequentially
            phase_results = sequential_delegation(phase_tasks, manager)

        results[phase["name"]] = phase_results

        # Add phase results to context for next phase
        for task, result in zip(phase_tasks, phase_results):
            manager.update_shared_context(task["role"], result)

    return results
```

Error Handling in Delegation

Subagents can fail. Networks time out, APIs return errors, and sometimes the subagent just gets confused. Robust delegation handles these failures gracefully:

```
async def resilient_delegation(
    task: dict,
    manager: SubagentManager,
    max_retries: int = 3,
    timeout_seconds: float = 60
) -> dict:
    """Delegate with retry logic and timeout handling."""

    for attempt in range(max_retries):
        try:
            agent_id = manager.spawn(
                role=task["role"],
                task=task["description"],
                context=task.get("context", {})
            )

            # Wait with timeout
            start_time = time.time()
            while manager.get_status(agent_id) == SubagentStatus
.RUNNING:
                if time.time() - start_time > timeout_seconds:
                    manager.terminate(agent_id)
                    raise TimeoutError(f"Subagent {agent_id} timed out")
                await asyncio.sleep(0.5)

            status = manager.get_status(agent_id)
            if status == SubagentStatus.COMPLETED:
                return {"success": True, "result": manager.get_result
(agent_id)}
            elif status == SubagentStatus.FAILED:
                error = manager.get_error(agent_id)
                if attempt < max_retries - 1:
                    continue  # Retry
                return {"success": False, "error": error}

        except TimeoutError as e:
            if attempt < max_retries - 1:
                continue
            return {"success": False, "error": str(e)}

    return {"success": False, "error": "Max retries exceeded"}
```

Collecting and Integrating Results

The coordinator's job is to both delegate tasks *and* to make sense of what comes back. Subagent results often need to be synthesized, conflicts need to be resolved, and gaps need to be identified:

```
def integrate_research_results(results: list[dict], llm) -> dict:
    """Synthesize results from multiple research subagents."""

    prompt = f"""You have research results from multiple specialists:

{format_results(results)}

Synthesize these into a coherent summary:
1. Identify key themes that appear across multiple sources
2. Note any contradictions or disagreements between sources
3. Highlight gaps where more research may be needed
4. Provide an overall conclusion

Be specific and cite which source each finding came from."""

    synthesis = llm.invoke(prompt)

    return {
        "synthesis": synthesis.content,
        "source_count": len(results),
        "sources": [r.get("source") for r in results]
    }
```

Multistep Reasoning

Deep agents reason through complex problems step by step, exploring multiple angles when needed.

Sequential Reasoning Chains

Sometimes you need to think through a problem one step at a time, with each step building on the previous. This is classic chain-of-thought reasoning extended across a longer horizon.

```
def sequential_reasoning(problem: str, max_steps: int, llm) -> dict:
    """Reason through a problem step by step."""

    steps = []
```

```
    current_state = f"Problem: {problem}\n\nLet me think through this
step by step."

    for i in range(max_steps):
        prompt = f"""{current_state}

Step {i + 1}: What's the next logical step in solving this problem?
Consider what we know, what we need to find out, and what follows
logically.

If we have enough information to give a final answer, say "CONCLUSION:
[answer]"
Otherwise, describe your reasoning for this step."""

        response = llm.invoke(prompt).content
        steps.append({"step": i + 1, "reasoning": response})

        if response.startswith("CONCLUSION:"):
            return {
                "conclusion": response.replace("CONCLUSION:", "")
.strip(),
                "steps": steps,
                "total_steps": i + 1
            }

        current_state += f"\n\nStep {i + 1}: {response}"

    return {
        "conclusion": None,
        "steps": steps,
        "total_steps": max_steps,
        "status": "max_steps_reached"
    }
```

When to Stop Going Deeper

Deep reasoning is powerful but expensive. Every additional reasoning step costs time and tokens. Deep agents need heuristics for when to stop.

Here are some reasonable stopping conditions:

- **Confidence threshold:** The agent believes it has a reliable answer.
- **Diminishing returns:** Recent steps aren't adding new insights.
- **Step limit:** A hard cap prevents runaway reasoning.
- **Resource budget:** Time or cost constraints have been reached.
- **Convergence:** Multiple approaches reach the same conclusion.

The following implementation covers three of the five stopping conditions in code. The confidence threshold condition is handled by the check `if confidence >= 0.85:`. When the model reports high confidence, the function immediately prompts for a final answer rather than continuing. Diminishing returns are handled by the plateau check: if confidence hasn't improved by more than 0.05 across the last three steps, the agent stops reasoning and commits. The step limit is the `max_steps` parameter, which provides a hard cap that guarantees termination regardless of the confidence level. The resource budget and convergence checks are left to the caller to implement at a higher level because they depend on system-wide constraints (such as cost, wall-clock time, or results from parallel reasoning paths) that a single function can't evaluate on its own.

```
def adaptive_reasoning(problem: str, llm, max_steps: int = 10) -> dict:
    """Reason with adaptive stopping based on confidence and progress
."""

    steps = []
    confidence_history = []

    for i in range(max_steps):
        # Generate next reasoning step
        context = format_reasoning_history(steps)

        prompt = f"""Problem: {problem}

Reasoning so far:
{context if context else "None yet - this is step 1"}

Continue reasoning. After your analysis, rate your confidence (0.0-1.0)
that you can now give a correct final answer.

Format:
REASONING: Your next step of analysis
CONFIDENCE: 0.X
FINAL_ANSWER: [only if confidence >= 0.85] Your answer"""

        response = llm.invoke(prompt).content
        reasoning, confidence, answer =
parse_reasoning_response(response)

        steps.append({"step": i + 1, "reasoning": reasoning})
        confidence_history.append(confidence)

        # Check stopping conditions
        if answer:  # Model provided final answer
            return {"answer": answer, "steps": steps, "confidence":
confidence}
```

```
        if confidence >= 0.85:  # High confidence - prompt for answer
            final = get_final_answer(problem, steps, llm)
            return {"answer": final, "steps": steps, "confidence":
confidence}

        # Check for plateau (no confidence improvement in 3 steps)
        if len(confidence_history) >= 3:
            recent = confidence_history[-3:]
            if max(recent) - min(recent) < 0.05:
                final = get_final_answer(problem, steps, llm)
                return {"answer": final, "steps": steps, "confidence":
confidence,
                        "note": "Stopped due to confidence plateau"}

    # Max steps reached
    final = get_final_answer(problem, steps, llm)
    return {"answer": final, "steps": steps, "confidence":
confidence_history[-1],
            "note": "Max steps reached"}
```

The key insight is that reasoning should be goal-directed. You're reasoning to reach a reliable conclusion. Once you're confident enough, or once more reasoning isn't helping, it's time to commit to an answer.

Self-Reflection and Metacognition

Here's where things get really interesting. Deep agents can think about their own thinking. They can evaluate whether they're making progress, critique their own outputs, and adjust strategies when something isn't working. Figure 7.5 shows the process.



Figure 7.5: After each task completes, the agent reflects on its progress and either continues to the next task, triggers a replan, or exits when all tasks are done.

Evaluating Progress Toward Goals

A deep agent working on a long project should periodically ask itself: "Am I actually making progress?" This prevents the agent from continuing down unproductive paths:

```
def evaluate_progress(goal: str, plan: Plan, completed_tasks: list[str],
llm) -> dict:
    """Assess progress toward the overall goal."""

    total_tasks = len(plan.tasks)
    completed_count = len(completed_tasks)

    prompt = f"""Goal: {goal}

Original plan:
{format_plan(plan)}

Completed tasks:
{format_completed(completed_tasks)}

Progress: {completed_count}/{total_tasks} tasks complete

Evaluate:
1. Are we on track to achieve the goal?
2. Have completed tasks actually moved us closer to the goal?
3. Are there any warning signs or blockers?
4. Should we adjust the remaining plan?

Be honest and specific."""

    evaluation = llm.invoke(prompt)

    return {
        "progress_fraction": completed_count / total_tasks,
        "evaluation": evaluation.content,
        "status": "on_track" if completed_count / total_tasks > 0.3 else
"needs_review"
    }
```

Strategy Adjustment Based on Results

Sometimes the original plan isn't working. Maybe research is turning up less useful information than expected. Maybe a venue fell through and you need to pivot. Deep agents adjust:

```
def adjust_strategy(current_plan: Plan, evaluation: dict, llm) -> Plan:
    """Modify plan based on progress evaluation."""
```

```
    if "on_track" in evaluation.get("status", ""):
        return current_plan  # No adjustment needed

    prompt = f"""Current plan:
{format_plan(current_plan)}

Progress evaluation:
{evaluation['evaluation']}

The current approach isn't working well. Propose adjustments:
1. Which remaining tasks should be modified or replaced?
2. Are there new tasks that should be added?
3. Should any tasks be removed or deprioritized?
4. What's the revised strategy?

Provide a revised plan."""

    revised = llm.invoke(prompt)

    # In practice, you'd parse this into a new Plan object
    return parse_plan(revised.content)
```

The Reflection Loop

Reflection is most powerful when built into the agent's core loop. After completing significant work, the agent evaluates and potentially revises before moving on. We define a `ReflectiveState` TypedDict that holds everything we need:

```
from langgraph.graph import StateGraph, END
from typing import TypedDict

class ReflectiveState(TypedDict):
    goal: str
    plan: dict
    current_task_idx: int
    task_results: dict
    reflection: str
    needs_replanning: bool
```

Each node function takes state and returns updated state. The `executed_task_node` runs whatever the current task is:

```
def execute_task_node(state: ReflectiveState) -> ReflectiveState:
    """Execute the current task."""
    task = state["plan"]["tasks"][state["current_task_idx"]]
    result = execute_task(task)
```

```
    new_results = {**state["task_results"], task["id"]: result}
    return {"task_results": new_results}
```

After execution, `reflect_node` asks: "Did this move us forward?" It uses the LLM to evaluate and then decides whether we need a strategy change:

```
def reflect_node(state: ReflectiveState) -> ReflectiveState:
    """Reflect on progress after task completion."""

    reflection_prompt = f"""Goal: {state['goal']}

Completed task: {state['plan']['tasks'][state['current_task_idx']]}
Result: {state['task_results']}

Reflect:
1. Did this task achieve what we expected?
2. Are we still on track for the overall goal?
3. Should we adjust the plan?

If the plan needs significant changes, say "REPLAN NEEDED".
Otherwise, say "CONTINUE"."""

    reflection = llm.invoke(reflection_prompt).content

    needs_replan = "REPLAN NEEDED" in reflection
    return {"reflection": reflection, "needs_replanning": needs_replan}
```

If reflection says we need a new strategy, `replan_node` generates an adjusted plan and resets us to the beginning:

```
def replan_node(state: ReflectiveState) -> ReflectiveState:
    """Create a new plan based on reflection."""
    new_plan = adjust_strategy(state["plan"], {"evaluation":
state["reflection"]}, llm)
    return {"plan": new_plan, "needs_replanning": False, "current_task_
idx": 0}

The route_after_reflection function determines where to go next. Should
we continue the plan, replan entirely, or finish:

def route_after_reflection(state: ReflectiveState) -> str:
    """Decide whether to continue, replan, or finish."""
    if state["needs_replanning"]:
        return "replan"
    if state["current_task_idx"] >= len(state["plan"]["tasks"]) - 1:
        return "finish"
    return "continue"
```

Finally, we build the LangGraph. The flow is: execute → reflect → (route) → continue/replan/finish:

```
# Build the graph
graph = StateGraph(ReflectiveState)
graph.add_node("execute", execute_task_node)
graph.add_node("reflect", reflect_node)
graph.add_node("replan", replan_node)

graph.set_entry_point("execute")
graph.add_edge("execute", "reflect")
graph.add_conditional_edges("reflect", route_after_reflection, {
    "continue": "execute",
    "replan": "replan",
    "finish": END
})
graph.add_edge("replan", "execute")

reflective_agent = graph.compile()
```

Backtracking and Recovery

Even with good planning and reflection, agents sometimes go down wrong paths. Deep agents need the ability to recognize dead ends and backtrack to try alternative approaches.

Detecting Dead Ends

A dead end might look like:

- Repeated failures on the same task
- Circular dependencies that can't be resolved
- External constraints that make the current path impossible
- Diminishing confidence in the current approach

Here's some code that implements this detection:

```
def detect_dead_end(state: dict, history: list[dict]) -> dict:
    """Check if the agent is stuck."""

    # Check for repeated failures
    recent_failures = [h for h in history[-5:] if h.get("status") ==
"failed"]
    if len(recent_failures) >= 3:
        return {"is_stuck": True, "reason": "repeated_failures"}
```

```
    # Check for cycling (same task attempted multiple times)
    recent_tasks = [h.get("task_id") for h in history[-10:]]
    for task_id in set(recent_tasks):
        if recent_tasks.count(task_id) >= 3:
            return {"is_stuck": True, "reason": "task_cycling", "task":
task_id}

    # Check for lack of progress
    if len(history) > 10:
        recent_progress = sum(1 for h in history[-10:] if h.get
("progress_made"))
        if recent_progress < 2:
            return {"is_stuck": True, "reason": "no_progress"}

    return {"is_stuck": False}
```

Rolling Back Decisions

When a dead end is detected, the agent needs to roll back to a known good state and try something different. This requires maintaining exploration history:

```
@dataclass
class ExplorationState:
    id: str
    task_state: dict
    decision_made: str
    alternatives: list[str]
    timestamp: float

class ExplorationHistory:
    def __init__(self):
        self.states: list[ExplorationState] = []
        self.current_idx: int = -1

    def checkpoint(self, state: dict, decision: str, alternatives:
list[str]):
        """Save a decision point we might want to return to."""
        exploration_state = ExplorationState(
            id=f"checkpoint_{len(self.states)}",
            task_state=state.copy(),
            decision_made=decision,
            alternatives=alternatives,
            timestamp=time.time()
        )
        self.states.append(exploration_state)
        self.current_idx = len(self.states) - 1
```

```
    def backtrack(self) -> tuple[dict, str] | None:
        """Return to most recent checkpoint with untried alternatives
."""
        for i in range(self.current_idx, -1, -1):
            state = self.states[i]
            if state.alternatives:
                # Try next alternative
                next_alternative = state.alternatives.pop(0)
                self.current_idx = i
                return state.task_state, next_alternative

        return None  # No alternatives left
```

Exploring Alternative Paths

When backtracking, the agent doesn't merely retry the same thing; it tries a different approach. The checkpoint system tracks which alternatives exist and which have been tried:

```
def handle_dead_end(state: dict, history: ExplorationHistory, llm)
-> dict:
    """Recover from a dead end by backtracking and trying alternatives
."""

    backtrack_result = history.backtrack()

    if backtrack_result is None:
        # No alternatives left - report failure
        return {
            "status": "failed",
            "reason": "All alternatives exhausted"
        }

    restored_state, alternative = backtrack_result

    # Log the backtrack
    print(f"Backtracking to try alternative: {alternative}")

    # Continue from restored state with new approach
    return {
        "status": "backtracked",
        "restored_state": restored_state,
        "new_approach": alternative
    }
```

Building StudyBuddy v7

Let's put all of this into practice. We're adding deep agent capabilities to StudyBuddy, and the central addition is the new Curriculum Planner Agent.

Where We Left Off

In v6, StudyBuddy gained persistence and memory. It remembers your learning history across sessions, tracks which concepts you've mastered versus struggled with, and uses the SM-2 algorithm for spaced repetition. The multi-agent team—Tutor, Card Generator, Quality Checker, and Scheduler—works together smoothly under the Learning Coordinator's supervision.

But here's what's missing: StudyBuddy is reactive. It helps you study AI engineering, but it doesn't help you figure out *what* to study or *in what order*. If you want to master AI engineering, you're on your own to decide the path.

What We're Adding

The Curriculum Planner Agent transforms StudyBuddy from a tutoring assistant into a full learning system. Tell it your goal—"I want to be job-ready as an AI engineer in 12 weeks"—and it will:

- Decompose that goal into a structured curriculum
- Identify prerequisite chains (what you need to learn first)
- Create a week-by-week study schedule
- Delegate flashcard generation to the Card Generator for each topic
- Set progress checkpoints with knowledge assessments
- Track your actual progress and adjust the plan when needed
- Handle backtracking when you struggle with a concept

This is deep agent territory: planning, task decomposition, subagent delegation, context management across a multi-week project, and adaptive replanning based on your performance. Let's build it.

The Curriculum Planner Agent

First, we define the planner's role and capabilities:

```
# api/agents/curriculum_planner.py

CURRICULUM_PLANNER_PROMPT = """You are StudyBuddy's Curriculum Planner,
an expert at designing effective learning paths.
```

```
Your job: Take a learning goal and create a structured curriculum that
guides the student from where they are to where they want to be.

When creating a curriculum:
1. Break the goal into major topics or modules
2. Identify prerequisites - what must be learned first
3. Sequence topics logically based on dependencies
4. Estimate time needed for each topic
5. Define checkpoints to assess understanding

Use backward chaining: Start from the goal, identify what's needed to
achieve it, then what's needed for that, until you reach foundational
concepts the student likely already knows.

You have access to the AI Engineering reference materials for domain
knowledge about what topics exist and how they relate.

Output format - respond with JSON:
{
    "goal": "The learning objective",
    "estimated_duration_weeks": 8,
    "modules": [
        {
            "id": "m1",
            "title": "Module title",
            "description": "What this covers",
            "prerequisites": [],
            "topics": ["topic1", "topic2"],
            "estimated_hours": 10,
            "checkpoint": "How to assess completion"
        }
    ],
    "learning_path": ["m1", "m2", "m3"]
}"""

def create_curriculum_planner(model_name: str = "gpt-4o"):
    """Create the Curriculum Planner agent."""
    return ChatOpenAI(model=model_name, temperature=0.4)
```

Task Decomposition for Learning Goals

When a user provides a goal like "Master RAG systems," the planner uses backward chaining to build a curriculum:

```
def generate_curriculum(
    llm: ChatOpenAI,
    goal: str,
    current_knowledge: list[str] = None,
```

```
    available_time_hours: int = None,
    context: str = "",
) -> dict:
    """
    Generate a complete curriculum for a learning goal.

    Args:
        llm: The language model
        goal: What the user wants to learn
        current_knowledge: Topics user already knows
        available_time_hours: Weekly time commitment
        context: Retrieved knowledge base content

    Returns:
        Curriculum dictionary with modules and learning path
    """
    current_knowledge = current_knowledge or []

    user_content = f"""Create a curriculum for this learning goal:
{goal}

Student's current knowledge: {', '.join(current_knowledge) if current_
knowledge else 'Assume foundational programming knowledge'}

{'Weekly time available: ' + str(available_time_hours) + ' hours' if
available_time_hours else ''}

Use backward chaining:
1. What does the student need to know to achieve this goal?
2. For each of those, what prerequisites are needed?
3. Continue until you reach topics the student already knows
4. Sequence everything based on dependencies

Reference material for AI engineering topics:
{context if context else 'Use your knowledge of AI engineering
fundamentals.'}

Provide the curriculum as JSON."""

    messages = [
        SystemMessage(content=CURRICULUM_PLANNER_PROMPT),
        HumanMessage(content=user_content),
    ]

    response = llm.invoke(messages)

    try:
        content = response.content.strip()
        if content.startswith("```"):
            content = content.split("```")[1]
```

```
            if content.startswith("json"):
                content = content[4:]
            content = content.strip()

        return json.loads(content)

    except json.JSONDecodeError:
        return {"error": "Failed to parse curriculum", "raw": response.
content}
```

Subagent Delegation for Flashcard Generation

Once we have a curriculum, the planner delegates flashcard generation to our existing Card Generator agent. Each topic in each module gets its own set of cards:

```
async def generate_curriculum_flashcards(
    curriculum: dict,
    card_generator_llm: ChatOpenAI,
    search_func,
    db,
) -> dict:
    """
    Generate flashcards for all topics in a curriculum.

    Uses the Card Generator agent as a subagent, delegating card
    creation for each topic.
    """
    from .card_generator import generate_cards_batch
    from ..services.flashcard_cache import cache_flashcards

    results = {"modules": {}, "total_cards": 0}

    for module in curriculum.get("modules", []):
        module_id = module["id"]
        module_cards = []

        for topic in module.get("topics", []):
            # Get context from knowledge base
            context = search_func(topic, k=4)

            # Delegate to Card Generator subagent
            cards = generate_cards_batch(
                card_generator_llm,
                topic=topic,
                context=context,
                count=5  # 5 cards per topic
            )
```

```
            if cards:
                # Cache the generated cards
                cached = cache_flashcards(
                    topic=topic,
                    source_context=context,
                    cards_data=cards,
                    db=db,
                    chapter_id=None  # Curriculum cards aren't chapter-
bound
                )
                module_cards.extend(cached)

        results["modules"][module_id] = {
            "title": module["title"],
            "card_count": len(module_cards),
            "card_ids": [c.id for c in module_cards]
        }
        results["total_cards"] += len(module_cards)

    return results
```

Progress Checkpoints and Assessments

The curriculum includes checkpoints, which are moments to assess whether the student has actually learned the material before moving on:

```
def evaluate_checkpoint(
    llm: ChatOpenAI,
    module: dict,
    user_performance: dict,
    db,
) -> dict:
    """
    Evaluate whether user has passed a module checkpoint.

    Args:
        llm: Language model for evaluation
        module: The module being evaluated
        user_performance: Spaced repetition stats for this module's
topics
        db: Database session

    Returns:
        Evaluation with pass/fail and recommendations
    """
    # Calculate performance metrics
    topics = module.get("topics", [])
    avg_ease = user_performance.get("average_ease_factor", 2.5)
```

```
    accuracy = user_performance.get("accuracy", 0)
    cards_studied = user_performance.get("cards_reviewed", 0)

    # Heuristic checkpoint evaluation
    passed = (
        accuracy >= 0.7 and  # 70% accuracy threshold
        avg_ease >= 2.3 and  # Cards aren't too difficult
        cards_studied >= len(topics) * 3  # Studied enough cards
    )

    # Get LLM assessment for nuance
    prompt = f"""Evaluate this student's checkpoint for module: {module
['title']}

Checkpoint criteria: {module.get('checkpoint', 'Demonstrate understanding
of key concepts')}

Performance data:
- Topics covered: {', '.join(topics)}
- Flashcard accuracy: {accuracy:.0%}
- Average ease factor: {avg_ease:.2f}
- Cards reviewed: {cards_studied}

Based on this data, has the student sufficiently mastered this module?
What specific areas might need more attention?

Respond with JSON:
{{"passed": true/false, "confidence": 0.0-1.0, "feedback": "...",
"recommendations": ["..."]}}"""

    response = llm.invoke(prompt)

    try:
        evaluation = json.loads(response.content)
        evaluation["metrics"] = {
            "accuracy": accuracy,
            "ease_factor": avg_ease,
            "cards_studied": cards_studied
        }
        return evaluation
    except:
        return {
            "passed": passed,
            "confidence": 0.7,
            "feedback": "Checkpoint evaluated based on performance
metrics.",
            "metrics": {"accuracy": accuracy, "ease_factor": avg_ease}
        }
```

Context Summarization for Long Study Sessions

When a user has been studying for weeks, we need to keep context manageable. The planner periodically summarizes progress:

```
def summarize_learning_progress(
    curriculum: dict,
    completed_modules: list[str],
    current_module: str,
    struggle_areas: list[str],
    llm: ChatOpenAI,
) -> str:
    """
    Create a compressed summary of learning progress.

    Used to maintain context without overwhelming the context window
    with full history.
    """
    progress_pct = len(completed_modules) / len(curriculum.get
("modules", [])) * 100

    prompt = f"""Summarize this student's learning progress concisely:

Goal: {curriculum['goal']}
Progress: {progress_pct:.0f}% complete

Completed modules:
{chr(10).join(f'- {m}' for m in completed_modules) if completed_modules
else '- None yet'}

Currently studying: {current_module}

Struggle areas: {', '.join(struggle_areas) if struggle_areas else 'None
identified'}

Provide a 2-3 sentence summary capturing:
1. Overall progress status
2. Current focus area
3. Any concerns or adjustments needed"""

    summary = llm.invoke(prompt)
    return summary.content
```

Managing Learning Context Across Sessions

A student might study for 30 minutes today, skip two days, then come back for an hour. The Curriculum Planner needs to maintain coherent context across these

fragmented sessions. We do this by storing session summaries in the database and loading relevant context when the student returns.

```
def get_session_context(user_id: str, curriculum_id: str, db) -> dict:
    """
    Load context for a returning student.

    Combines curriculum state, recent performance, and session history
    into a coherent context object.
    """
    # Get curriculum progress
    progress = db.query(CurriculumProgress).filter_by(
        user_id=user_id,
        curriculum_id=curriculum_id
    ).first()

    # Get recent study sessions (last 5)
    recent_sessions = db.query(StudySession).filter_by(
        user_id=user_id
    ).order_by(StudySession.ended_at.desc()).limit(5).all()

    # Get performance metrics for current module
    current_module = progress.current_module if progress else None
    performance = get_module_performance(user_id, current_module, db) if
current_module else {}

    # Build context
    return {
        "curriculum_goal": progress.goal if progress else None,
        "current_module": current_module,
        "modules_completed": progress.completed_modules if progress
else [],
        "overall_progress": progress.progress_percentage if progress
else 0,
        "recent_sessions": [
            {
                "date": s.ended_at.isoformat(),
                "duration_minutes": s.duration_minutes,
                "topics_studied": s.topics,
                "cards_reviewed": s.cards_reviewed
            }
            for s in recent_sessions
        ],
        "current_performance": performance,
        "days_since_last_session": calculate_days_since(recent_
sessions[0].ended_at) if recent_sessions else None
    }

def resume_study_session(user_id: str, curriculum_id: str, db, llm) ->
dict:
    """
```

```
    Resume a student's study session with appropriate context.

    Handles the case where a student returns after some time away.
    """
    context = get_session_context(user_id, curriculum_id, db)

    days_away = context.get("days_since_last_session")

    if days_away is None:
        # First session ever
        return {
            "message": "Welcome! Let's start your learning journey.",
            "action": "start_curriculum",
            "context": context
        }
    elif days_away > 7:
        # Been away a while - need review
        return {
            "message": f"Welcome back! It's been {days_away} days. Let's
do a quick review before continuing.",
            "action": "review_then_continue",
            "review_topics": context["modules_completed"][-2:],  #
Review recent modules
            "context": context
        }
    elif days_away > 2:
        # Brief break - gentle re-engagement
        summary = summarize_learning_progress(
            curriculum={"goal": context["curriculum_goal"], "modules":
[]},
            completed_modules=context["modules_completed"],
            current_module=context["current_module"],
            struggle_areas=[],
            llm=llm
        )
        return {
            "message": f"Welcome back! Here's where we left off:
{summary}",
            "action": "continue",
            "context": context
        }
    else:
        # Recent session - just continue
        return {
            "message": f"Ready to continue with {context['current_
module']}?",
            "action": "continue",
            "context": context
        }
```

This context management ensures that students get a personalized, coherent experience regardless of their study patterns. The agent adapts its behavior based on how long it's been since the last session and what the student was working on.

Backtracking When Learning Doesn't Work

If a student keeps struggling with a module, the planner can backtrack and try a different approach:

```
def handle_learning_struggle(
    curriculum: dict,
    struggling_module: dict,
    attempts: int,
    llm: ChatOpenAI,
) -> dict:
    """
    Handle when a student is stuck on a module.

    Returns adjusted plan with alternative approaches.
    """
    if attempts < 2:
        # First struggle: suggest review and more practice
        return {
            "action": "reinforce",
            "recommendation": "Review foundational concepts and practice
more flashcards",
            "additional_cards_needed": 10
        }

    elif attempts < 4:
        # Continued struggle: try alternative learning approach
        prompt = f"""A student is struggling with this module despite
multiple attempts:

Module: {struggling_module['title']}
Topics: {', '.join(struggling_module.get('topics', []))}
Attempts: {attempts}

Suggest alternative approaches:
1. Different ways to explain these concepts
2. Prerequisite topics that might need review
3. Simpler stepping stones to build up to this material

Provide JSON:
{{"alternative_approach": "...", "prerequisite_review": ["..."],
"simplified_path": ["..."]}}"""
```

```
        response = llm.invoke(prompt)
        try:
            alternatives = json.loads(response.content)
            return {"action": "alternative_approach", **alternatives}
        except:
            return {"action": "escalate", "recommendation": "Consider
1-on-1 tutoring"}

    else:
        # Persistent struggle: backtrack to prerequisites
        return {
            "action": "backtrack",
            "recommendation": "Return to prerequisite modules",
            "target_modules": struggling_module.get("prerequisites", []),
            "reason": "Foundational gaps detected"
        }
```

Memory-Connected Learning Modes

StudyBuddy v7 connects the tutoring and practice modes through persistent memory. When you struggle with flashcards, that information flows to the tutor. When the tutor identifies concepts you find challenging, those become focus areas for practice.

This bidirectional feedback loop is one of the hallmarks of a deep agent because it maintains context and learns from interactions over time, using that knowledge to personalize future responses.

Struggle Areas Flow to Tutoring

The scheduler tracks which topics you struggle with (less than 60% accuracy after 3+ reviews). These struggle areas are persisted to the memory store:

```
# In scheduler.py - after calculating struggle_topics
memory_store = MemoryStore(db)
for topic in struggle_topics:
    stats = topic_stats[topic]
    accuracy = stats["correct"] / stats["total"]
    memory_store.put(
        user_id=user_id,
        namespace="struggles",
        key=topic.lower().replace(" ", "_"),
        value={
            "topic": topic,
            "accuracy": round(accuracy, 2),
            "total_reviews": stats["total"],
            "identified_at": datetime.utcnow().isoformat(),
```

```
        },
    )
When you ask the tutor a question, it loads your struggle areas to
personalize explanations:
# In tutor_node - before calling tutor_explain
with get_db() as db:
    memory_store = MemoryStore(db)
    struggles = memory_store.search(user_id, namespace="struggles",
limit=5)
    preferences = memory_store.search(user_id, namespace="preferences",
limit=3)

memory_context = format_memories_for_tutor(struggles, preferences)
explanation = tutor_explain(tutor_llm, query, context, card_context,
memory_context)
```

The tutor system prompt includes these struggle areas, so it can be especially clear when explaining related concepts. If you've been struggling with "vector databases" in practice, and you ask the tutor about RAG systems, it knows to be especially thorough when explaining the vector search component.

Learning Insights Flow to Practice

After tutoring sessions, we extract memorable insights synchronously:

```
# After generating an explanation
conversation_text = f"Student asked: {query}\n\nTutor explained:
{explanation}"
memories = extract_memories_from_conversation(conversation_text,
tutor_llm)
for memory in memories:
    memory_store.put(
        user_id=user_id,
        namespace=memory.get("namespace", "background"),
        key=memory.get("key", "unknown"),
        value=memory.get("value", {}),
    )
```

The extraction identifies:

- **Preferences:** How the student likes things explained (visual, step-by-step, etc.)
- **Goals:** What they're studying for (certification, job interview, etc.)
- **Struggles:** Concepts they expressed confusion about during tutoring
- **Background:** Relevant prior knowledge that helps contextualize explanations

This creates a true feedback loop: practice performance informs tutoring, and tutoring insights inform practice recommendations.

The Frontend Shows Focus Areas

The sidebar displays your current struggle areas (labeled "Focus Areas") so you can see what the system has identified. These areas update after each review, so you can watch topics move on and off the list as your performance improves.

The v7 frontend uses Next.js (consistent with v5 and v6) and introduces two new React components for curriculum functionality:

- `FocusAreas.tsx`: Displays struggle topics in the sidebar with "Chat about this" links that automatically open the chat panel with a pre-filled question about the topic.
- `CurriculumModal.tsx`: A modal dialog for creating learning paths. It collects the user's goal and weekly time commitment, shows a loading state while the curriculum is generated, and displays the resulting modules with progress indicators.

The curriculum state is persisted to `localStorage` so students can resume their learning path across browser sessions. The main page component (`page.tsx`) manages two study modes—`chapter` for traditional chapter-based studying and `curriculum` for goal-based learning paths.

This transparency helps students understand why the tutor is being especially thorough on certain topics and gives them a clear sense of progress as they master difficult material.

Putting It Together

The Curriculum Planner integrates with the existing StudyBuddy system through a new API endpoint and updated supervisor routing:

```
# In api/index.py - new endpoint

class CurriculumRequest(BaseModel):
    goal: str
    current_knowledge: list[str] = []
    weekly_hours: int = 10

class CurriculumResponse(BaseModel):
    curriculum: dict
    flashcards_generated: int
    estimated_duration_weeks: int
```

```
@app.post("/api/curriculum", response_model=CurriculumResponse)
async def create_curriculum(request: CurriculumRequest):
    """Generate a personalized learning curriculum."""

    # Search for relevant context
    context = search_materials(request.goal, k=6)

    # Generate curriculum
    curriculum = generate_curriculum(
        curriculum_planner_llm,
        goal=request.goal,
        current_knowledge=request.current_knowledge,
        available_time_hours=request.weekly_hours,
        context=context,
    )

    if "error" in curriculum:
        raise HTTPException(status_code=500, detail=curriculum["error"])

    # Generate flashcards for all topics (async)
    with get_db() as db:
        card_results = await generate_curriculum_flashcards(
            curriculum=curriculum,
            card_generator_llm=card_generator_llm,
            search_func=search_materials,
            db=db,
        )

    return CurriculumResponse(
        curriculum=curriculum,
        flashcards_generated=card_results["total_cards"],
        estimated_duration_weeks=curriculum.get("estimated_duration_
weeks", 8)
    )
```

Curriculum Study Integration

With the curriculum created and flashcards generated, we need a way to fetch cards specifically for the current module the student is studying. This requires a new endpoint and updates to the scheduler's topic filtering:

```
# In api/index.py - curriculum flashcard endpoint

class CurriculumFlashcardRequest(BaseModel):
    curriculum_id: str
    module_id: str | None = None  # None = use current module
    current_topic: str | None = None  # For "Still Learning" flow
    previous_question: str | None = None
```

```
@app.post("/api/curriculum/flashcard", response_model=FlashcardResponse)
def get_curriculum_flashcard(
    request: CurriculumFlashcardRequest,
    db: Session = Depends(get_db),
):
    """Get a flashcard from the current curriculum module.

    Prioritizes spaced repetition (due cards first), then new cards,
    falling back to on-demand generation if no cards exist.
    """
    # 1. Get curriculum and progress
    curriculum = get_curriculum_by_id(db, request.curriculum_id)
    progress = get_curriculum_progress(db, request.curriculum_id,
"default")

    # 2. Determine which module to use
    module_id = request.module_id or (progress.current_module_id if
progress else None)
    current_module = find_module(curriculum, module_id)

    # 3. Get module's topics
    module_topics = current_module.get("topics", [])

    # 4. Try due cards first (spaced repetition priority)
    due_cards = get_due_cards(db, "default", limit=20,
topic_filter=module_topics)
    if due_cards:
        selected = random.choice(due_cards)
        return FlashcardResponse(
            question=selected["question"],
            answer=selected["answer"],
            topic=selected["topic"],
            source="curriculum",
            flashcard_id=selected["flashcard_id"],
        )

    # 5. Then try new cards (not yet reviewed)
    new_cards = get_new_cards(db, "default", limit=20,
topic_filter=module_topics)
    if new_cards:
        selected = random.choice(new_cards)
        return FlashcardResponse(...)

    # 6. No cards found - generate on-demand
    selected_topic = random.choice(module_topics)
    context = search_materials(selected_topic, k=4)
    card = generate_single_card(card_generator_llm, selected_topic, [],
context)

    # Cache and return
    cache_flashcards(topic=selected_topic, cards_data=[card], db=db)
    return FlashcardResponse(...)
```

The key insight is that curriculum modules contain multiple topics, so we need the scheduler's `get_due_cards()` and `get_new_cards()` functions to accept a list of topics:

```
# In api/agents/scheduler.py - updated topic filtering

def get_due_cards(
    db: Session,
    user_id: str = "default",
    limit: int = 10,
    topic_filter: str | list[str] | None = None,  # Now accepts list
) -> list[dict]:
    """Get flashcards due for review, optionally filtered by topic(s)
."""

    query = (
        db.query(CardReview, Flashcard)
        .join(Flashcard, CardReview.flashcard_id == Flashcard.id)
        .filter(CardReview.user_id == user_id, CardReview.next_review
<= now)
    )

    if topic_filter:
        if isinstance(topic_filter, list):
            # Multiple topics - use OR with case-insensitive matching
            from sqlalchemy import or_
            conditions = [Flashcard.topic.ilike(f"%{t}%") for t in
topic_filter]
            query = query.filter(or_(*conditions))
        else:
            # Single topic
            query = query.filter(Flashcard.topic.ilike(f"%{topic_
filter}%"))

    # ... rest of function unchanged
```

This multi-topic filtering enables curriculum study mode to draw from all topics within a module while still respecting spaced repetition priorities. The same update applies to `get_new_cards()`.

Running Locally

StudyBuddy v7 runs with two terminals—one for the backend and one for the frontend.

Terminal 1—Backend:

```
cd v7-deep-agents
uv sync
uv run uvicorn api.index:app --reload --port 8000
```

Terminal 2—Frontend:

```
cd v7-deep-agents/frontend
npm run dev
```

Visit `http://localhost:3000`. The Next.js dev server proxies `/api/*` requests to the FastAPI backend on port 8000 (configured in `next.config.ts`).

Deploying to Vercel

To deploy v7 to Vercel, you need to update two files in the repo root. First, update `vercel.json` to point to the `v7-deep-agents` directory:

```
{
    "version": 2,
    "builds": [
        {
            "src": "v7-deep-agents/api/index.py",
            "use": "@vercel/python"
        },
        {
            "src": "v7-deep-agents/frontend/package.json",
            "use": "@vercel/next"
        }
    ],
    "routes": [
        { "src": "/api/(.*)", "dest": "/v7-deep-agents/api/index.py" },
        { "src": "/(.*)", "dest": "/v7-deep-agents/frontend/$1" }
    ],
    "git": {
        "deploymentEnabled": {
            "main": true
        }
    },
    "ignoreCommand": "bash scripts/should-deploy.sh"
}
```

Second, update `scripts/should-deploy.sh` to trigger on v7 changes:

```
#!/bin/bash
# Deploy if changes are in v7-deep-agents, vercel config, or scripts
# Exit 1 (true) = proceed with build, Exit 0 (false) = skip build
git diff HEAD^ HEAD --name-only | grep -qE "^v7-deep-agents/|^vercel\
.json|^scripts/" && exit 1 || exit 0
```

If you haven't done so already, set your OpenAI API key environment variable in Vercel:

```
vercel env add OPENAI_API_KEY
```

Deploy:

```
vercel --prod
```

Or simply push to GitHub. Vercel will automatically deploy when changes are detected in the `v7-deep-agents` directory.

What's Next

We've built a curriculum planner that can take a high-level learning goal and turn it into a structured study plan with automatically generated flashcards. That's a major upgrade from reactive tutoring.

In Chapter 8, we'll shift focus to evaluation. How do we know if our agents are actually helping students learn? How do we test at scale? We'll dive into synthetic data generation for building evaluation datasets and set up the infrastructure for systematic testing and improvement.

You now have a deep agent that plans, delegates, tracks progress, and adapts. That's a serious capability to add to your toolkit.

Let's keep building.

CHAPTER

8

Evaluation Infrastructure

We've spent seven chapters building increasingly sophisticated AI systems: basic RAG, agentic RAG, multi-agent architectures, memory systems, and deep agents with planning and delegation. These are serious capabilities. But here's a question that should be nagging at you: *How do we know any of this actually works?*

You've run your agents, seen them produce responses, maybe even shown a demo to someone who said, "Wow, that's cool." But "cool" isn't a metric. "Seems to work" isn't a quality bar. And "I think it's getting better" isn't a measurement strategy.

This chapter is about getting serious. We're building the evaluation infrastructure that transforms hunches into hypotheses and vibes into verifiable improvements. Without this foundation, you're flying blind. With it, you can identify weaknesses systematically, measure progress, and build confidence that your system actually does what you think it does.

Why Evaluation Matters

Evaluating AI systems is fundamentally different from evaluating traditional software. With traditional software, you write tests that check specific behaviors: "When I call `add(2, 2)`, I expect to get `4` back." The expected behavior is deterministic and well-defined. Either the function returns 4 or it doesn't.

AI systems don't work that way. Ask your RAG system, "What is machine learning?" and there are a dozen valid responses. Some are better than others, but there's no single correct answer. The response depends on the model's training, the retrieved context, the specific prompt, and even random sampling in generation. You can run the same query multiple times and get meaningfully different responses each time.

This nondeterminism creates four interconnected challenges that make evaluation hard:

- **Outputs are open-ended.** When you ask "explain quantum computing to a beginner," there's no reference answer you can check against. A response might be accurate but confusing. Another might be clear but oversimplified. A third might be technically precise but completely inappropriate for a beginner. You need evaluation criteria that capture what "good" means across multiple dimensions.
- **Quality is subjective.** What's "good enough" depends entirely on context. A response that's perfect for a PhD student might be incomprehensible to a high schooler. A comprehensive 500-word explanation might be exactly right for one use case and too long for another. Your evaluation needs to account for who's using the system and what they're trying to accomplish.
- **Edge cases can be catastrophic.** A system that works great 95% of the time might generate dangerous misinformation the other 5%. Unlike traditional software, where bugs often manifest as crashes or error codes, AI failures can be subtle, confidently wrong, and hard to detect. Evaluation needs to actively seek out these failure modes.
- **Manual evaluation doesn't scale.** You can have humans review a hundred responses. Maybe a thousand if you've got the budget and the patience. But production systems generate millions of responses. You need automated evaluation that can scale without requiring human review of every output.

These challenges explain why vibe checks aren't sufficient as systems mature. Systematic evaluation replaces intuition with data: instead of "I think it's getting better," you can say "faithfulness improved from 0.78 to 0.85 after the prompt change." (We explain what "faithfulness" means in an evaluation context a bit later; refer to the section "The Four Core Metrics.") Numbers let you make decisions, track progress, and catch regressions. The goal isn't to eliminate human judgment, but to automate routine evaluation so humans can focus on cases that actually need their attention.

The Challenge of Creating Test Data

Here's the problem: systematic evaluation requires test data. Lots of it. You need examples that cover the breadth of what your system should handle, including

easy cases, hard cases, edge cases, and adversarial cases. Where does this data come from?

Collecting data from users is the obvious answer, but you face the cold-start problem: you don't have users yet, or enough of them, or they haven't encountered the edge cases you're worried about. Manual test creation has serious limitations because humans are bad at imagining edge cases and tend to write tests for the happy path. A test set of 50 hand-written questions might cover 10% of the actual query space.

Production data, once you have it, comes with privacy challenges and distribution problems. Users may have learned to work around your system's limitations, so their queries won't surface those limitations. What you need is a way to generate test data at scale without requiring manual creation or production traffic. That's where synthetic data comes in.

Introduction to Synthetic Data

Synthetic data is generated data that mimics the characteristics of real data without being collected from real users. For evaluation purposes, synthetic test data consists of questions, expected answers, and other test artifacts generated by models rather than humans.

The core idea is simple: use an LLM to generate the test cases that will be used to evaluate your system. This might sound circular—using AI to evaluate AI—but it works for several reasons. The generation model and evaluation target can be different, so biases don't necessarily transfer. Generated data can be reviewed and filtered by humans, giving you scale with human oversight. And for many evaluation tasks, you're not asking the generator to be creative; you're asking it to produce variations on patterns you've defined.

Synthetic data offers several compelling advantages for evaluation. Scale is the obvious one: you can generate thousands of test cases in minutes rather than months. Coverage improves because you can explicitly generate cases for different scenarios, difficulty levels, and edge cases. There are no privacy concerns because the data doesn't come from real users. And you have control: you can generate exactly the distribution of test cases you want.

But synthetic data isn't a silver bullet. The fundamental limitation is that generated data might not match real usage patterns. If your generation prompts assume users ask polite, well-formed questions, your test set won't prepare you for the terse, typo-filled, ambiguous queries real users submit. There's also quality variance: some generated test cases are excellent, others are nonsensical, and you need filtering to separate them.

Quality validation is essential for synthetic data. You can't just generate a thousand test cases and assume they're all good. Strategies include having humans review a sample to estimate overall quality, using a separate

model to filter out low-quality examples, checking for duplicates and near-duplicates that inflate metrics, and validating that the distribution matches your expectations.

In practice, generating synthetic test cases follows a straightforward pattern. You write a prompt that describes the kind of test case you want—including the topic, difficulty level, and expected format—and then call an LLM to produce a batch of them. For each test case, you typically want a question, an expected answer (which serves as a reference rather than a rigidly correct answer), and metadata like key concepts the answer should mention and the intended difficulty level. You might prompt the model to generate 10 easy factual questions, 15 medium questions that require explanation or connecting concepts, and 5 hard questions that demand synthesis or nuance. The model returns structured JSON that you can parse, validate, and feed into your evaluation pipeline. This is a basic approach; for production evaluation, you want something more sophisticated—which brings us to RAGAS.

NOTE **You can find all of this chapter's non-StudyBuddy code in the Jupyter notebook `chapter08.ipynb`, available in this book's GitHub repo.**

The RAGAS Framework

RAGAS (RAG Assessment) is a framework specifically designed for evaluating retrieval-augmented generation systems. It provides both evaluation metrics and test data generation capabilities, making it a natural fit for the evaluation infrastructure we're building.

The Four Core Metrics

RAGAS defines four primary metrics that together give you a comprehensive view of RAG performance. Think of them as covering the full pipeline: two metrics evaluate the retrieval step (did we find the right documents?) and two evaluate the generation step (did we produce a good answer from those documents?):

- **Faithfulness** measures whether the generated answer sticks to what's actually in the retrieved context. A faithful answer only makes claims that are supported by the source material. An unfaithful answer hallucinates: it invents facts, makes unsupported claims, or goes beyond what the context justifies. RAGAS calculates faithfulness by first decomposing the generated answer into individual claims, then checking each claim against the retrieved context. If your system generates an answer with ten distinct

claims and only seven can be traced back to the context, faithfulness is 0.7. This metric is arguably the most important for any RAG system because hallucination is the primary failure mode you're trying to prevent. A RAG system that makes things up is worse than useless; it's dangerous because users trust it to be grounded in source material.

- **Answer relevance** measures whether the response addresses what was asked. A relevant answer is on-topic and responsive to the question. An irrelevant answer might be factually correct but completely beside the point. Imagine asking "How do I configure chunking?" and getting a technically accurate explanation of what embeddings are. RAGAS assesses relevance by using the LLM to generate potential questions from the answer, then measuring the semantic similarity between those generated questions and the original question. If the answer would naturally prompt questions similar to the one that was asked, it's likely relevant. Low relevance scores often point to problems in your prompt template rather than your retrieval pipeline.
- **Context precision** measures retrieval quality from the precision angle: Of the documents retrieved, what proportion were relevant to answering the question? If you retrieve five chunks and only two are useful, precision is 0.4. RAGAS weights this by rank position, so relevant documents appearing earlier count more than relevant documents buried at the bottom of the results. High precision means your retrieval isn't polluting the context window with irrelevant information, which matters because irrelevant context can degrade generation quality. The model might get confused or distracted by content that has nothing to do with the question.
- **Context recall** measures the other side of retrieval quality: whether the retrieved documents contain the information needed to answer correctly. This requires a ground truth answer for comparison, because RAGAS needs to check what proportion of the expected answer's content is supported by the retrieved context. Low recall means your retrieval is missing information that should be there. Maybe the relevant content exists in your knowledge base but your chunking strategy splits it across chunks that don't get retrieved together, or your embedding model doesn't capture the semantic relationship between the query and the relevant content.

Together, these four metrics map to different failure modes: low faithfulness indicates hallucination; low relevance indicates prompt or generation problems; low precision indicates retrieval noise; and low recall indicates missing information. By watching all four, you can quickly diagnose where problems are occurring in your pipeline and focus your optimization efforts accordingly.

Basic RAGAS Evaluation

Running a basic evaluation with RAGAS requires preparing your evaluation data in a specific format. RAGAS expects a HuggingFace `Dataset` object with columns for `question`, `answer`, `contexts`, and `ground_truth`. The `question` column contains the queries you tested. The `answer` column contains what your RAG system actually generated. The `contexts` column contains the retrieved documents (as a list of strings for each row). And `ground_truth` contains the expected reference answers.

To configure RAGAS, you wrap your LLM and embedding model using `LangchainLLMWrapper` and `LangchainEmbeddingsWrapper` from the RAGAS library. You then initialize each metric—`Faithfulness`, `ResponseRelevancy`, `LLMContextPrecisionWithoutReference`, and `LLMContextRecall`—passing in the wrapped LLM and embeddings. Finally, you call `evaluate()` with your dataset and metrics. RAGAS returns results that you can convert to a pandas DataFrame for analysis.

An important detail: faithfulness and answer relevance don't require ground truth because they can be computed from just the question, answer, and context. Context recall does require ground truth because it measures whether the retrieved context contains the information needed for a known-correct answer. This means you can run a partial evaluation even without reference answers, which is useful when you're getting started and haven't created ground truth yet.

RAGAS makes multiple LLM calls under the hood for each evaluation—decomposing claims, generating questions, comparing content—so evaluating even a small dataset takes a noticeable amount of time and incurs API costs. Plan accordingly: use a less expensive model like `gpt-4o-mini` for evaluation if budget is a concern, and start with small test sets while you're developing your evaluation pipeline.

Interpreting Results

Score interpretation depends on your use case, but here are general guidelines:

METRIC	GOOD	ACCEPTABLE	NEEDS WORK
Faithfulness	> 0.9	0.7–0.9	< 0.7
Answer Relevance	> 0.8	0.6–0.8	< 0.6
Context Precision	> 0.8	0.6–0.8	< 0.6
Context Recall	> 0.8	0.6–0.8	< 0.6

These thresholds aren't absolute. For example, a chatbot for casual product recommendations can tolerate lower scores than a medical information system.

But they give you a starting point for assessing where your system stands and where improvement efforts should focus. If faithfulness is 0.65, that's a red flag: your system is hallucinating frequently. If context precision is 0.55, you're retrieving too much irrelevant content and should look at your retrieval configuration.

RAGAS Testset Generation

Beyond evaluation metrics, RAGAS also provides a powerful testset generation capability that uses a knowledge graph approach to create comprehensive, diverse test cases. The `TestsetGenerator` class takes your source documents, builds an internal knowledge graph that maps entities and relationships, and then generates questions that exercise different types of reasoning.

The process works in three stages:

1. RAGAS analyzes your documents and extracts a knowledge graph, identifying key entities, concepts, and the relationships between them.
2. RAGAS uses this graph to find interesting question patterns: single-hop questions that can be answered from one document, multi-hop questions that require connecting information across documents, and reasoning questions that require inference beyond what's explicitly stated.
3. RAGAS generates question-answer pairs for each pattern, ensuring diversity across question types.

To use the generator, you load your source documents (using LangChain document loaders and text splitters), initialize the `TestsetGenerator` with wrapped LLM and embedding models, and call `generate_with_langchain_docs()` with your documents and desired testset size. The result is a testset object that you can convert to a pandas `DataFrame`. Each row includes the generated question, a reference answer, and metadata about what type of question was generated (the `synthesizer_name` column tells you whether it's a single-hop, multi-hop, or other question type).

A few practical notes:

- RAGAS makes many LLM calls during knowledge graph extraction, so use a model with generous rate limits (like `gpt-4o-mini`) rather than one with restrictive throughput.
- Limit the number of document chunks you feed in during initial testing. Start with a handful of chunks and a small testset size to verify that everything works before scaling up.
- Be aware that testset generation can take several minutes even for small document sets because the knowledge graph extraction is computationally intensive.

Generating Test Cases

Beyond the knowledge graph approach that RAGAS provides, there are other patterns for generating effective test cases that stress different parts of your system. The key insight is that a single approach to test generation tends to produce a homogeneous test set that misses important failure modes. You want diversity across multiple dimensions: difficulty level, question type, and adversarial intent.

Difficulty-Based Generation

Different difficulty levels stress different parts of your system in useful ways. Easy questions are direct lookup tasks where the answer appears almost verbatim in your source documents. These test basic retrieval: Can your system find the right chunk and extract the answer? If you're failing easy questions, you have a fundamental retrieval problem that needs fixing before anything else.

Medium questions require understanding and paraphrasing. The answer exists in the documents, but not as a directly reusable snippet. Your system needs to understand the content and restate it in a way that addresses the specific question. These test the generation side of your pipeline: Can the model synthesize a coherent answer from relevant context? Medium questions also test whether your retrieval can find content when the query uses different terminology than the source documents.

Hard questions are where things get interesting. These require synthesis across multiple concepts, understanding of edge cases or exceptions to general rules, or nuanced distinctions that a shallow reading would miss. A hard question might ask about a scenario where two general principles conflict, or about the limitations of a technique that's usually presented in a positive light. Hard questions test whether your system can reason rather than just retrieve and repeat.

To generate test cases across the difficulty spectrum, prompt your LLM with explicit descriptions of what each level means, along with the source documents to draw from. Ask for structured output that includes not just the question and expected answer, but also metadata: for easy questions, the exact source quote; for medium questions, the reasoning chain; for hard questions, an explanation of why it's difficult. This metadata is invaluable when you're debugging low scores later.

A typical distribution for a balanced test set is roughly 30% easy, 50% medium, and 20% hard. Easy questions establish a performance floor, medium questions represent the bulk of real usage, and hard questions reveal the system's ceiling. If your system performs well on easy questions but struggles with medium ones, you know the problem is likely in generation rather than retrieval.

Question-Type Diversity

Difficulty (above) and adversarial intent (below) specify, respectively, how hard a question is and how challenging it's designed to be. Question type tackles a different dimension: what kind of "cognitive work" is required to answer each question. A test set dominated by a single question type will give you a misleadingly narrow picture of performance, even if the set has well-distributed difficulty and adversarial test cases.

There are four general question types you should consider:

- **Factual recall questions** are the most common type and are the easiest to generate. These questions have a single correct answer that lives somewhere in your documents. For example, "What year did the Berlin Wall fall?" or "What is the boiling point of water at sea level?" Factual recall questions test whether your retrieval can find the right chunk and whether your generation can extract the answer cleanly. Most automatically generated test sets are heavily weighted toward factual recall, which is why it's a good idea to deliberately generate other question types.
- **Explanatory questions** are "how" and "why" queries that require the model to synthesize a coherent explanation of something from potentially scattered context. For example, "Why does sourdough bread require a starter culture?" or "How does a vaccine train the immune system?" These kinds of questions test whether your system can construct a logical narrative and not just retrieve facts.
- **Comparative questions** ask the system to evaluate two or more things relative to each other. For example, "What is the difference between a lager and an ale?" or "How does classical conditioning differ from operant conditioning?" These kinds of questions require the system to consider multiple concepts simultaneously and articulate distinctions.
- **Procedural questions** ask the system for a sequence of steps or a how-to. For example, "How do I proof yeast before baking bread?" or "What's the process for changing a tire safely on the highway?" These kinds of questions test whether your system can construct an ordered response that users can follow to perform some task.

To generate different question types, prompt your LLM with explicit type definitions and ask it to produce a specified number of each type. A reasonable distribution for most knowledge bases is 30 percent factual recall, 25 percent explanatory, 25 percent comparative, and 20 percent procedural. However, the right mix depends on your content, the goals of your system, and how your users interact with the system.

Adversarial Test Cases

Your system will encounter queries designed to challenge it, whether intentionally adversarial or just unusual. Generating adversarial test cases proactively helps you find and fix failure modes before your users discover them.

There are four main categories of adversarial tests, each targeting a different failure mode:

- **Unanswerable questions** are queries that seem related to your domain but can't actually be answered from your documents. A good system should recognize that it doesn't have the information and say so, rather than confabulating a plausible-sounding answer. For example, if your knowledge base covers Python and JavaScript, an unanswerable question might ask about Rust's memory model; it's programming-related, but your system shouldn't pretend to have that information.
- **Ambiguous questions** can be interpreted multiple ways. "How do you handle errors?" could mean Python exception handling, HTTP error codes, or user-facing error messages. A good system should either ask for clarification, acknowledge the ambiguity and address the most likely interpretation, or give a brief answer covering the main interpretations.
- **Misleading questions** contain false premises or incorrect assumptions. "Since RAG systems don't need embeddings, how do they find relevant documents?" has a false premise: RAG systems absolutely do use embeddings! A good system should correct the assumption rather than accepting the faulty premise and answering based on it. This is a particularly important failure mode because users who have misconceptions are exactly the users who need accurate information most.
- **Out-of-scope questions** are related to your domain but outside what your documents actually cover. If your knowledge base covers RAG fundamentals but not advanced topics like constitutional AI, an out-of-scope question about constitutional AI should get an honest "I don't have detailed information on that topic" rather than a hallucinated answer. The system should acknowledge its limitations gracefully.

When generating adversarial test cases, include not just the question and expected behavior, but also what a bad response would look like. This makes it easier to write evaluators that can distinguish between good and bad handling of adversarial inputs.

Data Quality and Coverage

Not every generated test case is good. Common problems include nonsensical questions that don't make sense in context, incorrect expected answers where

the model got the reference wrong, near-duplicates that inflate your test set size without adding meaningful coverage, and questions that are either so vague they could mean anything or so specific they're trivially easy.

Building a validation pipeline is essential. The approach is to use a separate LLM call to evaluate each generated test case on multiple quality dimensions: Is the question clear and unambiguous? Is the expected answer actually correct based on the source documents? Can the question reasonably be answered from the available documents? And is this a useful test case—neither trivially easy nor impossibly hard? Score each dimension on a scale (1–5 works well) and filter out test cases below a quality threshold. A threshold of 3.5 out of 5 is a reasonable starting point, keeping roughly the top 70% of generated cases.

Deduplication is another critical step. LLMs tend to generate variations on the same themes, especially when prompted with the same source documents multiple times. Use semantic similarity to identify near-duplicates—questions that are worded differently but test the same concept in the same way—and keep only the best version of each. Your final test set should be diverse and high-quality rather than large and repetitive.

Beyond quality, you need to verify that your test set actually covers the breadth of your system's expected behavior. Coverage verification checks two dimensions: topic coverage and type coverage.

Topic coverage asks whether your test cases touch all the subjects your system should handle. If your knowledge base covers twelve different AI engineering topics but your test set only generates questions about three of them, you're leaving nine topics completely untested. A simple approach is to define your expected topic list up front, then check each generated question against that list using keyword matching or semantic similarity. Identify gaps—topics with zero test cases—and underrepresented areas—topics with only one or two cases—and generate extra targeted test cases to fill them.

Type coverage ensures that you have a mix of question types: factual recall, explanatory, comparative, procedural, and so on. If all your test cases are factual recall questions, you're not testing whether your system can explain concepts or walk through procedures. Similarly, if you've generated adversarial cases for only one of the four adversarial categories, you've got coverage gaps.

Build a coverage analysis function that takes your test cases and expected dimensions, then reports on what's covered and what's missing. Run this after every test generation pass and use the gaps to guide additional generation. The goal is a test set that exercises your system comprehensively, not just thoroughly in one area.

Plan for your test set to evolve over time. As your system improves, easy test cases become less informative and you need to generate harder ones. As you add new content to your knowledge base, you need new test cases that cover it. And as users report problems, you should add those real-world failure cases to your regression test set. Think of test set maintenance as an ongoing practice, not a one-time setup.

Metrics-Driven Development

Evaluation isn't about measuring quality once and calling it a day. It's about using measurements to drive systematic improvement. This is the philosophy of metrics-driven development: set baselines, make changes, measure impact, and iterate. Every change to your system—whether it's a prompt tweak, a retrieval configuration change, or a model upgrade—should be evaluated against your test set so you can see exactly what improved, what regressed, and what stayed the same.

The Improvement Loop

The basic improvement loop is straightforward in concept:

1. **Measure.** Run your evaluation suite and record scores across all metrics.
2. **Analyze.** Identify what's causing low scores. Is it a retrieval problem, a generation problem, or a data problem? Look at individual failure cases, not just aggregate metrics.
3. **Hypothesize.** Form a theory about what would improve things. "If we increase chunk overlap, we'll capture more context at chunk boundaries, which should improve recall."
4. **Change.** Implement your proposed improvement. Change one thing at a time so you can attribute any metric changes to that specific modification.
5. **Measure again.** Rerun the same evaluation suite with the same test data. Compare the new scores to your baseline.
6. **Iterate.** If it helped, keep it and look for the next improvement. If it didn't, revert and try a different hypothesis.

The discipline of this loop is what separates engineering from tinkering. Without measurement, you're making changes based on intuition and hoping they help. With measurement, you have evidence. You can tell your team, "This prompt change improved faithfulness by 7% with no regression in relevance," rather than, "I think this is better."

To support this loop, you need an evaluation tracker that records the results of every evaluation run along with the configuration that produced it. At minimum, track the timestamp, a version identifier, the metric scores, the configuration parameters (model, temperature, chunk size, prompt version, and so on), and any notes about what changed. This lets you compare any two versions side by side, spot trends over time, and detect regressions before they reach production.

Setting Quality Thresholds

Before you can improve, you need to define what "good enough" looks like. Quality thresholds should be set at three levels:

- The *minimum threshold* represents the floor below which your system is broken and shouldn't be deployed. For example, faithfulness below 0.7 means the system is hallucinating too often to be trusted.
- The *target threshold* represents the quality level you're aiming for. It's the point at which users would consider the system reliable and useful.
- The *stretch goal* represents excellence. It's the aspirational quality bar that you're working toward over the long term.

The right thresholds depend entirely on your use case. For a medical information system, faithfulness needs to be near-perfect—anything below 0.95 could mean patients receive dangerous misinformation. For a creative writing assistant, some "hallucination" might actually be desirable: generating novel ideas beyond the source material is a feature, not a bug. For an educational tutoring system like StudyBuddy, you want high faithfulness (students need accurate information) and high pedagogical quality (accurate information delivered poorly doesn't help learning).

Set thresholds based on user impact: At what quality level would users be satisfied? At what level might there be actual harm? Build an assessment function that takes current metric scores and evaluates them against your thresholds, returning a clear status: "critical" if any metric is below minimum, "improving" if some metrics are below target, and "good" if all targets are met. This makes it easy to monitor system health at a glance.

Avoiding Goodhart's Law

Goodhart's Law states: "When a measure becomes a target, it ceases to be a good measure." This is a real risk in AI evaluation. If you optimize purely for faithfulness, you might end up with a system that only quotes verbatim from source documents—technically faithful but useless as a tutor. If you optimize purely for relevance, you might get verbose answers that technically address the question but bury the useful information in padding.

Protect against this by using multiple metrics together so that gaming one metric hurts another. Maintain human evaluation as a periodic sanity check by having real people use the system and give qualitative feedback. Periodically review failure cases qualitatively rather than just looking at aggregate scores. And update your test sets as the system evolves, because a test set that was challenging six months ago might be too easy now, creating the illusion of sustained high performance.

The goal of evaluation is to understand your system's actual capabilities and limitations, not to produce impressive numbers. If your metrics look great but users are complaining, the metrics are wrong, or at least incomplete. Always prioritize real-world performance over benchmark scores.

Loading Datasets into LangSmith

LangSmith provides infrastructure for managing evaluation datasets, running evaluations at scale, and tracking results over time. It's the natural place to store your synthetic test data and run systematic evaluations because it handles the bookkeeping that's tedious to build yourself: dataset versioning, experiment tracking, result comparison, and visualization.

Creating and Managing Datasets

The LangSmith `Client` class is your entry point for dataset management. You create a dataset by calling `client.create_dataset()` with a name and description, then add examples using `client.create_examples()`. Each example has an `inputs` dictionary (what gets fed to your system) and an `outputs` dictionary (the expected results). For a tutoring evaluation dataset, the inputs would include the question, topic, and difficulty metadata, while the outputs would include the reference answer and key concepts.

When populating datasets from RAGAS-generated test sets, you iterate over the testset `DataFrame` rows, mapping the generated questions and ground truth answers to the LangSmith example format. Add metadata like topic and difficulty level to enable filtering and analysis later—you'll want to know whether your system struggles more with certain topics or certain difficulty levels.

Organizing Datasets

For serious evaluation, you'll want multiple datasets, each serving a different purpose:

- **Golden set:** A small, high-quality collection of human-verified examples used for critical testing. These are your most trusted test cases—every example has been reviewed by a human for correctness. Use this set for release gates and regression testing.
- **Regression set:** Cases that previously failed and are now fixed. You add to this set every time you fix a bug or failure mode. Running the regression set ensures that past fixes stay fixed—that new changes don't reintroduce old problems.

- **Comprehensive set:** A large synthetic set for broad coverage testing. This is your main evaluation set—hundreds or thousands of test cases covering the full breadth of your system's capabilities. Use this for general quality assessment and trend tracking.
- **Adversarial set:** Edge cases, adversarial queries, and unusual inputs designed to probe failure modes. This set helps you find the boundaries of your system's capabilities and ensures it fails gracefully when it can't handle a query.

Create each dataset with a clear name and description so your team knows what it's for. Use version suffixes (like `v1`, `v2`) when you make significant changes to a dataset, and keep the old version around so you can compare results across dataset versions.

Running Evaluations

LangSmith's `evaluate()` function is the core mechanism for running your system against a dataset with custom evaluators. You provide three things: a function that takes inputs and returns outputs (this is your system under test), the name of the dataset to evaluate against, and a list of evaluator functions that score each result.

Your system function receives the `inputs` dictionary from each dataset example and returns an `outputs` dictionary with whatever your evaluators need to assess. For a RAG system, this typically means returning both the generated answer and the retrieved contexts, since evaluators like faithfulness need both.

Each evaluator function receives two arguments: the `run` (containing your system's outputs) and the `example` (containing the dataset's inputs and expected outputs). The evaluator returns a dictionary with a key name, a numeric score, and an optional comment. You can use RAGAS metrics inside your evaluators, implement custom scoring logic, or combine both approaches.

The `experiment_prefix` parameter lets you label each evaluation run, making it easy to find and compare experiments later. Use descriptive prefixes that encode the change you're testing, such as: `rag-v2.1-larger-chunks` or `rag-v2.2-reranking-enabled`. Results are stored in LangSmith automatically, so you can compare experiments through the web UI or programmatically via the API.

Comparing Experiments

Once you've run multiple experiments, LangSmith lets you compare them side by side. You can list projects by experiment prefix, retrieve the runs for each experiment, and calculate aggregate metrics for comparison. The LangSmith web UI provides dashboards for visualizing these comparisons, making it easy to see trends and identify regressions.

Programmatic comparison is useful for CI/CD integration. You can write scripts that compare the latest experiment against the previous one, check whether any metrics regressed beyond a threshold, and block deployments if quality gates aren't met. This turns evaluation from a manual, periodic activity into an automated part of your development workflow.

The combination of automated evaluation in your development loop and a visual dashboard for broader analysis gives you both the speed of automation and the insight of human review. You can iterate quickly, catch regressions automatically, and periodically step back to look at the bigger picture of how your system is evolving over time.

Building StudyBuddy v8

Time to apply what we've learned. In the last chapter, we built a curriculum planner with automatically generated flashcards. Now we're adding evaluation infrastructure to systematically measure and improve quality.

What We're Adding

StudyBuddy v8 introduces comprehensive evaluation infrastructure:

- **Synthetic test data generation** using the RAGAS knowledge graph approach
- **Evaluation datasets** covering tutoring quality, flashcard effectiveness, and retrieval accuracy
- **LangSmith integration** for dataset management and experiment tracking
- **Baseline metrics** to measure current performance and track improvements
- **Custom evaluators** specific to educational content quality

Setting Up RAGAS for Our Domain

First, let's configure RAGAS to generate test data from our reference materials. Note that we're using RAGAS 0.4.x, which has a different API than earlier versions. Before we can generate test data, we need to set up our environment properly and load the source materials that will form the basis of our synthetic tests:

```
# api/evaluation/testset_generator.py
from pathlib import Path

# Load environment variables before any LangChain/RAGAS imports
# This ensures LANGCHAIN_TRACING_V2=false is respected
from dotenv import load_dotenv
```

```
load_dotenv()

# RAGAS uses asyncio.run() internally, which conflicts with FastAPI's
event loop.
# nest_asyncio allows nested event loops to work around this.
import nest_asyncio
nest_asyncio.apply()
```

Now we bring in the RAGAS components and LangChain utilities:

```
from ragas.testset import TestsetGenerator
from ragas.llms import LangchainLLMWrapper
from ragas.embeddings import LangchainEmbeddingsWrapper
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_core.documents import Document
import pandas as pd
```

The first function handles loading and chunking our reference documents. Notice that we use larger chunks than typical RAG: 1,500 characters with 200 overlap, which gives the test generator more context to work with:

```
def load_reference_documents(docs_path: str = "documents") -> list
[Document]:
    """Load and chunk reference documents for test generation."""
    documents_dir = Path(__file__).parent.parent.parent / docs_path
    guide_files = sorted(documents_dir.glob("ref-*.md"))

    # Use larger chunks for test generation
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=1500,
        chunk_overlap=200,
    )

    all_chunks = []
    for filepath in guide_files:
        loader = TextLoader(str(filepath))
        documents = loader.load()
        chunks = splitter.split_documents(documents)
        all_chunks.extend(chunks)
    return all_chunks
```

Next, we initialize the testset generator. The comment explains why we're using gpt-4o-mini: rate limits. They are critical here because RAGAS makes many LLM calls during knowledge graph extraction:

```
def create_testset_generator() -> TestsetGenerator:
    """Initialize RAGAS testset generator with wrapped LangChain models.
```

```
    Uses gpt-4o-mini to avoid rate limits - RAGAS makes many LLM
calls during
    knowledge graph extraction, and gpt-4o's 30k TPM limit is too
restrictive.
    """
    generator_llm = LangchainLLMWrapper(ChatOpenAI(model="gpt-4o-mini"))
    embeddings = LangchainEmbeddingsWrapper(OpenAIEmbeddings())
    return TestsetGenerator(llm=generator_llm,
embedding_model=embeddings)
```

Finally, the main generation function. RAGAS 0.4x's `generate_with_langchain_docs()` automatically creates diverse question types (simple, reasoning, and multi-hop), and we rename the columns to match our internal naming conventions:

```
def generate_tutoring_testset(
    documents: list[Document],
    test_size: int = 100
) -> pd.DataFrame:
    """Generate test cases for tutoring evaluation."""
    generator = create_testset_generator()

    # RAGAS 0.4.x automatically generates diverse question types
    testset = generator.generate_with_langchain_docs(
        documents=documents,
        testset_size=test_size,  # Note: parameter name is testset_size
    )

    df = testset.to_pandas()

    # Rename columns for consistency with our evaluation system
    if "user_input" in df.columns:
        df = df.rename(columns={"user_input": "question"})
    if "reference" in df.columns:
        df = df.rename(columns={"reference": "ground_truth"})

    return df
```

Note that RAGAS 0.4.x requires additional packages: `rapidfuzz` for string distance calculations and `nest-asyncio` to work within FastAPI's event loop. The `nest_asyncio.apply()` call must happen before importing RAGAS to avoid "Event loop is closed" errors.

Building Evaluation Datasets

We need different evaluation datasets for different purposes:

```
# api/evaluation/dataset_builder.py
from langsmith import Client
```

```
import json

client = Client()

def create_tutoring_evaluation_dataset(testset_df, dataset_name: str):
    """Create LangSmith dataset for tutoring evaluation."""
    dataset = client.create_dataset(
        dataset_name=dataset_name,
        description="Synthetic test cases for tutoring quality
evaluation"
    )

    for _, row in testset_df.iterrows():
        client.create_example(
            inputs={
                "question": row["question"],
                "topic": extract_topic(row["question"]),
                "difficulty": row.get("evolution_type", "simple")
            },
            outputs={
                "reference_answer": row["ground_truth"],
                "key_concepts": extract_key_concepts(row["ground_truth"])
            },
            dataset_id=dataset.id
        )
    return dataset

def create_flashcard_evaluation_dataset(topics: list[str]):
    """Create dataset for flashcard quality evaluation."""
    dataset = client.create_dataset(
        dataset_name="flashcard-quality-v1",
        description="Test cases for flashcard clarity and effectiveness"
    )

    # Generate flashcard evaluation cases
    llm = ChatOpenAI(model="gpt-4o")
    for topic in topics:
        # Generate examples of good and bad flashcards for each topic
        prompt = f"""For the topic "{topic}", generate examples for
flashcard evaluation:

1. A GOOD flashcard (clear, focused, testable)
2. A BAD flashcard - too vague
3. A BAD flashcard - too complex (multiple concepts)
4. A BAD flashcard - unclear wording

For each, provide:
- question: The flashcard question
- answer: The expected answer
- quality: "good" or "bad"
```

```
- issue: If bad, what's wrong with it

Format as JSON array."""
        examples = json.loads(llm.invoke(prompt).content)
        for ex in examples:
            client.create_example(
                inputs={
                    "topic": topic,
                    "flashcard_question": ex["question"],
                    "flashcard_answer": ex["answer"]
                },
                outputs={
                    "expected_quality": ex["quality"],
                    "expected_issue": ex.get("issue", "none")
                },
                dataset_id=dataset.id
            )
    return dataset

def create_retrieval_evaluation_dataset(documents: list):
    """Create dataset for retrieval quality evaluation."""
    dataset = client.create_dataset(
        dataset_name="retrieval-quality-v1",
        description="Test cases for measuring retrieval accuracy"
    )

    llm = ChatOpenAI(model="gpt-4o")
    # For each document, generate questions it should answer
    for doc in documents:
        prompt = f"""Based on this content, generate 3 questions
that should
retrieve this document:

Content:
{doc.page_content[:1500]}

For each question:
- question: The query
- expected_chunks: Key phrases that should appear in retrieved content
- source_file: {doc.metadata.get('source', 'unknown')}

Format as JSON array."""
        questions = json.loads(llm.invoke(prompt).content)
        for q in questions:
            client.create_example(
                inputs={"query": q["question"]},
                outputs={
                    "expected_chunks": q["expected_chunks"],
                    "source_file": q["source_file"]
```

```
                },
                dataset_id=dataset.id
            )
    return dataset
```

Custom Evaluators for Educational Content

Generic RAGAS metrics measure things like faithfulness and relevance, but educational content has specific quality requirements that demand custom evaluation logic.

First, we define a helper function that extracts JSON from LLM responses. LLMs often wrap JSON in Markdown code blocks, so we need to handle both cases:

```
# api/evaluation/evaluators.py
import json
import re
import logging
from langchain_openai import ChatOpenAI

logger = logging.getLogger(__name__)

def _extract_json(text: str) -> dict:
    """Extract JSON from LLM response, handling markdown code blocks.

    LLMs often wrap JSON in markdown code blocks like ```json ... ```.
    This helper extracts the JSON regardless of formatting.
    """
    # Try to extract JSON from markdown code block
    json_match = re.search(r"```(?:json)?\s*([\s\S]*?)```", text)
    if json_match:
        return json.loads(json_match.group(1).strip())
    # Try parsing the whole response as JSON
    return json.loads(text)
```

The tutoring quality evaluator judges how well our tutor helps students learn. It scores five dimensions: accuracy, clarity, completeness, pedagogy (does it teach effectively?), and engagement:

```
def tutoring_quality_evaluator(run, example) -> dict:
    """Evaluate tutoring response quality for learning effectiveness."""
    llm = ChatOpenAI(model="gpt-4o-mini")
    question = example.inputs.get("question", "")
    response = run.outputs.get("response", "")
    reference = example.outputs.get("reference_answer", "")

    prompt = f"""Evaluate this tutoring response for educational
effectiveness.
```

```
Student Question: {question}
Tutor Response: {response}
Reference Answer: {reference}

Evaluate on these criteria (1-5 scale):
1. ACCURACY: Is the information factually correct?
2. CLARITY: Is the explanation clear and easy to understand?
3. COMPLETENESS: Does it cover the key concepts needed to answer?
4. PEDAGOGY: Does it teach effectively (examples, analogies, building
blocks)?
5. ENGAGEMENT: Is it engaging and encouraging for a learner?

Provide scores and brief justification for each.
Format as JSON: {{"accuracy": N, "clarity": N, "completeness": N,
"pedagogy": N,
"engagement": N, "overall": N, "justification": "..."}}
The overall score should be the average of the five criteria, rounded to
nearest integer."""

    try:
        result = _extract_json(llm.invoke(prompt).content)
        overall = result.get("overall", 3)
        return {
            "key": "tutoring_quality",
            "score": overall / 5.0,  # Normalize to 0-1
            "comment": result.get("justification", "")
        }
    except (json.JSONDecodeError, Exception) as e:
        logger.warning(f"Tutoring evaluator failed: {e}")
        return {
            "key": "tutoring_quality",
            "score": 0.5,  # Default middle score on failure
            "comment": f"Evaluation failed: {e}"
        }
```

The flashcard quality evaluator examines whether generated flashcards are effective learning tools. Good flashcards test exactly one concept clearly and unambiguously:

```
def flashcard_quality_evaluator(run, example) -> dict:
    """Evaluate generated flashcard quality."""
    llm = ChatOpenAI(model="gpt-4o-mini")
    topic = example.inputs.get("topic", "")
    generated_q = run.outputs.get("question", "")
    generated_a = run.outputs.get("answer", "")

    prompt = f"""Evaluate this flashcard for learning effectiveness.

Topic: {topic}
Question: {generated_q}
Answer: {generated_a}
```

```
Evaluate:
1. FOCUS: Does it test exactly ONE concept? (not too broad or narrow)
2. CLARITY: Is the question unambiguous?
3. TESTABILITY: Can someone clearly know if they got it right?
4. ANSWER_QUALITY: Is the answer accurate and appropriately detailed?
5. LEARNING_VALUE: Will this help someone learn the topic?

Score each 1-5, then overall 1-5.
Identify any specific issues.
Format as JSON: {{"focus": N, "clarity": N, "testability": N, "answer_
quality": N,
"learning_value": N, "overall": N, "issues": ["..."]}}"""

    try:
        result = _extract_json(llm.invoke(prompt).content)
        overall = result.get("overall", 3)
        issues = result.get("issues", [])
        return {
            "key": "flashcard_quality",
            "score": overall / 5.0,
            "comment": "; ".join(issues) if issues else "No issues
found"
        }
    except (json.JSONDecodeError, Exception) as e:
        logger.warning(f"Flashcard evaluator failed: {e}")
        return {
            "key": "flashcard_quality",
            "score": 0.5,
            "comment": f"Evaluation failed: {e}"
        }
```

The retrieval precision evaluator checks whether our system retrieves content containing the phrases we expect. This is simpler than LLM-based evaluation (we just check string matches):

```
def retrieval_precision_evaluator(run, example) -> dict:
    """Evaluate whether retrieval found the right content."""
    expected_chunks = example.outputs.get("expected_chunks", [])
    retrieved = run.outputs.get("contexts", [])

    if not expected_chunks:
        return {
            "key": "retrieval_precision",
            "score": 1.0,  # No expectations = automatic pass
            "comment": "No expected chunks to verify"
        }

    # Combine all retrieved contexts into one searchable string
    if isinstance(retrieved, list):
        retrieved_text = " ".join(str(c) for c in retrieved).lower()
```

```
    else:
        retrieved_text = str(retrieved).lower()

    # Check how many expected phrases appear in retrieved content
    found = sum(1 for chunk in expected_chunks if chunk.lower() in
retrieved_text)
    precision = found / len(expected_chunks)

    return {
        "key": "retrieval_precision",
        "score": precision,
        "comment": f"Found {found}/{len(expected_chunks)} expected
chunks"
    }
```

Running Baseline Evaluation

Now let's establish our baseline metrics. The key insight here is that LangSmith's `evaluate()` function returns an `ExperimentResults` object that you iterate over. Each result contains `evaluation_results` with a list of `EvaluationResult` objects that have `.key` and `.score` attributes. We'll start with a helper that attempts to connect to LangSmith and returns None if unavailable, letting us skip evaluations gracefully during local development without LangSmith credentials:

```
# api/evaluation/run_baseline.py
import logging
from langsmith import Client
from langsmith.evaluation import evaluate
from .evaluators import (
    tutoring_quality_evaluator,
    flashcard_quality_evaluator,
    retrieval_precision_evaluator
)

logger = logging.getLogger(__name__)

def _get_langsmith_client() -> Client | None:
    """Get LangSmith client with error handling."""
    try:
        client = Client()
        list(client.list_datasets(limit=1))
        return client
    except Exception as e:
        logger.warning(f"LangSmith unavailable: {e}")
        return None
```

The main tutoring evaluation function uses LangSmith's `evaluate()` function. The key insight is that it returns an iterator (`ExperimentResults`) that you must

consume. Each result contains evaluation results with `.key` and `.score` attributes rather than dictionary access:

```
def run_tutoring_evaluation(
    tutor_func,
    dataset_name: str = "tutoring-evaluation-v1",
    experiment_prefix: str = "tutor-baseline"
) -> dict:
    """Run tutoring evaluation against LangSmith dataset."""
    client = _get_langsmith_client()
    if not client:
        return {"error": "LangSmith unavailable", "tutoring_quality":
0.0}

    try:
        results = evaluate(
            tutor_func,
            data=dataset_name,
            evaluators=[tutoring_quality_evaluator],
            experiment_prefix=experiment_prefix,
        )

        # Extract aggregate score from ExperimentResults
        # Each result is a dict with 'run', 'example', 'evaluation_
results' keys
        # evaluation_results['results'] contains EvaluationResult
objects
        scores = []
        result_list = list(results)  # Consume the iterator
        for r in result_list:
            eval_results = r.get("evaluation_results", {})
            results_list = eval_results.get("results", [])
            for eval_result in results_list:
                # EvaluationResult objects have .key and .score
attributes
                key = getattr(eval_result, "key", None)
                score = getattr(eval_result, "score", None)
                if key == "tutoring_quality" and score is not None:
                    scores.append(score)

        avg_score = sum(scores) / len(scores) if scores else 0.0
        logger.info(f"Tutoring evaluation complete: {len(scores)}
samples, avg={avg_score:.3f}")
        return {
            "tutoring_quality": avg_score,
            "sample_size": len(scores),
            "experiment": experiment_prefix
        }
    except Exception as e:
        logger.error(f"Tutoring evaluation failed: {e}")
        return {"error": str(e), "tutoring_quality": 0.0}
```

The same pattern applies to flashcard and retrieval evaluations. The important detail is consuming the `ExperimentResults` iterator with `list()` and then extracting scores using `getattr()`, since `EvaluationResult` objects use attribute access, not dictionary access.

Evaluation API Endpoint

Finally, let's add an API endpoint so we can trigger evaluations and view results:

```
# In api/index.py
from .evaluation.run_baseline import (
    run_tutoring_evaluation,
    run_flashcard_evaluation,
    run_retrieval_evaluation
)

class EvaluationRequest(BaseModel):
    dataset_type: str  # "tutoring", "flashcard", "retrieval", "all"
    dataset_name: Optional[str] = None  # Name of LangSmith dataset to
evaluate against
    experiment_name: Optional[str] = None

class EvaluationResult(BaseModel):
    dataset_type: str
    experiment_name: str
    metrics: dict
    sample_size: int
    timestamp: str

@app.post("/api/evaluation/run", response_model=EvaluationResult)
def run_evaluation(request: EvaluationRequest):
    """Run evaluation on specified dataset."""
    from datetime import datetime
    experiment_name = request.experiment_name or f"eval-{datetime.now()
.strftime('%Y%m%d-%H%M')}"

    # Define wrapper functions that connect to our system
    def tutor_system(inputs: dict) -> dict:
        context = search_materials(inputs.get("question", ""), k=4)
        response = tutor_explain(tutor_llm, inputs.get("question", ""),
context, None, "")
        return {"response": response}

    def flashcard_system(inputs: dict) -> dict:
        context = search_materials(inputs.get("topic", ""), k=4)
        card = generate_single_card(card_generator_llm, inputs.get
("topic", ""), [], context)
        return {
```

```
            "question": card.get("question", "") if card else "",
            "answer": card.get("answer", "") if card else ""
        }

    def retrieval_system(inputs: dict) -> dict:
        context = search_materials(inputs.get("query", ""), k=5)
        chunks = context.split("\n\n---\n\n") if context else []
        return {"contexts": chunks}

    metrics = {}
    sample_size = 0

    if request.dataset_type in ["tutoring", "all"]:
        dataset = request.dataset_name or "tutoring-evaluation-v1"
        result = run_tutoring_evaluation(
            tutor_system,
            dataset_name=dataset,
            experiment_prefix=f"tutor-{experiment_name}"
        )
        metrics["tutoring_quality"] = result.get("tutoring_quality", 0)
        sample_size += result.get("sample_size", 0)

    # Similar patterns for flashcard and retrieval...

    return EvaluationResult(
        dataset_type=request.dataset_type,
        experiment_name=experiment_name,
        metrics=metrics,
        sample_size=sample_size,
        timestamp=datetime.now().isoformat()
    )

@app.post("/api/evaluation/generate-testset")
def generate_test_data(test_size: int = 100):
    """Generate synthetic test data from reference materials."""
    from .evaluation.testset_generator import (
        load_reference_documents,
        generate_tutoring_testset,
        validate_testset
    )
    from .evaluation.dataset_builder import
create_tutoring_evaluation_dataset
    from datetime import datetime

    # Load and chunk reference documents
    documents = load_reference_documents("documents")
    if not documents:
        raise HTTPException(status_code=404, detail="No reference
documents found")
```

```
    # Generate synthetic test cases using RAGAS
    testset = generate_tutoring_testset(documents, test_size)
    testset = validate_testset(testset)  # Filter low-quality cases

    # Create LangSmith dataset with timestamp to avoid conflicts
    dataset_name = f"tutoring-eval-{datetime.now().strftime('%Y%m%d-
%H%M%S')}"
    result = create_tutoring_evaluation_dataset(testset, dataset_name)

    if not result:
        raise HTTPException(status_code=500, detail="Failed to create
dataset")

    return {
        "status": "success",
        "test_cases_generated": len(testset),
        "dataset_id": str(result["id"]),  # Convert UUID to string
        "dataset_name": result["name"]
    }
```

Evaluation Dashboard

To make evaluation results actionable, we need a way to visualize them. Here's an endpoint that provides dashboard data by aggregating results. This transforms raw metrics into something useful for iteration.

First, we define the response model and set up the main endpoint. It fetches recent experiments and calculates current scores from the most recent one:

```
# api/evaluation/dashboard.py
from langsmith import Client
from datetime import datetime, timedelta
from collections import defaultdict

client = Client()

class DashboardMetrics(BaseModel):
    current_scores: dict[str, float]
    score_trends: dict[str, list[dict]]
    weak_areas: list[dict]
    recent_experiments: list[dict]
    recommendations: list[str]

@app.get("/api/evaluation/dashboard", response_model=DashboardMetrics)
def get_evaluation_dashboard():
    """Get evaluation dashboard data."""
    # Fetch recent experiment results
    experiments = list(client.list_projects(
```

```
        reference_dataset_name="tutoring-evaluation-v1"
    ))[:10]

    # Calculate current scores (most recent experiment)
    latest = experiments[0] if experiments else None
    current_scores = get_experiment_scores(latest) if latest else {}

    # Build score trends over time
    score_trends = build_score_trends(experiments)

    # Identify weak areas by analyzing failure cases
    weak_areas = identify_weak_areas(latest) if latest else []

    # Generate recommendations based on metrics
    recommendations = generate_recommendations(current_scores, weak_
areas)

    return DashboardMetrics(
        current_scores=current_scores,
        score_trends=score_trends,
        weak_areas=weak_areas,
        recent_experiments=[
            {"name": e.name, "date": e.created_at.isoformat()}
            for e in experiments[:5]
        ],
        recommendations=recommendations
    )
```

The `identify_weak_areas()` function groups evaluation results by topic and finds areas where scores consistently fall below threshold. This tells us exactly which content needs improvement:

```
def identify_weak_areas(experiment) -> list[dict]:
    """Identify content areas with low scores."""
    weak_areas = []
    runs = client.list_runs(project_name=experiment.name)

    # Group scores by topic
    topic_scores = defaultdict(list)
    for run in runs:
        topic = run.inputs.get("topic", "unknown")
        score = run.feedback_stats.get("tutoring_quality", {}).get
("avg", 0)
        topic_scores[topic].append(score)

    # Find topics with consistently low scores
    for topic, scores in topic_scores.items():
        avg = sum(scores) / len(scores) if scores else 0
        if avg < 0.7:  # Below threshold
            weak_areas.append({
```

```
                "topic": topic,
                "average_score": avg,
                "sample_count": len(scores),
                "suggestion": f"Review and improve content coverage for
{topic}"
            })

    return sorted(weak_areas, key=lambda x: x["average_score"])
```

The `generate_recommendations()` function translates metric scores into actionable advice. Rather than just reporting numbers, it suggests specific next steps:

```
def generate_recommendations(scores: dict, weak_areas: list) ->
list[str]:
    """Generate actionable recommendations from evaluation data."""
    recommendations = []

    # Check each metric against thresholds
    if scores.get("faithfulness", 1) < 0.85:
        recommendations.append(
            "Faithfulness is below target. Consider strengthening
grounding "
            "instructions in your prompts or reducing temperature."
        )
    if scores.get("retrieval_precision", 1) < 0.75:
        recommendations.append(
            "Retrieval precision needs improvement. Chapter 9 covers
reranking "
            "and hybrid search techniques that can help."
        )
    if scores.get("tutoring_quality", 1) < 0.8:
        recommendations.append(
            "Tutoring quality has room for improvement. Review the weak
areas "
            "identified below and consider adding more examples to
prompts."
        )

    # Add recommendations based on weak areas
    if weak_areas:
        topics = [w["topic"] for w in weak_areas[:3]]
        recommendations.append(
            f"Content quality is low for: {', '.join(topics)}. "
            "Consider adding more reference materials for these topics."
        )

    if not recommendations:
        recommendations.append(
            "All metrics are meeting targets. Consider running
adversarial "
```

```
            "test cases to identify edge case failures."
        )

    return recommendations
```

This dashboard gives you visibility into system health at a glance. You can see current scores, track trends over time, identify specific weak spots, and get actionable recommendations for improvement. It transforms evaluation from a one-time activity into an ongoing practice.

Testing the Evaluation System

Finally, let's verify everything works correctly with unit tests:

```
# tests/test_evaluation.py
def test_testset_generation():
    """Verify we can generate synthetic test data."""
    from api.evaluation.testset_generator import (
        load_reference_documents,
        generate_tutoring_testset
    )
    docs = load_reference_documents("documents")
    assert len(docs) > 0

    # Generate small testset for speed
    testset = generate_tutoring_testset(docs, test_size=5)
    assert len(testset) == 5
    assert "question" in testset.columns
    assert "ground_truth" in testset.columns

def test_tutoring_evaluator():
    """Verify tutoring evaluator produces valid scores."""
    from api.evaluation.evaluators import tutoring_quality_evaluator

    # Mock run and example that mimic LangSmith structures
    class MockRun:
        outputs = {"response": "Machine learning is a way for computers
to learn from data..."}

    class MockExample:
        inputs = {"question": "What is machine learning?"}
        outputs = {"reference_answer": "Machine learning is a subset
of AI..."}

    result = tutoring_quality_evaluator(MockRun(), MockExample())
    assert "score" in result
    assert 0 <= result["score"] <= 1
    assert result["key"] == "tutoring_quality"
```

```
def test_baseline_evaluation_runs():
    """Verify baseline evaluation completes without errors."""
    from api.evaluation.run_baseline import run_baseline_evaluation

    # Define mock functions for testing
    def mock_tutor(inputs):
        return {"response": "Mock tutoring response."}

    def mock_flashcard(inputs):
        return {"question": "What is X?", "answer": "X is Y."}

    def mock_retrieval(inputs):
        return {"contexts": ["Mock context"]}

    # This is an integration test - requires LangSmith setup
    results = run_baseline_evaluation(mock_tutor, mock_flashcard, mock_
retrieval)
    assert "tutoring" in results
    assert "flashcards" in results
    assert "retrieval" in results
```

Running Locally

StudyBuddy v8 runs with two terminals—one for the backend and one for the frontend.

Terminal 1—Backend:

```
cd v8-evaluation
uv sync
uv run uvicorn api.index:app --reload --port 8000
```

Terminal 2—Frontend:

```
cd v8-evaluation/frontend
npm run dev
```

Visit `http://localhost:3000`. The Next.js dev server proxies `/api/*` requests to the FastAPI backend on port 8000 (configured in `next.config.ts`).

Using the Evaluation Infrastructure

With all the components in place, here's how to use the evaluation system in practice. Assuming you have already installed the new dependencies from the previous section, you also need to ensure your environment variables are set. Your `.env` file needs:

```
OPENAI_API_KEY=sk-...
LANGSMITH_API_KEY=lsv2_...
```

Step 1: Generate Synthetic Test Data

The first step is generating test cases from your reference documents:

```
curl -X POST http://localhost:8000/api/evaluation/generate-testset \
  -H "Content-Type: application/json" \
  -d '{"test_size": 5}'
```

This takes a few minutes and creates a `tutoring-eval-YYYYMMDD-HHMMSS` dataset in LangSmith.

Step 2: Run the Evaluation

Once you have test data, run the evaluation against your system:

```
# Evaluate all components
curl -X POST http://localhost:8000/api/evaluation/run \
  -H "Content-Type: application/json" \
  -d '{"dataset_type": "all", "dataset_name": "tutoring-eval-YYYYMMDD-
HHMMSS"}'

# Or evaluate specific components
curl -X POST http://localhost:8000/api/evaluation/run \
  -H "Content-Type: application/json" \
  -d '{"dataset_type": "tutoring", "dataset_name": "tutoring-eval-
YYYYMMDD-HHMMSS"}'
```

Either way, replace `tutoring-eval-YYYYMMDD-HHMMSS` with the actual dataset name from the `generate-testset` response or your LangSmith dashboard.

These commands return metrics that look like this:

```
{
  "metrics": {
    "tutoring_quality": 0.82,
    "flashcard_quality": 0.78,
    "retrieval_precision": 0.85
  },
  "sample_size": 50
}
```

Step 3: View the Dashboard

Get aggregated metrics, trends, and recommendations:

```
curl http://localhost:8000/api/evaluation/dashboard
```

The dashboard returns current scores, historical trends across experiments, topics with low scores (weak areas), and actionable recommendations for improvement.

Viewing the Results in LangSmith

Results are also tracked in LangSmith at `https://smith.langchain.com`. Navigate to your project (`studybuddy-v8`), view experiments by prefix (like `tutor-baseline-*`), and compare runs to see how changes affect quality over time.

Important Notes

The evaluation endpoints are disabled on Vercel because they're too compute-heavy for serverless environments. Run evaluations locally during development. Generate test sets once, then reuse them for multiple evaluation runs to save on API costs.

What's Next

In Chapter 9, we'll use this infrastructure to actually improve things. We'll add advanced retrieval techniques like hybrid search, reranking, and semantic chunking, and measure whether each change actually helps. The evaluation system we've built is what makes principled optimization possible. You've got the tools to measure what matters. Now let's put them to work.

Let's keep building!

CHAPTER

9

Advanced Retrieval

Basic vector search is impressive—until it isn't. It handles synonyms and paraphrasing beautifully, but ask it about a specific error code and you'll get generic results. Search with casual language when your docs are written in engineering-speak, and you'll miss exactly what you need. As your application matures, those blind spots go from minor annoyances to real problems.

This chapter is about fixing retrieval. We're going to layer multiple search strategies on top of basic vector search. Each technique addresses a specific weakness, and you don't need all of them for every application. The goal is to know when each one helps and why.

By the end of this chapter, you'll have a full toolkit of advanced retrieval techniques and a clear framework for deciding which ones your application actually needs. And because we built evaluation infrastructure in Chapter 8, you'll be able to prove—with actual numbers—that your upgrades were worth it.

Beyond Basic Vector Search

Dense vector retrieval is genuinely impressive. You embed a query, find the nearest vectors in your database, and retrieve semantically similar content. It handles common synonyms and paraphrasing beautifully. Someone can ask about "machine learning algorithms" and retrieve documents that discuss "ML techniques" or "statistical learning methods." Ask about something being

"broken" and it'll find documents that say "malfunctioning" just fine. The semantic understanding is real, and for many applications, basic vector search is all you need.

But dense retrieval has blind spots that become more significant as your application matures. The first is *exact match failure*. Dense search captures the general neighborhood of meaning, but it loses specificity for identifiers, codes, and proper nouns. Ask about "error code E-4012" and dense search might return generic content about error handling rather than the specific documentation for that exact code. The embedding model understands "error codes" as a concept but doesn't treat "E-4012" as a unique identifier that must match precisely.

The second blind spot is what you might call *cross-register vocabulary mismatch*. Dense search handles straightforward synonyms such as "broken" and "malfunctioning" because embedding models see those words used interchangeably during training. The real problem emerges when user language and document language come from different registers or domains. A user might type "my app crashes when I hit save" while the documentation discusses "unhandled exceptions in the persistence layer." Those describe the same problem, but they live in completely different regions of embedding space. One is casual end-user language; the other is formal engineering terminology. Similarly, someone searching for "make it faster" won't reliably find documents about "query optimization," "connection pooling," or "cache invalidation strategies" because the user's vague intent doesn't map cleanly to any of those specific technical solutions. The embedding model learned synonyms, but it didn't learn that a frustrated user saying "slow" might need documentation about database indexing.

Another subtle issue is the curse of *high-dimensional similarity*. In embedding space, many documents might be roughly equidistant from your query. The differences between your top 5 results and results 6 through 10 might be tiny—essentially noise. This means small changes in query phrasing can dramatically change which documents make the cut, leading to inconsistent user experiences.

The good news is that we have techniques to address each of these limitations. The bad news is that every technique adds complexity and latency. The goal is to add just enough sophistication to meet your quality requirements, and no more.

Here's a decision framework for when to upgrade your retrieval:

- **Hybrid search** makes sense when your queries mix conceptual questions with specific terms. Technical documentation, e-commerce catalogs, legal databases, medical records—anywhere users might search for exact identifiers alongside natural language questions. If users ever search for codes, IDs, names, or technical terms that must match exactly, hybrid search helps.
- **Reranking** pays off when precision matters more than latency. If retrieving the wrong document has real consequences—think: legal advice, medical

information, financial decisions—the extra 100–200 ms for reranking is worth it. Reranking also helps when your initial retrieval is noisy and you need to surface the best results.

- **Query expansion** helps when vocabulary mismatch is a known problem. If your user research shows people searching with different terms than your content uses, or if you see a lot of zero-result searches, expansion bridges that gap. It's also useful when queries are ambiguous and could benefit from exploring multiple interpretations.
- **RAG-Fusion** shines when comprehensive coverage matters. Research queries, complex questions that touch multiple topics, situations where missing relevant content is worse than including some irrelevant content—these are cases in which RAG-Fusion can help. If your users need exhaustive results rather than quick answers, RAG-Fusion can be useful.
- **Semantic chunking** helps when your content has clear structure that fixed-size chunking destroys. This includes technical documentation with code examples, academic papers with methodology sections, legal documents with numbered clauses, or anywhere that cutting content at arbitrary positions loses important context.

Let's dig into each technique.

Hybrid Search

Hybrid search combines dense vector retrieval with sparse keyword matching. Dense search finds semantically similar content. Sparse search (typically Best Matching 25, or BM25) finds exact term matches. Together, they cover each other's weaknesses in a way that neither can achieve alone.

BM25 is a classic information retrieval algorithm that's been powering search engines since the 1990s. It scores documents based on two complementary signals: *term frequency* and *inverse document frequency*. If your query contains the term "E-4012" and a document contains that exact term, BM25 will score it highly. It doesn't care about semantics; it cares about matching terms. That's exactly what you want for specific identifiers, names, and technical terms.

The algorithm gives higher scores to documents where query terms appear frequently, but it also considers how rare those terms are across the entire corpus. A term that appears in every document isn't very useful for distinguishing relevant results. A term that appears in only a few documents is a stronger signal of relevance. BM25 also includes a document-length normalization factor, which prevents longer documents from automatically scoring higher simply because they contain more words in total. This balance of frequency, rarity, and length makes BM25 remarkably effective for keyword-oriented queries, even decades after its invention.

In LangChain, BM25 retrieval is available through the `BM25Retriever` class from `langchain_community.retrievers`. You initialize it with your document collection using the `from_documents()` class method, set the number of results you want via the `k` property, and invoke it with a query string. The retriever returns documents ranked by BM25 score, with exact term matches floating to the top.

The challenge with hybrid search is combining results from two completely different scoring systems. Dense search might return documents with cosine similarity scores between 0 and 1. BM25 returns scores on a completely different scale that depends on document length and corpus statistics. You can't just average them, because a BM25 score of 15 isn't comparable to a cosine similarity of 0.85. To solve this problem, you need Reciprocal Rank Fusion.

NOTE **You can find all of this chapter's non-StudyBuddy code in the Jupyter notebook `chapter09.ipynb`, available in this book's GitHub repo.**

Reciprocal Rank Fusion

Reciprocal Rank Fusion (RRF) solves the scoring system conundrum elegantly. Instead of combining the incompatible raw scores from dense and sparse searches, it combines *rankings*. The intuition is simple: if a document appears highly ranked in multiple result lists, it's probably relevant. The actual score doesn't matter; only the position in each ranked list does.

RRF assigns each document a score based on its rank in each list using the formula `1 / (k + rank)`, where `k` is a constant (typically 60) that prevents the top-ranked document from dominating too heavily. Documents that appear in multiple lists accumulate scores from each appearance. The final ranking is based on these accumulated scores.

The `k` parameter controls how much rank position matters. Higher values flatten the differences between ranks: position 1 and position 5 become more similar in their contribution. Lower values amplify rank differences, making the top-ranked document much more valuable than lower-ranked ones. The default of 60 works well for most applications, but you can tune it based on your specific needs.

Implementing RRF is straightforward. You iterate through each result list, and for each document at each rank, you add its RRF contribution to a running score dictionary. Documents are identified by a content hash so that duplicates appearing in multiple lists accumulate properly. After processing all lists, you sort by accumulated score and return the top results. The entire function is typically under 20 lines of Python.

You can also use a weighted variant of RRF where each result list gets a different weight. This lets you express a preference. For example, maybe you

trust dense search slightly more than BM25, so you give it a weight of 0.6 versus 0.4. The weighted score becomes the weight times the RRF contribution, giving you fine-grained control over how much each search modality influences the final ranking.

Here's the core RRF function:

```
def reciprocal_rank_fusion(
    result_lists: list[list],
    k: int = 60
) -> list:
    scores = {}
    for results in result_lists:
        for rank, (doc, _) in enumerate(results):
            doc_id = hash(doc.page_content)
            if doc_id not in scores:
                scores[doc_id] = {"doc": doc, "score": 0}
            scores[doc_id]["score"] += 1 / (k + rank + 1)
    sorted_results = sorted(
        scores.values(),
        key=lambda x: x["score"],
        reverse=True
    )
    return [item["doc"] for item in sorted_results]
```

Pass this function a list of ranked result lists—where each is a list of `(doc, score)` tuples—and it returns documents sorted by their fused rank. Note that the actual scores get discarded because only position matters.

Hybrid Search in Practice

A complete hybrid search implementation brings both approaches together behind a single interface. The typical pattern is to create a class that holds both a vector store (such as Qdrant) for dense search and a BM25 retriever for sparse search. When a query comes in, you run both searches, retrieve `k * 2` candidates from each to ensure good coverage, and then fuse the results using RRF. The `similarity_search_with_score()` method on the vector store returns document-score pairs, while the BM25 retriever returns ranked documents that you convert to document-score pairs based on rank position.

Consider an e-commerce product search scenario. A customer searches for "blue Nike running shoes size 10." Dense search understands "running shoes" conceptually and might return various athletic footwear: trail runners, cross-trainers, and walking shoes. But BM25 catches "Nike" as an exact brand match and "size 10" as a specific attribute. Hybrid search gives you both: semantic understanding of the product category plus exact matching on brand and size. The customer gets Nike running shoes in their size, not generic athletic footwear.

The same pattern applies to technical documentation. Someone searching for "ConnectionTimeout exception in DatabaseClient" needs both the semantic understanding of "exception handling" concepts and the exact matching of "ConnectionTimeout" and "DatabaseClient" as specific class names. Dense search alone might return general exception handling documentation. BM25 alone might miss conceptually related content. Together, they find the precise documentation for that specific error case.

Reranking

Initial retrieval casts a wide net. You ask for 20 candidates because you want to make sure you don't miss anything relevant. But those 20 results aren't perfectly ordered. The document at position 8 might be more relevant than the one at position 2. Initial retrieval optimizes for recall, meaning that it gets relevant documents into the candidate set. But you also need precision, meaning that the best documents appear at the top.

Reranking is a second pass that reorders your initial results using a more powerful model. It's a two-stage architecture: the first stage retrieves candidates quickly, and the second stage ranks them precisely. This pattern appears throughout information retrieval, from web search to recommendation systems.

Bi-Encoders vs. Cross-Encoders

The key difference is in how the models process input. Your embedding model (a bi-encoder) processes the query and each document independently. The query gets embedded once. Each document gets embedded once, typically during indexing. At search time, you're just doing vector similarity math, which is extremely fast. But processing the query and documents independently means the model can't attend to fine-grained interactions between them.

A cross-encoder, by contrast, processes the query and document as a single concatenated input. The model's attention mechanism can look at both simultaneously, finding subtle relevance signals that only emerge when you consider them together. "How do I fix error E-4012?" and a document that mentions "Error E-4012 occurs when. . ." have an obvious lexical match, but the cross-encoder can also understand that "fix" relates to "occurs when" as a problem-and-solution pattern. This joint processing is more accurate but slower: you can't precompute anything because the score depends on the specific query.

That's why you use cross-encoders for reranking a small candidate set, typically 10 to 50 documents, rather than searching your entire corpus. Reranking 20 candidates with a cross-encoder takes 100–300 milliseconds. Reranking a million documents would take hours.

Using Cohere for Reranking

Cohere's reranking API is the easiest way to add this capability to a production system. You send the query and an array of candidate document texts to the `co.rerank()` method, specifying the model (such as `rerank-english-v3.0`) and how many top results you want back. The API returns reranked results with relevance scores, so you can see exactly how confident the model is in each ranking. Each result includes the index of the original document in your candidate list and a relevance score between 0 and 1.

The implementation pattern is clean:

1. Perform a broad initial retrieval. Fetch 20 candidates when you only need 5 final results.
2. Pass those candidates to Cohere's reranking endpoint.
3. Map the reranked indices back to the original document objects and return the top results.

Here's a stripped-down version of a two-stage retrieval function:

```
def search_with_reranking(
    query: str,
    vector_store,
    k: int = 5,
    initial_k: int = 20
) -> list:

    # Stage 1: Broad retrieval
    candidates = vector_store.similarity_search(query, k=initial_k)
    if not candidates:
        return []

    # Check for Cohere availability (not shown)

    # Stage 2: Rerank with Cohere
    import cohere
    co = cohere.Client(api_key=api_key)
    rerank_response = co.rerank(
        query=query,
        documents=[doc.page_content for doc in candidates],
        top_n=k,
        model="rerank-english-v3.0"
    )
    # Return reranked documents
    reranked = []
    for result in rerank_response.results:
        doc = candidates[result.index]
        doc.metadata["rerank_score"] = result.relevance_score
        reranked.append(doc)
    return reranked
```

The first stage retrieves broadly for recall, while the second stage reranks with Cohere for precision. Everything else is implementation detail on top of this basic pattern. The whole process adds about 100–250 milliseconds to your retrieval pipeline, which is noticeable but acceptable for most interactive applications.

Alternatives to Cohere include cross-encoder models from Hugging Face that you can run locally. Models like the MS MARCO MiniLM cross-encoders are popular choices. Running locally gives you more control and eliminates API costs, but you'll need to manage GPU infrastructure for reasonable latency. On CPU, cross-encoder inference can be significantly slower than calling an API.

When to Rerank

When does reranking make sense? The clearest case is when precision matters more than latency. Some examples:

- Legal document retrieval, where surfacing the wrong clause could have real consequences
- Medical information systems, where accuracy is paramount
- Customer support, where the first result needs to actually answer the question to avoid frustrating back-and-forth

Reranking also helps when your initial retrieval is noisy. If you're retrieving from a large, diverse corpus where many documents are somewhat relevant, reranking separates the truly relevant from the tangentially related. It's less valuable when your initial retrieval is already highly precise. If your top 5 results are always great, spending 200 milliseconds to confirm that doesn't add much value.

For real-time systems where every millisecond counts, you might skip reranking or use it selectively for complex queries. For batch processing or asynchronous workflows, the latency doesn't matter at all. The key is to measure the precision improvement against the latency cost for your specific use case.

Query Expansion

Users don't always use the same words as your documents. Someone might search for "laptop won't start" when your documentation says "notebook fails to boot." They might ask about "slow performance" when your content discusses "latency issues" or "response time degradation." This vocabulary mismatch is one of the most common causes of retrieval failures, and it's frustrating because the right information exists; your app just can't find it with the user's phrasing.

Query expansion addresses this by generating multiple query variations that cover different phrasings of the same intent. You search with all variations, then

combine the results. If one phrasing misses the relevant document, another may find it.

LLM-Based Expansion

The most flexible approach uses an LLM to generate query variations. You pass the original query to the model with a prompt asking for alternative phrasings that use synonyms, vary in specificity, or approach the topic from different angles. The LLM returns a JSON array of variations, which you parse and combine with the original query. Searching with each variation and deduplicating by content hash ensures that you don't return the same document multiple times.

A typical expansion function takes the original query and an LLM instance, prompts the model for three variations (using synonyms, more specific phrasing, and more general phrasing), then returns the original query plus whatever the model generated. The search function then iterates through all queries, runs each one against the vector store, tracks seen documents via content hashes, and returns the top `k` unique results.

In practice, an expansion function is really just a prompt wrapper:

```
def expand_query(query: str, llm) -> list[str]:
    prompt = f"""Generate 3 alternative search queries for:

Original: {query}

Create variations using:
1. Synonyms and related terms
2. More specific phrasing
3. More general phrasing

Return as a JSON array of strings. Only the array, no explanation."""

    response = llm.invoke(prompt)
    content = extract_json(response.content)

    # Always include original query
    return [query] + variations
```

Notice that the original query is always included in the return list. This is important because you don't want to replace the user's phrasing, only augment it.

Risks and Alternatives

The risk with LLM-based query expansion is query drift. Your variations might stray from the original intent. If someone asks about "Python exceptions" and your expansion includes "snake handling problems," you've drifted into

irrelevant territory. The fix is careful prompt engineering—be explicit that variations should preserve the original meaning—and optionally a filtering step that checks whether generated variations are semantically similar enough to the original query before executing them.

A simpler approach that doesn't require an LLM is synonym expansion using a thesaurus or domain-specific term mappings. For technical domains, you might maintain a dictionary of common synonyms: "error" maps to "exception," "bug," "issue," "problem," and "fault." This is faster, more predictable, and cheaper than LLM-based expansion, though less flexible. You can also combine approaches: use static synonym expansion for common terms and LLM expansion for complex queries where static mappings aren't enough.

RAG-Fusion

RAG-Fusion takes query expansion further by systematically generating diverse queries and combining their results with Reciprocal Rank Fusion. It was introduced in a 2023 article by Adrian H. Raudaschl titled "Forget RAG, the Future is RAG Fusion," which described an approach intended to improve retrieval quality for complex queries. The key insight is that different query phrasings can activate different regions of the embedding space. By searching with multiple phrasings and combining results, you cover more ground than any single query could.

The implementation follows these general steps:

1. Use an LLM to generate several diverse search queries from the original question, prompting for different angles on the topic, varying levels of specificity, and alternative phrasings. Include the original query alongside the generated variations.
2. Search with each query independently, collecting ranked results from each search.
3. Combine all result lists using RRF to produce a single fused ranking.

RAG-Fusion is especially useful for research-style queries where you want comprehensive coverage. Someone researching "best practices for microservices architecture" benefits from queries that cover design patterns, deployment strategies, communication protocols, failure handling, and monitoring. Each angle might surface different relevant documents that a single query would miss.

The cost is more LLM calls (for query generation) and more vector searches (one per generated query). For a simple factual question such as "What year was Python released?" RAG-Fusion is overkill because you're spending extra time and money to find the same answer multiple ways. For complex, multifaceted

questions, the extra coverage is worth the cost. The typical configuration generates 3–4 additional queries, resulting in 4–5 total searches. At roughly 20–50 milliseconds per vector search plus the LLM call for query generation, the total overhead is usually 200–500 milliseconds.

Here's a basic RAG-Fusion function:

```
def rag_fusion(
    query: str,
    vector_store,
    llm,
    k: int = 5,
    num_queries: int = 4
) -> list:
    prompt = f"""Generate {num_queries} different search queries to find
information for:
\"{query}\"

Make queries diverse:
- Different angles on the topic
- Varying levels of specificity
- Alternative phrasings and synonyms

Return as a JSON array of strings."""

    response = llm.invoke(prompt)
    content = extract_json(response.content)

    # Include original
    queries = [query] + queries

    # Search with each query
    all_ranked_results = []
    for q in queries:
        results = vector_store.similarity_search_with_score(q, k=k*2)
        all_ranked_results.append(results)

    # Combine with RRF
    combined = reciprocal_rank_fusion(all_ranked_results)

    return combined[:k]
```

The `rag_fusion()` function takes the original query, a vector store, and an LLM. It prompts the LLM to generate `num_queries` diverse alternatives (defaulting to 4), runs a similarity search for each query, including the original, feeds all the ranked result lists through the `reciprocal_rank_fusion()` function (shown earlier), and returns the top `k` results.

Semantic Chunking

Every retrieval technique we've discussed operates on chunks—pieces of documents that were split during indexing. If your chunks are poorly constructed, no amount of fancy retrieval will save you. Garbage in, garbage out.

Fixed-size chunking is simple and predictable. Split every 500 characters, add some overlap, done. But it completely ignores document structure. A chunk might start mid-sentence, split a code example in half, or separate a concept from its explanation. When your retrieval returns half a code snippet, the LLM can't help the user fix their bug.

Embedding-Based Boundary Detection

Semantic chunking uses embeddings to identify natural topic boundaries. The algorithm embeds each sentence or small paragraph, then looks for points where consecutive sentences have significantly different embeddings, indicating a topic shift. Chunks are created at these natural boundaries rather than at arbitrary character positions. In LangChain, this is available through the `SemanticChunker` class from the `langchain_experimental` package.

You initialize the chunker with an embedding model and configure how it detects breakpoints. The `breakpoint_threshold_type` parameter controls the detection method: `"percentile"` is the most common, which treats any similarity gap above a given percentile as a topic boundary. The `breakpoint_threshold_amount` sets the sensitivity. A higher threshold like 95 only splits at dramatic topic changes, producing fewer, larger chunks. A lower threshold like 80 splits more aggressively, producing more, smaller chunks.

The right threshold depends on your content and retrieval needs. Technical documentation with clear section boundaries might work well with aggressive splitting, since each section is likely self-contained. Narrative content with smooth transitions between ideas might need conservative splitting to avoid fragmenting coherent arguments. Experimentation is key: try different thresholds and measure retrieval quality with your evaluation dataset.

Structure-Aware Chunking

For structured documents such as Markdown or HTML, you can take an even more direct approach. Structure-aware chunking respects headers and sections explicitly, splitting at structural boundaries rather than relying on embedding similarity to detect them. LangChain's `MarkdownHeaderTextSplitter` splits Markdown documents at header boundaries, preserving the header hierarchy in each chunk's metadata. You configure which heading levels to split on, and the splitter produces chunks that correspond to document sections.

The best chunking strategy depends on your content. Technical documentation benefits from structure-aware splitting that keeps code examples intact with their explanations. Narrative content such as articles or books works well with semantic chunking that follows the flow of ideas. Mixed content might need a hybrid approach: split by structure first, then apply semantic chunking to any sections that are still too large. This two-pass strategy gives you the structural preservation of header-based splitting and the granularity of semantic splitting for long sections.

Compression and Filtering

A common problem is that your system has retrieved the right documents, but they contain more content than the LLM needs. The techniques of contextual compression and relevance filtering tackle this problem from two different angles.

Contextual Compression

Sometimes you retrieve the right document, but it's too long. A 2,000-token chunk might contain only 200 tokens of actually relevant content. Including the whole thing wastes context window space and can dilute the signal with noise. The LLM has to wade through irrelevant content to find the useful parts.

Contextual compression extracts only the relevant portions of retrieved documents. Given the query and a retrieved document, it returns just the parts that help answer the question. Everything else gets filtered out. In LangChain, the `ContextualCompressionRetriever` wraps any base retriever and applies a compressor to each result. The `LLMChainExtractor` compressor uses an LLM to identify and extract the relevant portions of each document, returning trimmed versions that focus on the query at hand.

The trade-off is additional LLM calls. Every retrieved document goes through the compressor, adding latency and cost. For applications where context window space is precious or precision matters more than speed, compression is valuable. With modern LLMs that have 128K or larger context windows, the pressure to compress is lower than it used to be. But even with large context windows, feeding cleaner, more focused context to the model generally produces better answers than flooding it with partially relevant content.

Relevance Filtering as a Lightweight Alternative

A lighter-weight alternative is relevance filtering without full compression. The idea is simple: compute the cosine similarity between the query embedding and each retrieved document's embedding, then drop any document below a

similarity threshold. Instead of extracting relevant passages, you're making a binary decision about whether each document is worth including at all.

This approach is faster than full LLM compression because it only requires embedding computations, not LLM inference. The threshold parameter requires tuning for your specific embedding model and domain. With `text-embedding-3-small`, cosine similarities for related content tend to fall in the 0.2–0.6 range, so a threshold of 0.3 is a reasonable starting point. The filtering removes clearly irrelevant documents but can't extract the relevant portions from partially relevant documents, which is where full compression has the advantage.

Hierarchical Retrieval

When you're searching a large corpus, flat retrieval can be inefficient. Searching millions of chunks to find five relevant ones means most of your compute is wasted evaluating clearly irrelevant content.

Hierarchical retrieval uses document summaries as a first-stage filter. You search the summaries to identify promising documents, then search within those documents for specific chunks. It's like using a table of contents before reading individual chapters, because it makes no sense to read every page of a 500-page book just to find the section you need.

The implementation requires maintaining two separate indexes: one for document summaries and one for document chunks. When a query arrives, you first search the summary index to identify the top three to five most relevant documents. Then you search the chunk index, but filtered to only include chunks from those identified documents. This dramatically reduces the search space while maintaining high recall, because the summary stage acts as a coarse filter that eliminates entire documents that aren't relevant.

Summaries can be generated automatically using an LLM, extracted from document abstracts or introductions if available, or created manually for high-value documents. The summary quality matters because it determines whether the first stage correctly identifies relevant documents. A poor summary might cause the system to miss an important document entirely, and no amount of precise chunk-level search can recover from that.

This pattern pays off for large collections where the first stage dramatically reduces the search space. If you have 100,000 documents and summaries help you narrow the set to 10 candidates, you've reduced the number of documents considered in the detailed search by a factor of 10,000. For smaller collections where searching all chunks is already fast, the overhead of maintaining two indexes might not be worth the complexity.

Systematic Comparison of Retrievers

With all these techniques available, how do you decide what to use? The answer is measurement. Build a comparison harness, run your evaluation dataset through different retriever configurations, and let the numbers guide your decisions.

However, before you run any comparisons, it will help to have a high-level map of what you're comparing. Table 9.1 summarizes the techniques from this chapter across the dimensions that matter most for decision-making.

Table 9.1: Advanced retrieval techniques at a glance

Technique	Latency Added	Best For	Main Trade-Off
Hybrid search	Low (40–80 ms)	Queries with specific terms, codes, identifiers	Requires maintaining two indexes
Reranking	Medium (100–250 ms)	Precision-critical applications	API cost and latency per query
Query expansion	Medium (100–250 ms)	Vocabulary mismatch between users and documents	Risk of query drift
RAG-Fusion	High (200–500 ms)	Complex, multifaceted research queries	Multiple LLM calls and vector searches
Semantic chunking	None	Structured documents, code examples	Slower indexing, harder to tune
Contextual compression	High (200–500 ms per document)	Smaller context window, long chunks	LLM call for each retrieved document
Relevance filtering	Low (50–100 ms)	Quick noise removal, high-confidence corpus	Coarser than compression and might drop useful documents
Hierarchical retrieval	Low (two fast searches)	Very large corpora (100,000 documents or more)	Two indexes to maintain and sync

Controlling Variables

The key to good comparisons is controlling variables. When you test hybrid search against basic vector search, everything else should stay the same: same

documents, same chunking, same embedding model, same number of retrieved results. Otherwise, you won't know what caused the improvement. This seems obvious, but it's easy to accidentally change multiple things at once when you're iterating quickly.

A comparison harness doesn't need to be complicated. Define each retriever configuration with a name, a retrieval function, and a description. Prepare a set of test queries with known ground truth answers. Then loop through each configuration, running every test query, timing the retrieval, and scoring the results against ground truth. Aggregate the scores to get mean quality, mean latency, and P95 latency for each configuration. That's the core loop. You can wrap it in a `RetrieverConfig` dataclass and a `compare_retrievers()` function for reusability.

Segment Analysis

Beyond the basic comparison, you want to understand performance across different query types. Some retrievers might excel at short factual queries but struggle with complex multipart questions. Others might have the opposite profile. Segmenting your test set by query characteristics—simple, moderate, and complex based on word count or the presence of comparison keywords—reveals these patterns. A retriever that looks mediocre overall might actually be the best choice for your specific query distribution.

For rigorous evaluation, use RAGAS metrics rather than simple keyword matching. RAGAS's `context_precision` and `context_recall` metrics tell you not just whether keywords match, but whether the retrieved context supports answering the question correctly. Context precision measures what fraction of retrieved documents are relevant, while context recall measures what fraction of relevant information was successfully retrieved. Together, they give you a complete picture of retrieval quality.

The Decision Framework

When comparing retrievers, consider multiple dimensions beyond just quality scores: quality metrics like context precision and recall, latency at both the mean and P95, cost in terms of API calls and tokens processed, and implementation complexity, including maintenance burden and ease of debugging.

The "best" retriever depends on your priorities. A financial services application might accept higher latency for better precision. A consumer search application might prioritize speed over marginal quality gains. A startup with limited budget might favor simpler approaches that don't require expensive reranking API calls.

Here's a practical framework for making the decision:

- Start simple: basic vector search with good chunking can take you surprisingly far.
- Add hybrid search if you have specific identifiers, codes, or names that users search for.
- Add reranking if precision is critical and you can tolerate approximately 200 milliseconds of extra latency.
- Add query expansion if user research shows that vocabulary mismatch is causing failures.
- Add RAG-Fusion if comprehensive coverage matters more than speed.

The key is to measure before and after each change. Don't assume that more sophisticated means better. Sometimes the simple approach wins, and you've saved yourself the complexity.

Building StudyBuddy v9

Let's put all of this into practice. In Chapter 8, we built evaluation infrastructure with synthetic test datasets and knowledge graph coverage. Now we're going to upgrade the retrieval system and prove—with actual numbers—that the upgrades work.

Where We Left Off

StudyBuddy v8 has comprehensive evaluation datasets generated from the knowledge graph approach. We have question-answer pairs covering the full curriculum, flashcard quality validation sets, and edge cases at varying difficulty levels. The LangSmith pipelines are ready to run evaluations at scale.

But the retrieval system is still basic vector search. We're using fixed-size chunks, single-query retrieval, and no reranking. Now we'll improve it.

What We're Adding

The v9 retrieval pipeline combines four major upgrades:

- **Semantic chunking** that respects section boundaries in the reference materials
- **Hybrid search** combining dense vectors with BM25 keyword matching

- **Cohere reranking** for precision in the final results
- **RAG-Fusion** for complex queries that benefit from multiple search angles

Semantic Chunking for Reference Materials

The reference materials are structured Markdown documents with clear section headers. Fixed-size chunking splits them arbitrarily, so a code example might get cut in half or an explanation might be separated from its context. Semantic chunking respects the structure, and each header becomes a boundary where we won't split:

```
# In api/services/indexing.py

from langchain_experimental.text_splitter import SemanticChunker
from langchain_text_splitters import MarkdownHeaderTextSplitter
from langchain_openai import OpenAIEmbeddings

def chunk_reference_document(content: str, source: str) -> list:
    """
    Chunk reference documents using a hybrid approach:
    1. Split by markdown headers first
    2. Apply semantic chunking within large sections
    """
    # First pass: split by headers
    header_splitter = MarkdownHeaderTextSplitter(
        headers_to_split_on=[
            ("#", "h1"),
            ("##", "h2"),
            ("###", "h3"),
        ]
    )

    header_chunks = header_splitter.split_text(content)

Next, apply semantic chunking to sections that are too large. Sections
over 1500 characters get split again at natural topic boundaries
detected by embedding similarity.

    # Second pass: semantic chunking for large sections
    embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
    semantic_chunker = SemanticChunker(
        embeddings=embeddings,
        breakpoint_threshold_type="percentile",
        breakpoint_threshold_amount=90
    )
```

Finally, preserve metadata and return the chunks. Each chunk carries its source so you can trace back to the original document:

```
    final_chunks = []
    for chunk in header_chunks:
        # If chunk is large, apply semantic chunking
        if len(chunk.page_content) > 1500:
            sub_chunks = semantic_chunker.create_documents(
                [chunk.page_content]
            )
            for sub in sub_chunks:
                sub.metadata.update(chunk.metadata)
                sub.metadata["source"] = source
            final_chunks.extend(sub_chunks)
        else:
            chunk.metadata["source"] = source
            final_chunks.append(chunk)

    return final_chunks
```

Hybrid Search Implementation

The hybrid retriever combines Qdrant's dense search with BM25 sparse matching. We need both a vector store for dense search and a collection of documents for BM25. The `weight` parameter lets us tune how much to trust dense versus sparse results (0.6 favors dense by default):

```
# In api/services/retrieval.py

from langchain_community.retrievers import BM25Retriever
from langchain_qdrant import QdrantVectorStore

class HybridRetriever:
    """
    Combines dense vector search with BM25 keyword matching.
    Uses Reciprocal Rank Fusion to combine results.
    """

    def __init__(
        self,
        vector_store: QdrantVectorStore,
        documents: list,
        dense_weight: float = 0.6
    ):
        self.vector_store = vector_store
        self.dense_weight = dense_weight
```

```
        self.bm25_retriever = BM25Retriever.from_documents(documents)
```

The main search method runs both queries and fuses results. We fetch k*2 candidates from each to ensure good coverage before fusion.

```
    def search(self, query: str, k: int = 5) -> list:
        """Perform hybrid search with RRF fusion."""
        # Dense search
        dense_results = (
            self.vector_store.similarity_search_with_score(
                query, k=k*2
            )
        )

        # Sparse search
        self.bm25_retriever.k = k * 2
        sparse_docs = self.bm25_retriever.invoke(query)
        sparse_results = [
            (doc, 1.0 / (i + 1))
            for i, doc in enumerate(sparse_docs)
        ]
```

The fusion step combines rankings from both searches. Documents appearing near the top in multiple lists accumulate higher scores:

```
        # RRF fusion
        combined = self._reciprocal_rank_fusion(
            [dense_results, sparse_results],
            weights=[self.dense_weight, 1 - self.dense_weight]
        )

        return combined[:k]
```

The RRF implementation iterates through each ranked list. For every document at each rank position, it adds a contribution to that document's running score. The formula `weight * (1 / (k + rank + 1))` means top-ranked documents contribute the most, but lower-ranked documents from multiple lists can still accumulate significant scores:

```
    def _reciprocal_rank_fusion(
        self,
        result_lists: list,
        weights: list,
        k: int = 60
    ) -> list:
        """Combine ranked lists using weighted RRF."""
```

```
    scores = {}

    for results, weight in zip(result_lists, weights):
        for rank, (doc, _) in enumerate(results):
            doc_id = hash(doc.page_content)

            if doc_id not in scores:
                scores[doc_id] = {
                    "doc": doc, "score": 0
                }

            scores[doc_id]["score"] += (
                weight * (1 / (k + rank + 1))
            )

    sorted_results = sorted(
        scores.values(),
        key=lambda x: x["score"],
        reverse=True
    )

    return [item["doc"] for item in sorted_results]
```

Cohere Reranking Integration

After hybrid search retrieves candidates, we rerank for precision. You need a Cohere API key for this. Create a free Cohere account and log in. On your Dashboard, click API Keys, then copy the default trial key. Add the following to the `.env` file in the `v9-optimized-retrieval` directory:

```
COHERE_API_KEY=your_trial_key
```

Now here's the rerank code:

```
# In api/services/retrieval.py

import cohere
import os

class RerankedRetriever:
    """Wraps a base retriever with Cohere reranking."""

    def __init__(
        self, base_retriever, rerank_top_n: int = 5
    ):
        self.base_retriever = base_retriever
        self.rerank_top_n = rerank_top_n
        # Gracefully handle missing API key
        api_key = os.getenv("COHERE_API_KEY")
```

```
        if api_key and api_key != "your_cohere_api_key_here":
            self.cohere_client = cohere.Client(
                api_key=api_key
            )
        else:
            self.cohere_client = None
```

The search method fetches candidates, then reranks them. We fetch four times the requested results to give the reranker more options. The base retriever's order doesn't matter. What matters is getting good candidates into the pool:

```
    def search(self, query: str, k: int = 5) -> list:
        """Search with base retriever, then rerank."""
        # Get candidates from base retriever
        candidates = self.base_retriever.search(
            query, k=k*4
        )

        if not candidates:
            return []

        # If no Cohere client, return base results
        if self.cohere_client is None:
            return candidates[:k]
```

```
Call Cohere's rerank endpoint with the query and candidate documents.
The response includes relevance scores we attach to each document's
metadata.
```

```
        try:
            # Rerank with Cohere
            rerank_response = self.cohere_client.rerank(
                query=query,
                documents=[
                    doc.page_content for doc in candidates
                ],
                top_n=min(
                    self.rerank_top_n, len(candidates)
                ),
                model="rerank-english-v3.0"
            )

            # Return reranked documents
            reranked = []
            for result in rerank_response.results:
                doc = candidates[result.index]
                doc.metadata["rerank_score"] = (
                    result.relevance_score
                )
                reranked.append(doc)
```

```
            return reranked[:k]

        except Exception as e:
            print(f"Cohere reranking failed: {e}")
            return candidates[:k]
```

RAG-Fusion for Complex Queries

For tutoring questions that might have multiple angles, we use RAG-Fusion. This retriever generates multiple phrasings of the user's question, searches with each variation, then combines the results. The intuition is that different wordings activate different regions of embedding space, so combining them gives broader coverage. This can be useful for complex, multifaceted questions but may be unnecessary for simple factual queries.

Initialize with a base retriever and LLM. The LLM generates query variations, and the base retriever performs the actual searches:

```
# In api/services/retrieval.py

from langchain_openai import ChatOpenAI
import json

class RAGFusionRetriever:
    """
    Generates multiple query variations and fuses
    results. Best for complex, multifaceted questions.
    """

    def __init__(
        self, base_retriever, llm: ChatOpenAI
    ):
        self.base_retriever = base_retriever
        self.llm = llm
```

The main search generates variations, searches with each, then fuses. By default it creates three variations plus the original query (four total searches). Each variation gets `k*2` candidates to ensure good fusion results:

```
    def search(
        self,
        query: str,
        k: int = 5,
        num_queries: int = 3
    ) -> list:
        """Generate variations, search each, fuse."""
```

```
        variations = self._generate_variations(
            query, num_queries
        )
        all_queries = [query] + variations

        # Search with each variation
        all_results = []
        for q in all_queries:
            results = self.base_retriever.search(
                q, k=k*2
            )
            ranked = [
                (doc, 1.0 / (i + 1))
                for i, doc in enumerate(results)
            ]
            all_results.append(ranked)

        # Fuse with RRF
        fused = self._reciprocal_rank_fusion(all_results)
        return fused[:k]
```

Query generation uses an LLM with a carefully crafted prompt. We ask for different angles on the topic to maximize coverage. The response parsing handles common issues like Markdown code blocks wrapping the JSON:

```
    def _generate_variations(
        self, query: str, n: int
    ) -> list:
        """Use LLM to generate query variations."""
        prompt = f"""Generate {n} alternative search queries for:
"{query}"

Make each query approach the topic from a different angle.
Return ONLY a JSON array of strings, no markdown formatting."""

        try:
            response = self.llm.invoke(prompt)
            content = response.content.strip()

            # Strip markdown code blocks if present
            if content.startswith("```"):
                lines = content.split("\n")
                if lines[0].startswith("```"):
                    lines = lines[1:]
                if lines and lines[-1].strip() == "```":
                    lines = lines[:-1]
                content = "\n".join(lines)
```

```
            return json.loads(content)
        except (json.JSONDecodeError, Exception):
            return []
```

The fusion method is similar to `HybridRetriever` but without weights. Since all query variations are equally valid interpretations, they receive equal weight in the ranking:

```
    def _reciprocal_rank_fusion(
        self, result_lists: list, k: int = 60
    ) -> list:
        """Combine multiple ranked lists using RRF."""
        scores = {}

        for results in result_lists:
            for rank, (doc, _) in enumerate(results):
                doc_id = hash(doc.page_content)

                if doc_id not in scores:
                    scores[doc_id] = {
                        "doc": doc, "score": 0
                    }

                scores[doc_id]["score"] += (
                    1 / (k + rank + 1)
                )

        sorted_results = sorted(
            scores.values(),
            key=lambda x: x["score"],
            reverse=True
        )

        return [
            item["doc"] for item in sorted_results
        ]
```

The Adaptive Retriever

We wire everything together with a unified interface that chooses the right strategy: simple queries get fast basic search, while complex ones get full fusion power. This gives you good performance for simple queries while handling nuanced questions properly.

Wire up all three retriever types in initialization. The chain goes: `HybridRetriever` → `RerankedRetriever` → `RAGFusionRetriever`:

```
# In api/services/retrieval.py
```

```
class AdaptiveRetriever:
    """
    Selects retrieval strategy based on query
    characteristics.
    """

    def __init__(
        self,
        vector_store: QdrantVectorStore,
        documents: list,
        llm: ChatOpenAI
    ):
        self.hybrid = HybridRetriever(
            vector_store, documents
        )
        self.reranked = RerankedRetriever(self.hybrid)
        self.fusion = RAGFusionRetriever(
            self.reranked, llm
        )
        self.llm = llm
```

Route based on query complexity. Simple queries (≤4 words) skip expensive operations. Queries containing comparison keywords or conjunctions trigger the full fusion pipeline:

```
    def search(self, query: str, k: int = 5) -> list:
        """Adaptive search based on query complexity."""
        complexity = self._assess_complexity(query)

        if complexity == "simple":
            return self.hybrid.search(query, k)
        elif complexity == "moderate":
            return self.reranked.search(query, k)
        else:
            return self.fusion.search(query, k)

    def _assess_complexity(self, query: str) -> str:
        """Quick heuristic for query complexity."""
        words = query.split()

        if len(words) <= 4:
            return "simple"

        complex_indicators = [
            "compare", "difference", "relationship",
            "how does", "why does", "explain how",
            "and", "or", "versus", "vs"
        ]

        query_lower = query.lower()
```

```
        if any(
            ind in query_lower
            for ind in complex_indicators
        ):
            return "complex"

        return "moderate"
```

Running Evaluations

With the new retrieval system in place, we run our v8 evaluation datasets. This is where all the infrastructure from the previous chapter pays off. Instead of manually checking "does this feel better?" we get concrete numbers.

The comparison approach is straightforward: run the same test cases through both the old and new retrievers, then compare RAGAS metrics. Any improvement should be statistically significant, not just noise from query-to-query variation:

```
# In api/evaluation/retrieval_comparison.py

from ragas import evaluate
from ragas.metrics import context_precision, context_recall
from datasets import Dataset

def compare_retrieval_versions(
    old_retriever,
    new_retriever,
    test_cases: list[dict]
) -> dict:
    """Compare old and new retrieval systems."""
    results = {"old": [], "new": []}

    for case in test_cases:
        query = case["question"]
        ground_truth = case["ground_truth"]

        old_contexts = [
            doc.page_content
            for doc in old_retriever.search(query, k=5)
        ]
        new_contexts = [
            doc.page_content
            for doc in new_retriever.search(query, k=5)
        ]

        results["old"].append({
            "question": query,
            "contexts": old_contexts,
            "ground_truth": ground_truth
```

```
        })
        results["new"].append({
            "question": query,
            "contexts": new_contexts,
            "ground_truth": ground_truth
        })
```

Evaluate both versions with RAGAS. The metrics (`context_precision` and `context_recall`) tell us how well the retrieved content supports answering the questions:

```
    # Evaluate both
    old_scores = evaluate(
        Dataset.from_dict({
            "question": [
                r["question"] for r in results["old"]
            ],
            "contexts": [
                r["contexts"] for r in results["old"]
            ],
            "ground_truth": [
                r["ground_truth"]
                for r in results["old"]
            ]
        }),
        metrics=[context_precision, context_recall]
    )

    new_scores = evaluate(
        Dataset.from_dict({
            "question": [
                r["question"] for r in results["new"]
            ],
            "contexts": [
                r["contexts"] for r in results["new"]
            ],
            "ground_truth": [
                r["ground_truth"]
                for r in results["new"]
            ]
        }),
        metrics=[context_precision, context_recall]
    )
```

Compute and return the improvement delta. Positive numbers mean the new retriever is better:

```
    return {
        "old": old_scores,
```

```
        "new": new_scores,
        "improvement": {
            "context_precision": (
                new_scores["context_precision"]
                - old_scores["context_precision"]
            ),
            "context_recall": (
                new_scores["context_recall"]
                - old_scores["context_recall"]
            )
        }
    }
```

The Retrieval Comparison Tool

Beyond just running evaluations, we want a way to visually inspect what's happening. A side-by-side comparison tool helps debug cases where the new retriever performs worse (yes, this happens) and understand why the new retriever performs better when it does:

```
# In api/evaluation/retrieval_comparison.py

import time

def side_by_side_comparison(
    query: str,
    old_retriever_fn,
    new_retriever_fn,
    k: int = 5
) -> dict:
    """
    Generate a side-by-side comparison for manual
    inspection.

    Useful for debugging and understanding retriever
    behavior.
    """
    # Time both retrievers
    start = time.time()
    old_results = old_retriever_fn(query, k)
    old_latency = time.time() - start

    start = time.time()
    new_results = new_retriever_fn(query, k)
    new_latency = time.time() - start
```

Build structured result objects for each retriever. We include a 200-character preview (so you can quickly scan), source metadata, and scores when available:

```
comparison = {
    "query": query,
    "old_retriever": {
        "latency_ms": round(
            old_latency * 1000, 2
        ),
        "results": [
            {
                "rank": i + 1,
                "content_preview": (
                    doc.page_content[:200]
                    + "..."
                ),
                "source": doc.metadata.get(
                    "source", "unknown"
                ),
                "score": doc.metadata.get(
                    "score", None
                )
            }
            for i, doc in enumerate(old_results)
        ]
    },
```

Do the same for the new retriever. Note that it uses `rerank_score` instead of the generic `score`:

```
    "new_retriever": {
        "latency_ms": round(
            new_latency * 1000, 2
        ),
        "results": [
            {
                "rank": i + 1,
                "content_preview": (
                    doc.page_content[:200]
                    + "..."
                ),
                "source": doc.metadata.get(
                    "source", "unknown"
                ),
                "score": doc.metadata.get(
                    "rerank_score", None
                )
            }
            for i, doc in enumerate(new_results)
        ]
    },
```

Compute overlap as the count of documents appearing in both result sets. High overlap (>3) means the retrievers largely agree; low overlap suggests they're finding different relevant content:

```
        "overlap": len(
            set(
                hash(d.page_content)
                for d in old_results
            )
            & set(
                hash(d.page_content)
                for d in new_results
            )
        )
    }

    return comparison
```

The report generator iterates through queries and formats a Markdown document. This is useful for sharing evaluation results with teammates or including in documentation.

```
def generate_comparison_report(
    test_queries: list[str],
    old_retriever,
    new_retriever
) -> str:
    """Generate a markdown comparison report."""

    report = "# Retrieval Comparison Report\n\n"

    for query in test_queries:
        comparison = side_by_side_comparison(
            query, old_retriever, new_retriever
        )

        report += f"## Query: {query}\n\n"
        report += (
            f"**Overlap:** "
            f"{comparison['overlap']}/5 "
            f"results in common\n\n"
        )
```

List results from each retriever with formatting for scores. The new retriever's score is more prominent since reranking provides it.

```
        report += "### Old Retriever\n"
        for r in comparison[
            "old_retriever"
```

```
        ]["results"]:
            report += (
                f"{r['rank']}. [{r['source']}] "
                f"{r['content_preview']}\n"
            )

        report += "\n### New Retriever\n"
        for r in comparison[
            "new_retriever"
        ]["results"]:
            score_str = (
                f" (score: {r['score']:.3f})"
                if r['score'] else ""
            )
            report += (
                f"{r['rank']}. "
                f"[{r['source']}]{score_str} "
                f"{r['content_preview']}\n"
            )

        report += "\n---\n\n"

    return report
```

Running Locally

StudyBuddy v9 runs with two terminals: one for the backend and one for the frontend. On the backend terminal, run the following commands:

```
cd v9-optimized-retrieval
uv sync
uv run uvicorn api.index:app --reload --port 8000
```

Now run the following on the frontend terminal:

```
cd v9-optimized-retrieval/frontend
npm run dev
```

Using the New Features

With v9 running, you can test the advanced retrieval in several ways.

Chat Normally and Observe the Difference

The tutor now uses adaptive retrieval automatically. Ask a simple question like "What is an embedding?" and watch the response. Then ask something

complex like "Compare the trade-offs between dense retrieval and BM25 for different query types" and notice how the system adapts. The retrieval strategy changes based on query complexity, though the user experience feels seamless.

Compare Retrieval Strategies Directly

The new comparison endpoint lets you see exactly what's happening under the hood:

```
curl -X POST http://localhost:8000/api/evaluation/compare-retrievers \
  -H "Content-Type: application/json" \
  -d '{
    "queries": [
      "What is RAG?",
      "error code E-4012",
      "Compare embeddings and keyword search for different use cases"
    ],
    "k": 5
  }' | jq .
```

This returns a side-by-side comparison showing: results from the simple retriever (basic vector search), results from the adaptive retriever (hybrid + reranking + fusion), latency for each approach, and overlap between result sets.

The response looks like this:

```
{
  "comparisons": [
    {
      "query": "What is RAG?",
      "old_retriever": {
        "latency_ms": 45.2,
        "results": [...]
      },
      "new_retriever": {
        "latency_ms": 156.8,
        "results": [...]
      },
      "overlap": 3
    }
  ],
  "summary": {
    "total_queries": 3,
    "avg_simple_latency_ms": 42.1,
    "avg_adaptive_latency_ms": 178.3,
    "avg_overlap": 2.7
  }
}
```

Run the Full Evaluation Pipeline

To get quantified metrics, use the evaluation infrastructure from v8:

```
# First, generate a test set (if you haven't already)
curl -X POST http://localhost:8000/api/evaluation/generate-testset \
  -H "Content-Type: application/json" \
  -d '{"test_size": 5}'

# Then run retrieval evaluation
curl -X POST http://localhost:8000/api/evaluation/run \
  -H "Content-Type: application/json" \
  -d '{"dataset_name": "your-dataset-name", "eval_type": "retrieval"}'
```

This gives you RAGAS metrics—context precision and context recall—that quantify how much better the new retrieval performs compared to baseline.

Inspect the Semantic Chunks

You can see how documents were chunked by checking the indexing output when the server starts:

```
Indexing reference guides with semantic chunking (v9)...
Indexed 47 chunks from Embeddings
Indexed 52 chunks from Vector Databases
Indexed 38 chunks from Rag Pipeline
...
Total: 412 semantic chunks indexed from 12 guides
```

Compare this to v8's fixed-size chunking output. You'll typically see fewer but more coherent chunks, since semantic chunking respects section boundaries rather than cutting at arbitrary character positions.

What's Next

In Chapter 10, we'll transform StudyBuddy from a backend system into a full-stack application with a real user interface. Dashboard views showing learning progress, flashcard review screens with swipe-to-answer interactions, and document upload for custom learning materials—the whole experience. We'll also generalize StudyBuddy so users can learn any subject, not just AI engineering. The retrieval upgrades from this chapter will power all of it, delivering fast, accurate results to users through a polished interface.

Let's keep building!

CHAPTER 10

Full-Stack Applications

You've built something real. Over the last nine chapters, you've gone from basic LLM calls to sophisticated multi-agent systems with memory, evaluation infrastructure, and production-grade retrieval. That's serious engineering. But up to this point, we've been most concerned with getting the plumbing working. That's fine for developers, but it's not how you ship products.

This chapter is about crossing the finish line. We're transforming backend agent systems into full-stack applications that people can use. Dashboards that visualize learning progress. Chat interfaces that stream responses in real time. File upload workflows that let users bring their own content. The whole experience.

The patterns here apply far beyond any single project. Whether you're building a customer support assistant, a content generation tool, or a knowledge management system, the architecture stays the same. The frontend talks to the backend, the backend orchestrates agents, the agents do the heavy lifting, and users see great content. Clean separation. Clear responsibilities.

Industry Use Cases

Before we dive into architecture, let's talk about what's working in production out there in the real world. The LLM application landscape has matured significantly, and certain patterns have emerged as clear winners.

Customer support automation leads the pack in deployed applications. Companies route incoming tickets through AI agents that can handle routine inquiries, gather information for complex cases, and escalate appropriately. The best implementations don't try to replace human agents entirely. Instead, they augment the team by handling the repetitive stuff and providing context when humans need to step in.

Content generation tools have found their groove in specific niches rather than general-purpose writing. Marketing teams use them for product descriptions, social media posts, and email variations. Documentation teams use them to draft technical content that subject matter experts refine. The key insight is that AI-generated content works best as part of a workflow: first draft from AI, refinement from humans, final review before publishing.

Knowledge management represents perhaps the most valuable enterprise use case. Organizations have decades of documentation scattered across wikis, shared drives, and individual hard drives. RAG-powered search lets employees find relevant information using natural language rather than guessing the exact keywords someone used five years ago. The best implementations combine semantic search with proper access controls.

Learning and training applications have grown rapidly in both corporate and consumer contexts. Adaptive tutoring systems adjust difficulty based on performance. Onboarding assistants help new employees navigate company processes. Language learning apps provide conversational practice at any hour. These applications share a common pattern: they maintain state about the learner across sessions, personalize content based on observed behavior, and provide feedback that helps users improve.

What doesn't work is equally instructive. Applications that try to hide the AI tend to backfire when users discover the deception. Overpromising capabilities leads to disappointment and churn. Ignoring latency makes experiences feel sluggish even when the underlying system is powerful. And shipping without proper evaluation means you're finding bugs through user complaints rather than systematic testing.

The lessons from successful deployments are clear:

- **Start narrow and expand.** Companies that succeeded focused on one use case, nailed it, and then broadened the scope. Those that tried to build general-purpose AI assistants from day one often ended up with mediocre performance across the board.
- **Invest in feedback loops early.** The applications that improve fastest have mechanisms for collecting user corrections and incorporating them into system improvements.

- **Set expectations clearly.** Users who understand what an AI can and can't do have dramatically higher satisfaction than those who discover limitations through failure.
- **Measure everything.** Teams that track latency, accuracy, user satisfaction, and cost per query make better decisions than those flying blind.

Architecture Overview

Full-stack AI applications follow a three-layer architecture that keeps concerns cleanly separated (see Figure 10.1). The frontend handles user interaction and display. The backend manages business logic, authentication, and orchestration. The agent layer performs AI operations. Each layer has distinct responsibilities, and the interfaces between them should be explicit and stable.

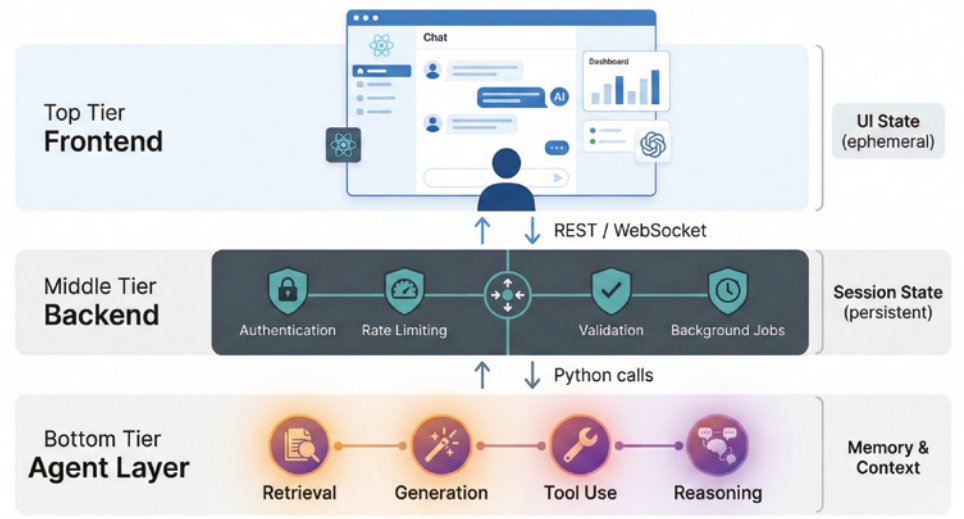

Figure 10.1: The three-tier architecture of a full-stack AI application.

The Frontend Layer

The frontend is everything the user sees and interacts with. In a web application, this generally means the HTML and web framework components, CSS styling, and client-side JavaScript that renders the interface. The frontend knows nothing about how agents work internally. It sends requests, receives responses, and displays results. This ignorance is intentional. When you upgrade your agent architecture, the frontend shouldn't need to change. When you redesign the UI, the agent logic stays untouched.

The Backend Layer

The backend sits between the frontend and agent layers, handling concerns that neither should worry about:

- **Authentication** verifies that requests come from legitimate users.
- **Rate limiting** protects your API budget from runaway usage.
- **Request validation** ensures that malformed inputs don't reach your agents.
- **Background job scheduling** manages long-running tasks.

The backend exposes a clean REST or WebSocket API to the frontend while orchestrating potentially complex agent interactions behind the scenes.

The Agent Layer

The agent layer is where AI operations happen. Retrieval, generation, tool use, multistep reasoning—all of that lives here. Agents receive structured inputs and return structured outputs. They don't know about HTTP status codes or session cookies. They don't handle authentication. They just do their job: take a question and context and return an answer. This isolation makes agents easier to test, easier to swap out, and easier to reason about.

Bringing It All Together

State management spans all three layers, and getting it right is critical:

- **Frontend state** covers UI concerns such as which panel is open, what the user has typed, and whether a request is in flight. This state is ephemeral and can be reconstructed if lost.
- **Backend state** includes session and orchestration data: which user is authenticated, what background jobs are running, and what conversations are in progress.
- **Agent state** maintains conversation memory and retrieved context: what was discussed previously and what documents are relevant.

Each layer owns its state and synchronizes with others through well-defined interfaces.

The pattern of clean separation pays dividends as your application grows. When you need to support mobile apps alongside your web interface, you add a new frontend without touching the backend or agents. When you want to swap OpenAI for an open-source model, you modify the agent layer without touching the frontend or backend. Modularity enables evolution.

Communication between layers happens through well-defined APIs. The frontend talks to the backend via REST endpoints for standard operations and WebSockets for real-time streaming. The backend talks to agents through Python function calls or service classes. These boundaries should be explicit in your codebase. For example, you might use a `services/` directory for agent wrappers and an `api/` directory for endpoint definitions.

One common mistake is letting agent concerns leak into the backend. When your route handlers start containing prompt templates or retrieval logic, you've violated the separation. Extract agent logic into dedicated modules, expose clean interfaces, and keep your route handlers thin. Another mistake is putting business logic in the frontend. Tasks such as validation, authorization, and rate limiting belong in the backend, where they can't be bypassed.

The pattern of clean separation pays dividends as your application grows and changes. For example, if down the road you need to support mobile apps alongside your web interface, you can add a new frontend without touching the backend or the agents. Similarly, if later you want to swap OpenAI for, say, an open-source model, you can modify the agent layer without touching the frontend or backend.

The sections that follow build from the inside out. We start with the backend architecture that powers everything, then tackle the frontend components that users interact with. From there, it's on to state management across all three layers, then finally agent integration to see how the backend talks to the AI.

Backend Architecture

FastAPI has become the framework of choice for AI application backends, and for good reason. It's fast, it handles async operations naturally, and its automatic OpenAPI documentation makes frontend integration smoother. The patterns you establish here determine how well your application scales and how pleasant it is to maintain.

Request handling in an AI backend differs from traditional CRUD applications because of response times. A typical database query returns in milliseconds. An LLM call might take several seconds. That mismatch means you need to think carefully about timeouts, streaming, and background processing. For streaming chat endpoints, FastAPI's `StreamingResponse` paired with an async generator function is the standard approach. You define your route, accept a Pydantic request model containing the user's message and optional conversation ID, then return a `StreamingResponse` that wraps your generator. Inside the generator, you iterate over chunks from your agent's `astream()` method, formatting each chunk as an SSE `data:` line and yielding it. When the stream is complete,

you yield a final `data: [DONE]` sentinel so the client knows to stop reading. Headers like `Cache-Control: no-cache` and `Connection: keep-alive` prevent intermediaries from buffering the stream.

NOTE **You can find all of this chapter's non-StudyBuddy code in the Jupyter notebook `chapter10.ipynb`, available in this book's GitHub repo.**

Here's the function that generates an SSE stream in this chapter's Jupyter notebook:

```
async def generate_sse_stream(message: str):
    async for chunk in agent_service.chat_stream(message):
        if chunk:
            yield f"data: {json.dumps(chunk)}\n\n"
    yield "data: [DONE]\n\n"
```

In the notebook, `generate_sse_stream()` runs standalone so you can inspect the raw SSE output. In an actual API, you wrap it in FastAPI's `StreamingResponse` with `media_type="text/event-stream"` and a couple of headers (`Cache-Control: no-cache` and `Connection: keep-alive`) to prevent proxies from buffering the stream. The full route implementation is in `v10-full-stack/api/index.py` in the repo. What matters here is the generator itself: it yields one `data:` line per chunk, JSON-encodes to preserve newlines, and signals completion with `[DONE]`.

WebSocket integration enables bidirectional real-time communication that SSE can't match. When you need the client to send messages while the server is still streaming, or push notifications without polling, WebSockets are the answer. The typical pattern uses a `ConnectionManager` class that maintains active connections keyed by user ID, with `connect()`, `disconnect()`, and `send_message()` methods. Your WebSocket endpoint accepts connections, registers them with the manager, and enters a receive loop that dispatches incoming messages to your agent layer, streaming chunks back as they arrive. The trade-off is complexity: WebSocket connections require more careful state management, and handling reconnection gracefully takes effort.

Background job processing handles tasks that take too long for a synchronous request-response cycle. Indexing uploaded documents, generating comprehensive reports, and running batch evaluations are operations that might take minutes. The pattern is to accept the request, queue the job, return a job ID immediately, and let clients poll for completion. FastAPI's `BackgroundTasks` parameter lets you fire off a background function after the response has been sent. You create a `JobStatus` model with fields for the job ID, status, optional result, and optional error. Your upload endpoint saves the file, creates a pending job, adds the indexing function to `background_tasks`, and returns the job ID. A separate status endpoint lets clients poll by job ID.

Queue management becomes important at scale. The simple in-memory approach works for development, but production systems need persistent queues that survive server restarts. Redis with RQ or Celery are popular choices. For AI applications specifically, queue design needs to account for variable job duration, since an LLM call might take one second or thirty seconds, depending on output length and model load. The solution is often separate queues for different job types:

- **Fast queue** for chat responses and quick lookups
- **Slow queue** for document indexing and content generation
- **Batch queue** for nightly processing and bulk operations

Workers can be allocated to queues based on expected load patterns.

Timeout handling requires special attention. HTTP requests typically time out after 30 to 60 seconds, but generating a comprehensive response might take longer. Accept the initial request, return a job ID, and switch to async polling or WebSocket updates for results.

Health checks and graceful shutdown round out production backend concerns. A health endpoint that checks database connectivity and agent availability prevents your load balancer from routing traffic to broken instances.

Frontend Components

Modern AI applications demand interfaces that go far beyond the terminal REPL you've been using for development. Users expect responsive, polished experiences that feel like the other software they use daily. That means proper loading states, streaming text, intuitive navigation, and mobile support.

Chat interfaces are a core component of conversational AI applications. The basic pattern involves a message list showing the conversation history and an input field for new messages. Key elements to get right:

- **Visual distinction** between user inputs and AI responses through different background colors, alignment, or avatars, making conversations scannable at a glance
- **Timestamps** to help users orient themselves in long conversations
- **Markdown rendering** so AI responses can include formatted code, lists, and emphasis without displaying raw syntax
- **Auto-scroll** that follows new messages as they arrive but still lets users scroll up through history without getting pulled back to the bottom

Streaming responses deserve special attention because they fundamentally change how users perceive AI interactions. A response that takes ten seconds to

generate feels much faster when users see it appear word by word than when they stare at a spinner for the full duration. The implementation involves your frontend consuming a stream from the backend and progressively updating the displayed content. In JavaScript, you use the Fetch API's `response.body.getReader()` to read chunks from a `ReadableStream`, decode each chunk with `TextDecoder`, and parse Server-Sent Events (SSE) lines to extract content. As each chunk arrives, you accumulate the text and update the displayed message via state. A custom React hook such as `useStreamingChat` can encapsulate this logic cleanly, managing the message array, a `currentResponse` string for the in-progress stream, and an `isStreaming` flag.

Here's an example function to handle streaming responses:

```
async function streamChat(message, onChunk) {
    const response = await fetch('/api/chat/stream', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ message }),
    });

    const reader = response.body.getReader();
    const decoder = new TextDecoder();

    while (true) {
        const { done, value } = await reader.read();
        if (done) break;

        const lines = decoder.decode(value).split('\n');
        for (const line of lines) {
            if (!line.startsWith('data: ')) continue;
            const data = line.slice(6);
            if (data === '[DONE]') return;
            try {
                onChunk(JSON.parse(data));
            } catch {
                onChunk(data);
            }
        }
    }
}
```

Each chunk arrives as a `data:` line. JSON decoding restores any newlines the LLM included in its response, which prevents text such as code blocks and bullet points from collapsing into a single line.

Mobile-responsive design is nonnegotiable for consumer applications. Use responsive CSS that adapts layouts to screen size. Ensure touch targets are at least 44 pixels in each dimension. Test on actual devices, not just browser dev tools, because the feel of an interface changes when you're holding it in your hand.

User Experience Considerations

The difference between an AI application that feels smooth and one that feels frustrating often comes down to user experience details. Loading states, feedback mechanisms, and accessibility aren't afterthoughts—they're core to whether people want to use what you've built.

Loading states deserve careful design because AI operations take longer than users expect from modern software. The hierarchy of loading experiences, from worst to best:

- **Frozen interface** with no indication that anything is happening. The worst possible experience.
- **Generic spinner.** Better, but tells users nothing about progress or expected wait time.
- **Skeleton screen** showing the shape of the content that will appear. Reduces perceived wait time significantly.
- **Progress indicator with stage information** so users know what's happening. This could be a component that displays stages such as "Analyzing your request," "Finding relevant information," and "Generating response," transitioning from pending to active to complete.

ARIA (Accessible Rich Internet Applications) attributes are HTML annotations that communicate dynamic UI state to assistive technologies such as screen readers. Adding `role="status"` and `aria-live="polite"` to your progress indicator ensures that screen readers announce updates as they happen, so users who can't see the spinner still know the system is working.

Transparency about AI behavior builds trust. Users should understand that they're interacting with an AI, what it's capable of, and where its information comes from. Showing source citations for RAG applications lets users verify claims. Explaining when the AI is uncertain helps set appropriate expectations.

Progressive disclosure manages complexity by showing users only what they need at each moment. The initial interface might show just a chat input and response. Users who want more can expand panels showing source documents, confidence scores, or intermediate reasoning steps. Power users get access to advanced configuration without overwhelming newcomers.

Accessibility isn't optional; it's essential for any application serving a broad audience and often legally required for enterprise software. The fundamentals:

- **Semantic HTML elements** that screen readers understand
- **ARIA labels** for interactive components that lack visible text
- **Keyboard navigation** for everything a mouse can do
- **Sufficient color contrast** for users with visual impairments

A chat input, for example, should use a `<form>` with a visually hidden `<label>`, a `<textarea>` with `aria-describedby` pointing to a keyboard shortcut hint, and a submit button with an `aria-label`. These details cost almost nothing to implement but make the difference between an app that works for everyone and one that excludes a significant portion of your user base.

Error messages deserve special attention in AI applications because failures are more common and more confusing than in traditional software. Good error handling distinguishes between infrastructure errors (API rate limits, network failures) and capability errors (the AI genuinely doesn't know how to help). Infrastructure errors should trigger retries or clear user guidance. Capability errors should prompt graceful degradation. And the tone matters: "I'm having trouble connecting right now. Want to try again?" is far better than "Error: request failed."

Consider how errors affect user trust. A single catastrophic failure—losing a user's work, generating offensive content, or exposing private information—can permanently damage the relationship. Autosave user inputs before sending them to the AI. Filter outputs before displaying them. Validate that responses make sense before presenting them as answers.

State Management

State management in full-stack AI applications spans all three layers—backend, frontend, and agent—each with different persistence requirements and synchronization challenges. Getting this right determines whether your app feels responsive and reliable or sluggish and buggy.

Client-side state covers everything that lives in the user's browser: which tabs are selected, whether a modal is open, and what the user has typed. Manage this with whatever framework you're using: React's `useState` and `useReducer`, Vue's reactive system, or vanilla JavaScript. The key principle is that client state should be reconstructable. If the user refreshes the page, some state loss is acceptable.

For state that should survive page refreshes but doesn't need server persistence, `localStorage` provides a simple solution. Draft messages, dark mode preferences, and recently accessed items are all good candidates. The pattern is a custom hook (such as `usePersistentState`) that initializes from `localStorage` on mount and syncs back on every change, wrapping `useState` with a lazy initializer and a `useEffect` that writes back whenever the value updates.

Server-side state is the source of truth for anything that matters. Data such as user profiles, conversation history, generated content, and progress data all live in your database. PostgreSQL remains the right choice for most applications. The discipline of treating the server as the source of truth simplifies debugging and enables features like multi-device sync automatically.

State synchronization between client and server requires careful thought. The naive approach fetches fresh data on every interaction, creating unnecessary latency. A better approach uses optimistic updates: update the UI immediately as if the action succeeded, send the request to the server, and roll back only if the server reports failure. You add the message to local state with a temporary ID and `sending` status, fire the API call, then replace the temporary message with the server-confirmed one on success, or mark it `failed` on failure. The user sees instant responsiveness, and the rare rollback is a small price for the dramatic improvement in perceived speed.

Here's an example implementation of optimistic updates:

```
store = OptimisticMessageStore()

# Step 1: Show the message immediately with a temp ID
temp_id = store.add_optimistic("What's my order status?")
# UI renders the message instantly, status = "sending"

# Step 2a: Server confirms — swap temp ID for real ID
store.confirm(temp_id, real_id="msg-server-abc123")
# UI updates, status = "sent"

# Step 2b: Or server rejects — mark it failed so user can retry
store.rollback(temp_id, error="Network timeout — please retry")
# UI shows retry option, status = "failed"
```

The user sees the message the moment they hit send. The `confirm/rollback` distinction handles the two outcomes: success silently updates the ID, while failure displays a retry prompt. Either way, the interface never freezes while waiting for the server.

Handling offline scenarios gracefully matters, especially on mobile where connections can be unreliable. The minimal version detects network unavailability and shows a banner. A more sophisticated approach queues actions taken while offline and replays them when connectivity returns. For AI applications, consider caching recent responses so users can review previous conversations offline.

The boundary between client and server state can blur. A useful heuristic: if losing the data would be merely inconvenient, it can live on the client. If losing it would be a problem, it belongs on the server. Disk space is cheap; user frustration is expensive.

Agent Integration

Connecting your frontend and backend to your agent layer requires thoughtful interface design. The goal is clean abstraction: your backend shouldn't need to know the internal structure of your agents, and your agents shouldn't know they're being called from a web application.

The service layer pattern wraps your agents in classes that expose simple, typed interfaces. You define an `AgentService` class that accepts your agent and retriever as constructor arguments, then exposes high-level methods like `chat()` and `chat_stream()`. The `chat()` method invokes the agent, extracts the response content, and returns a structured `ChatResponse` dataclass with fields for `content`, `sources`, and `confidence`. The `chat_stream()` method is an async generator that calls `agent.astream()` with a stream mode of `"updates"`, iterates over events, and yields content chunks as strings. Your FastAPI routes call these service methods without ever touching agent internals. When you upgrade the underlying model or swap retrieval strategies, no backend routes or frontend code need to change.

Long-running operations need special handling. When a user asks for something that takes 30 seconds, you can't just leave them staring at a spinner. The pattern is an async generator that yields `ProgressUpdate` dataclasses (with fields for `stage`, `message`, and `progress` as a float from 0.0 to 1.0) at meaningful milestones, then yields the result at the end. The frontend reads this stream and updates a progress bar and stage indicator in real time.

Error handling in agent integrations needs to distinguish between different failure modes:

- **Network errors** are transient and should trigger automatic retries.
- **Rate limit errors** need backoff, since retrying `immediately` just makes things worse.
- **Authentication errors** need user action, such as refreshing an API key.
- **Agent errors** are cases where the AI genuinely doesn't know how to help, so they need graceful degradation and helpful messages.

The retry logic deserves careful calibration. The typical pattern combines three strategies:

- **Exponential backoff:** Wait longer after each consecutive failure.
- **Maximum retry count:** Give up after *N* attempts rather than retrying forever.
- **Circuit breaking:** If too many requests fail in a window, stop trying temporarily to let the service recover.

The Python library Tenacity handles retry logic with composable decorators. You combine a stop condition, a wait strategy, and a filter that specifies which exception types should trigger retries:

```
@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=2, max=10),
    retry=retry_if_exception_type(RetryableError),
)
```

```
async def call_agent_with_retry(message: str) -> str:
    result = await agent_service.chat(message)
    return result.content
```

The specific values here—three attempts, exponential backoff starting at two seconds and capping at ten—are reasonable starting points, but they're configurable. For example, a high-volume application might use a more aggressive backoff, while batch jobs might tolerate more retries. You define custom exception classes—`RetryableError` for transient failures and `UserActionRequired` for permanent ones—and route different upstream errors into the appropriate bucket, so Tenacity knows which failures to retry and which to surface immediately.

Database Design for Agent Apps

Database schema design for AI applications follows the same principles as traditional applications, with a few AI-specific considerations. You're storing user data, tracking progress, persisting generated content, and enabling analytics. The schema you design now determines how easily you can add features later.

User and session management form the foundation. Even before you implement authentication, designing your schema with user isolation in mind pays dividends. Every piece of user-generated or user-specific data should have a `user_id` foreign key. This makes multi-tenancy a schema addition rather than a rewrite. Your `User` model needs an ID (UUID strings work well), an email, and a creation timestamp. From this root, everything branches out through relationships.

Workspaces (or projects, or programs—the name depends on your domain) organize user content into logical groups. Each workspace has its own name, description, optional JSON settings, and timestamps. The `Workspace` model connects back to `User` via `user_id` and forward to `Document` records through a relationship.

Document storage tracks uploaded files and their indexed content. The actual vector embeddings live in your vector database, but metadata belongs in your relational database, where you can query and join it easily. A `Document` model captures the workspace, filename, file type, file size, and storage path. Critically, it also tracks indexing workflow state:

- `status` field (`pending`, `processing`, `indexed`, or `failed`) for real-time progress display
- `chunks_count` for how many chunks were produced during indexing
- `indexed_at` timestamp for when indexing completed
- `error_message` for surfacing problems without querying the vector store

Conversation history enables continuity across interactions. A `Conversation` model links to both user and workspace, tracks activity timestamps, and maintains a message count. The `Message` model stores the basics—role, content, and timestamp—plus AI-specific fields that traditional applications don't need:

- `tokens_used` and `model` for cost tracking and model versioning
- `feedback` and `feedback_text` for quality signals you can query later
- `created_at` timestamps for analytics and debugging

Generated content storage matters for AI applications that create artifacts users want to keep. A `GeneratedContent` model stores the content type, title, and full content alongside generation metadata. One particularly valuable field is `input_hash`, an indexed hash of the input that produced this content, enabling cache lookups before running expensive generation.

This schema design anticipates growth. Adding authentication means adding password hashes to the `User` model, not restructuring everything. Adding team features means adding a `Team` model and `team_id` foreign keys. The foundation supports evolution.

Building StudyBuddy v10

Here's where it all comes together. We're transforming StudyBuddy from a backend system with a basic interface into a full-stack application that users actually want to use. More importantly, we're generalizing it so users can learn any subject, not just AI engineering.

Let's take stock of where we left off. StudyBuddy v9 delivered production-grade retrieval with semantic chunking, hybrid search, reranking, and RAG-Fusion. The evaluation infrastructure from v8 proved these improvements with actual metrics. But users still interact through a basic web interface designed for testing, not for learning. And the system only works with the AI engineering reference guides we've been using all along.

Version 10 changes both of those things. The major feature is generalizing StudyBuddy to learn any subject. Users can upload their own documents, create custom topic lists, and manage multiple learning programs simultaneously. Want to study Spanish vocabulary in the morning and AWS certification in the afternoon? You can do that. The second major change is building a proper user interface: dashboards that visualize learning progress, streaming chat that feels responsive, and polished interactions that make the system enjoyable to use every day.

Learning Program Architecture

The core concept is a Learning Program, a container that holds everything related to studying a particular subject. Each program has its own knowledge base (uploaded documents indexed in their own Qdrant collection), its own topic list defining what to study, its own flashcards, and its own progress tracking. Users can create as many programs as they want and switch between them freely:

```
from dataclasses import dataclass
from typing import Optional
from datetime import datetime

@dataclass
class LearningProgram:
    id: str
    user_id: str
    name: str
    description: str  # Required - describes what the user wants
to learn
    topic_list: dict  # Structured topic hierarchy
    collection_name: str  # Qdrant collection for this program
    created_at: datetime

    @property
    def qdrant_collection(self) -> str:
        """Collection name for this program's vectors."""
        return f"program_{self.id}"
The retrieval infrastructure from Chapter 9—hybrid search, reranking,
RAG-Fusion—adapts naturally to program-scoped collections. Each program
gets its own retriever instance configured with the program's Qdrant
collection:
from langchain_openai import OpenAIEmbeddings
from langchain_qdrant import QdrantVectorStore
from qdrant_client import QdrantClient

def get_program_retriever(program: LearningProgram):
    """Create a retriever for a program's knowledge base.

    This reuses the v9 retrieval infrastructure with a
    program-specific Qdrant collection.
    """
    embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

    client = QdrantClient(url="http://localhost:6333")

    vector_store = QdrantVectorStore(
        client=client,
        collection_name=program.qdrant_collection,
```

```
        embedding=embeddings
    )

    # Use adaptive retrieval from Chapter 9
    # Simple queries get hybrid search, complex queries get RAG-Fusion
    return AdaptiveRetriever(
        vector_store=vector_store,
        embeddings=embeddings,
        collection_name=program.qdrant_collection
    )
```

When a user creates a new program, they provide a name and description (both required), then choose between two options. They can start empty and upload their own documents, which the AI tutor will use directly, or they can let the AI generate a curriculum based on their description. The curriculum generator from Chapter 7 gets repurposed here: give it a topic and depth level, and it creates a comprehensive learning plan. When a program is created, the system automatically generates an initial set of flashcards in the background, so users have something to study immediately.

The curriculum generation flow demonstrates how AI can bootstrap content that users then customize. The generated curriculum is a starting point, not a final product. Users can edit topics, add their own materials, and adjust the structure. This human-in-the-loop approach produces better results than either pure AI generation or purely manual creation. The AI handles the tedious work of structuring a comprehensive curriculum; the human provides domain expertise and personal learning preferences.

For complex topics, curriculum generation benefits from additional context. When a user asks to learn "machine learning," the results are better if we ask clarifying questions: What's your math background? Are you interested in research or application? Do you have a specific project in mind? These questions let us tailor the curriculum to the learner rather than generating generic content:

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

curriculum_prompt = ChatPromptTemplate.from_template("""
You are an expert curriculum designer. Create a comprehensive
learning curriculum for the topic: {topic}

Depth level: {depth} (beginner, intermediate, or advanced)

Structure your response as a markdown document with:
- Chapters using # Chapter N: Title format
- Topics using ## Topic Name format
- Subtopics using - Subtopic format
```

```
Include {chapter_count} chapters covering the subject systematically,
from foundational concepts to advanced applications.

Focus on practical, actionable learning objectives.
""")

async def generate_curriculum(
    topic: str,
    depth: str = "intermediate",
    chapter_count: int = 8
) -> str:
    """Generate a curriculum for any topic."""

    llm = ChatOpenAI(model="gpt-4o", temperature=0.7)

    chain = curriculum_prompt | llm

    result = await chain.ainvoke({
        "topic": topic,
        "depth": depth,
        "chapter_count": chapter_count
    })

    return result.content
```

Document Upload and Indexing

Users upload documents to build their program's knowledge base. The system accepts PDF, Markdown, and plain-text files. Each upload triggers a background job that parses the document, chunks it using the semantic chunking from Chapter 9, generates embeddings, and stores everything in the program's dedicated Qdrant collection.

The upload interface uses drag-and-drop for a modern feel. Users can drop multiple files at once, and each file shows its indexing status: uploading, processing, indexed, or failed. A progress indicator shows how indexing is progressing for large documents. And notifications alert users when indexing completes so they can start studying immediately.

File size limits protect your infrastructure from abuse. A reasonable limit for most applications is 10–50 MB per file. Documents larger than that are often better split into chapters or sections anyway. The limit should be communicated clearly in the UI so users don't waste time uploading files that will be rejected.

Duplicate detection saves users from accidentally indexing the same document twice. Compute a hash of the file content when the file is uploaded and check it against existing documents in the program. If a duplicate is detected, inform the user and offer options: skip the duplicate, replace the existing document, or index anyway with a different name.

First, the constraints and imports:

```
from fastapi import UploadFile, BackgroundTasks
from pathlib import Path
import aiofiles
import hashlib

UPLOAD_DIR = Path("uploads")
ALLOWED_TYPES = {"application/pdf", "text/markdown", "text/plain"}
MAX_FILE_SIZE = 50 * 1024 * 1024  # 50MB
```

Next, the endpoint itself:

```
@app.post("/api/programs/{program_id}/documents")
async def upload_document(
    program_id: str,
    file: UploadFile,
    background_tasks: BackgroundTasks,
    user: User = Depends(get_current_user)
):
    """Upload a document to a learning program."""

    # Verify program belongs to user
    program = await get_program(program_id)
    if program.user_id != user.id:
        raise HTTPException(status_code=403, detail="Not your program")

    # Validate file type
    if file.content_type not in ALLOWED_TYPES:
        raise HTTPException(
            status_code=400,
            detail=f"Unsupported file type. Allowed: PDF, Markdown,
plain text"
        )

    # Read and validate file size
    content = await file.read()
    if len(content) > MAX_FILE_SIZE:
        raise HTTPException(
            status_code=400,
            detail=f"File exceeds {MAX_FILE_SIZE // 1024 // 1024}MB limit"
        )
```

Deduplication is a fast hash check against documents already in this program. If we've seen this exact file before, we surface that to the user rather than silently re-indexing:

```
    # Check for duplicates
    content_hash = hashlib.sha256(content).hexdigest()
```

```
existing = await get_document_by_hash(program_id, content_hash)
if existing:
    raise HTTPException(
        status_code=409,
        detail=f"Document already uploaded as '{existing.filename}'"
    )
```

With the file validated, we persist it to disk and create a database record with `status="pending"`. This is what the UI polls on. The actual indexing runs in the background afterward:

```
# Save uploaded file
file_path = UPLOAD_DIR / program_id / file.filename
file_path.parent.mkdir(parents=True, exist_ok=True)

async with aiofiles.open(file_path, 'wb') as f:
    await f.write(content)

# Create document record
document = Document(
    program_id=program_id,
    filename=file.filename,
    file_type=file.content_type,
    file_size=len(content),
    file_path=str(file_path),
    content_hash=content_hash,
    status="pending"
)
await save_document(document)

# Queue indexing
background_tasks.add_task(
    index_document,
    document_id=document.id,
    program_id=program_id
)

return {
    "document_id": document.id,
    "status": "pending",
    "message": "Document uploaded. Indexing in progress."
}
```

The indexing job uses the advanced retrieval pipeline from v9. Semantic chunking respects document structure. The chunks get embedded and stored in a program-specific Qdrant collection, keeping each program's knowledge base isolated:

```
from langchain_community.document_loaders import (
    PyPDFLoader,
```

```
    UnstructuredMarkdownLoader,
    TextLoader
)
from qdrant_client import QdrantClient
from langchain_qdrant import QdrantVectorStore
from langchain_openai import OpenAIEmbeddings

async def index_document(document_id: str, program_id: str):
    """Index a document into the program's vector store."""

    document = await get_document(document_id)
    program = await get_program(program_id)

    try:
        # Update status
        document.status = "processing"
        await save_document(document)

        # Load document based on type
        file_path = document.file_path
        if document.file_type == "application/pdf":
            loader = PyPDFLoader(file_path)
        elif file_path.endswith(".md"):
            loader = UnstructuredMarkdownLoader(file_path)
        else:
            loader = TextLoader(file_path)

        raw_docs = loader.load()
```

Now chunk the loaded text. The semantic chunking strategy from Chapter 9 handles splitting on meaningful boundaries (paragraph breaks, section headers, code blocks) rather than fixed token counts:

```
        # Semantic chunking from v9
        chunks = semantic_chunk(raw_docs)

        # Add metadata
        for chunk in chunks:
            chunk.metadata["document_id"] = document_id
            chunk.metadata["program_id"] = program_id
            chunk.metadata["filename"] = document.filename
```

Embed and store the chunks. Each program's knowledge base lives in its own Qdrant collection, so switching programs switches the vector search scope automatically:

```
        # Store in program-specific collection
        embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

        vector_store = QdrantVectorStore.from_documents(
```

```
            documents=chunks,
            embedding=embeddings,
            collection_name=program.qdrant_collection,
            url="http://localhost:6333"
        )

        # Update document status
        document.status = "indexed"
        document.chunks_count = len(chunks)
        document.indexed_at = datetime.utcnow()
        await save_document(document)

    except Exception as e:
        document.status = "failed"
        document.error_message = str(e)
        await save_document(document)
        raise
```

Flashcards and Spaced Repetition

Each learning program maintains its own flashcard collection. Flashcards follow the "minimum information principle"—each card tests one atomic fact with a specific, direct question. The generation prompt explicitly avoids common pitfalls like combining multiple concepts or asking vague questions. Good flashcards look like "What error is raised when accessing a missing dictionary key with `d[key]`?" with the answer "`KeyError`," not "What is a dictionary and how do you use it?"

The flashcard model extends the spaced repetition system from earlier chapters with program scoping:

```
class Flashcard(Base):
    """Flashcard scoped to a learning program."""
    __tablename__ = 'flashcards'

    id = Column(String, primary_key=True, default=generate_uuid)
    program_id = Column(String, ForeignKey('learning_programs.id'),
nullable=False)
    topic = Column(String, nullable=False)
    question = Column(Text, nullable=False)
    answer = Column(Text, nullable=False)
    source_context = Column(Text)
    content_hash = Column(String, index=True)  # For deduplication

    # Spaced repetition fields (SM-2 algorithm)
    ease_factor = Column(Float, default=2.5)
    interval = Column(Integer, default=0)  # Days until next review
    repetitions = Column(Integer, default=0)
```

```
    next_review = Column(DateTime)

    created_at = Column(DateTime, default=datetime.utcnow)

    program = relationship("LearningProgram",
back_populates="flashcards")
```

The `due-cards` endpoint returns flashcards ready for review, filtered by program:

```
@app.get("/api/programs/{program_id}/due-cards")
async def get_due_cards(
    program_id: str,
    limit: int = 10,
    user: User = Depends(get_current_user)
):
    """Get flashcards due for review in a program."""
    program = await get_program(program_id, user.id)

    async with get_session() as session:
        result = await session.execute(
            select(Flashcard)
            .where(
                Flashcard.program_id == program_id,
                Flashcard.next_review <= datetime.utcnow()
            )
            .order_by(Flashcard.next_review)
            .limit(limit)
        )
        due_cards = result.scalars().all()

    return {"cards": due_cards, "count": len(due_cards)}


@app.post("/api/programs/{program_id}/flashcards/{card_id}/review")
async def review_flashcard(
    program_id: str,
    card_id: str,
    quality: int,  # 0-5 rating from user
    user: User = Depends(get_current_user)
):
    """Record a flashcard review and update spaced repetition
schedule."""
    await get_program(program_id, user.id)  # Verify ownership

    card = await get_flashcard(card_id, program_id)

    # SM-2 algorithm update
    if quality >= 3:  # Correct response
```

```
        if card.repetitions == 0:
            card.interval = 1
        elif card.repetitions == 1:
            card.interval = 6
        else:
            card.interval = int(card.interval * card.ease_factor)
        card.repetitions += 1
    else:  # Incorrect response
        card.repetitions = 0
        card.interval = 1

    # Update ease factor
    card.ease_factor = max(1.3, card.ease_factor + 0.1 - (5 -
quality) * 0.08)
    card.next_review = datetime.utcnow() + timedelta(days=card.interval)

    await save_flashcard(card)
    return {"next_review": card.next_review, "interval": card.interval}
```

The Full-Stack Interface

The frontend provides a complete learning experience. The main dashboard shows all learning programs with progress summaries. Each program has its own view with detailed analytics, document management, and study tools.

The program selector lets users switch between programs with a single click. Progress charts show performance over time. The document panel displays uploaded materials with indexing status. And the study interface—chat and flashcards—adapts to whichever program is currently selected.

The chat interface renders AI responses with full Markdown support (headers, lists, code blocks) and LaTeX math equations using KaTeX. This is important for technical subjects where the tutor might include formulas or code examples in explanations. The implementation uses `react-markdown` with the `remark-math` and `rehype-katex` plugins, plus a helper function to convert the LLM's `\( \)` and `\[ \]` delimiters to the `$` and `$$` format that `remark-math` expects.

The frontend uses Next.js for a production-ready React setup with server-side rendering capabilities, automatic code splitting, and seamless API routing. Initialize a new project with `npx create-next-app@latest frontend`, then install dependencies with `npm install`. The following examples use React hooks for state management and the Fetch API for backend communication:

```
function Dashboard() {
    const [programs, setPrograms] = useState([]);
    const [selectedProgram, setSelectedProgram] = useState(null);
    const [view, setView] = useState('overview'); // overview, study,
documents
```

```
    useEffect(() => {
        loadPrograms();
    }, []);

    const loadPrograms = async () => {
        const response = await fetch('/api/programs');
        const data = await response.json();
        setPrograms(data.programs);

        if (data.programs.length > 0 && !selectedProgram) {
            setSelectedProgram(data.programs[0]);
        }
    };

    return (
        <div className="dashboard">
            <Sidebar
                programs={programs}
                selectedProgram={selectedProgram}
                onSelectProgram={setSelectedProgram}
                onCreateProgram={handleCreateProgram}
            />

            <main className="main-content">
                <Header
                    program={selectedProgram}
                    view={view}
                    onViewChange={setView}
                />

                {view === 'overview' && (
                    <ProgramOverview program={selectedProgram} />
                )}

                {view === 'study' && (
                    <StudyInterface program={selectedProgram} />
                )}

                {view === 'documents' && (
                    <DocumentManager program={selectedProgram} />
                )}
            </main>
        </div>
    );
}
```

The progress dashboard shows learning analytics per program. Charts display review accuracy over time, topic mastery levels, and upcoming reviews. Users can see at a glance which subjects need attention and which they've mastered.

Since flashcards are generated in the background when a program is created, the dashboard and study interface poll for updates every few seconds while generation is in progress. Once the expected number of cards has been generated, polling stops automatically. This gives users real-time feedback as their initial flashcard set populates without requiring manual page refreshes:

```
function ProgramOverview({ program }) {
    const [stats, setStats] = useState(null);

    useEffect(() => {
        if (program) {
            loadStats(program.id);
        }
    }, [program]);

    const loadStats = async (programId) => {
        const response = await fetch(`/api/programs/${programId}/
stats`);
        const data = await response.json();
        setStats(data);
    };

    if (!stats) return <LoadingSkeleton />;

    return (
        <div className="program-overview">
            <div className="stats-grid">
                <StatCard
                    title="Topics Studied"
                    value={stats.topicsStudied}
                    total={stats.totalTopics}
                />
                <StatCard
                    title="Flashcards Reviewed"
                    value={stats.cardsReviewed}
                    subtitle="this week"
                />
                <StatCard
                    title="Accuracy"
                    value={`${stats.accuracy}%`}
                    trend={stats.accuracyTrend}
                />
                <StatCard
                    title="Due for Review"
                    value={stats.dueForReview}
                    urgent={stats.dueForReview > 20}
                />
            </div>
```

```
            <div className="charts-row">
                <ProgressChart data={stats.progressHistory} />
                <TopicMasteryChart data={stats.topicMastery} />
            </div>

            <UpcomingReviews reviews={stats.upcomingReviews} />
        </div>
    );
}
```

Streaming Responses

Long generation tasks—creating flashcards for a new topic, generating a curriculum, or answering complex questions—use HTTP streaming with SSE to keep users engaged. The interface shows what's happening in real time rather than making users wait for completion. SSE is simpler than WebSockets for unidirectional streaming and works well with serverless platforms like Vercel.

The chat endpoint retrieves context from the program's knowledge base and streams the LLM response:

```
from fastapi.responses import StreamingResponse
from langchain_openai import ChatOpenAI
import json

@app.post("/api/programs/{program_id}/chat")
async def chat_with_tutor(
    program_id: str,
    request: ChatRequest,
    db: Session = Depends(get_db_dependency),
):
    """Chat with the AI tutor using HTTP streaming."""
    user_id = get_current_user_id()
    program = await get_program_or_404(program_id, user_id, db)

    # Initialize retriever for this program's knowledge base
    retriever = get_program_retriever(program)

    # Retrieve context
    docs = retriever.search(request.message, k=3)
    context = "\n\n".join(doc.page_content for doc in docs) if docs
else ""

    # Build system prompt with retrieved context
    system_prompt = f"""You are a helpful tutor for: {program.name}

Use the following context to answer questions accurately:
```

```
{context}

If the context doesn't contain relevant information, say so honestly."""

    # Build messages from conversation history
    messages = [("system", system_prompt)]
    for msg in request.history:
        messages.append((msg.role, msg.content))
    messages.append(("user", request.message))

    # Create streaming LLM
    llm = ChatOpenAI(model="gpt-4o", streaming=True)

    async def generate():
        """Stream the response chunks as SSE."""
        async for chunk in llm.astream(messages):
            if chunk.content:
                # JSON-encode to preserve newlines in SSE format
                yield f"data: {json.dumps(chunk.content)}\n\n"
        yield "data: [DONE]\n\n"

    return StreamingResponse(
        generate(),
        media_type="text/event-stream",
        headers={
            "Cache-Control": "no-cache",
            "Connection": "keep-alive",
        },
    )
```

The frontend consumes this stream by reading chunks and JSON-parsing each one to restore formatting:

```
const sendMessage = async () => {
    const message = input.trim();
    setMessages(prev => [...prev, { role: 'user', content: message }]);
    setIsLoading(true);

    try {
        const response = await fetch(`/api/programs/${program.id}/
chat`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ message, history: messages }),
        });

        const reader = response.body.getReader();
        const decoder = new TextDecoder();
        let assistantMessage = '';
```

```
            // Add placeholder message for streaming
            setMessages(prev => [...prev, { role: 'assistant', content:
'' }]);

            while (true) {
                const { done, value } = await reader.read();
                if (done) break;

                const chunk = decoder.decode(value);
                const lines = chunk.split('\n');

                for (const line of lines) {
                    if (line.startsWith('data: ')) {
                        const data = line.slice(6);
                        if (data === '[DONE]') break;

                        // Parse JSON to restore newlines
                        try {
                            assistantMessage += JSON.parse(data);
                        } catch {
                            assistantMessage += data;
                        }

                        // Update the streaming message
                        setMessages(prev => {
                            const newMessages = [...prev];
                            newMessages[newMessages.length - 1] = {
                                role: 'assistant',
                                content: assistantMessage,
                            };
                            return newMessages;
                        });
                    }
                }
            }
        } finally {
            setIsLoading(false);
        }
    };
```

The JSON encoding is a subtle but important detail. Raw newlines in SSE data break the protocol (each `data:` line must be a single line). By JSON-encoding each chunk on the server and parsing it on the client, we preserve the LLM's formatting, including line breaks, code blocks, and bullet points.

User-Scoped Data Isolation

Even though full authentication isn't necessary, we design for user isolation anyway. Every database query filters by `user_id`. Every Qdrant collection is

namespaced by program. This means adding authentication later, if required, would be a configuration change, not a rewrite:

```
async def get_user_programs(user_id: str) -> list[LearningProgram]:
    """Get all programs for a user."""

    async with get_session() as session:
        result = await session.execute(
            select(LearningProgram)
            .where(LearningProgram.user_id == user_id)
            .order_by(LearningProgram.updated_at.desc())
        )
        return result.scalars().all()

async def get_program(
    program_id: str,
    user_id: str
) -> LearningProgram:
    """Get a specific program, verifying ownership."""

    async with get_session() as session:
        result = await session.execute(
            select(LearningProgram)
            .where(
                LearningProgram.id == program_id,
                LearningProgram.user_id == user_id
            )
        )
        program = result.scalar_one_or_none()

        if not program:
            raise HTTPException(
                status_code=404,
                detail="Program not found"
            )

        return program
```

Running the Application

With all the pieces in place, here's how to run StudyBuddy v10. The setup assumes you have PostgreSQL and Qdrant running locally.

First, install the dependencies:

```
cd v10-full-stack

# Install Python dependencies
uv sync
```

```
# Install frontend dependencies
cd frontend
npm install
cd ..
```

Configure your environment by creating a `.env` file:

```
# .env
OPENAI_API_KEY=sk-your-key-here
POSTGRES_URL=postgresql://localhost/studybuddy
QDRANT_URL=http://localhost:6333
```

Start Qdrant if it's not already running:

```
docker run -p 6333:6333 qdrant/qdrant
```

The database tables are created automatically on startup. Open a second terminal and run the following:

```
# Check if database exists
psql -l | grep studybuddy

# If no output, create it
createdb studybuddy
```

Now start the API server in the second terminal you just opened:

```
# Terminal 2: Start the API server
uv run uvicorn api.index:app --reload --port 8000
```

Finally, open a third terminal and start the frontend development server:

```
# Terminal 3: Start the frontend dev server
cd frontend
npm run dev
```

Open `http://localhost:3000` in your browser. You should see the StudyBuddy interface ready to create your first learning program.

Deploying to Vercel

To deploy v10 to Vercel, you need to update two files in the repo root.

First, update `vercel.json` to point to the `v10-full-stack` directory:

```
{
    "version": 4,
    "builds": [
        {
            "src": "v10-full-stack/api/index.py",
```

```
            "use": "@vercel/python"
        },
        {
            "src": "v10-full-stack/frontend/package.json",
            "use": "@vercel/next"
        }
    ],
    "routes": [
        { "src": "/api/(.*)", "dest": "/v10-full-stack/api/index.py" },
        { "src": "/(.*)", "dest": "/v10-full-stack/frontend/$1" }
    ],
    "git": {
        "deploymentEnabled": {
            "main": true
        }
    },
    "ignoreCommand": "bash scripts/should-deploy.sh"
}
```

Second, update `scripts/should-deploy.sh` to trigger on `v10` changes:

```
#!/bin/bash
# Deploy if changes are in v10-full-stack or vercel config
# Exit 1 (true) = proceed with build, Exit 0 (false) = skip build
git diff HEAD^ HEAD --name-only | grep -qE "^v10-full-stack/|^vercel\
.json" && exit 1 || exit 0
```

If you haven't done so already, set your OpenAI API key environment variable in Vercel by running the following command from the project root and pasting the key when prompted:

```
vercel env add OPENAI_API_KEY
```

Then redeploy:

```
vercel --prod
```

Or simply push to GitHub. Vercel will automatically deploy when changes are detected in the `v10-full-stack` directory.

To create a Qdrant Cloud instance:

1. Go to `cloud.qdrant.io`.
2. Sign up or log in.
3. In the Clusters tab of your dashboard, click Create.
4. Choose the Free tier.
5. Select a region close to your Vercel deployment.

Once created, click your cluster to see:

- **URL:** In the Overview tab, the Endpoint value shows your cluster address, which will look something like `https://your-cluster-id.aws.cloud.qdrant.io`.
- **API key:** Click the API Keys tab and then click Create to generate a new API key.

Then add these to Vercel's Environment Variables:

1. Go to your project, click Settings, then click Environment Variables.
2. Click Add Environment Variable, type **QDRANT_URL** in the Key field, paste your Qdrant Cloud URL in the Value field, then click Save.
3. Click Add Environment Variable, type **QDRANT_API_KEY** in the Key field, paste your Qdrant Cloud API key in the Value field, then click Save.

Deploy online or at the command line:

```
vercel --prod
```

Or simply push to GitHub. Vercel will automatically deploy when changes are detected in the `v10-full-stack` directory.

What You've Built

You've now built a full-stack AI application from the ground up. The agent layer from previous chapters, the retrieval infrastructure from Chapter 9, the UI patterns from this chapter all come together into something you could actually ship. And that's the whole point: build, ship, share.

Keep building. Keep shipping. Keep sharing.

APPENDIX

A

Setting Up Your Dev Environment

Here's the deal: You can't build AI applications without the right tools. This appendix is your complete guide to setting up a professional AI engineering development environment from scratch. We're talking about the full stack: your code editor, terminal, Python, version control, API access, containerization, and everything you need to build, ship, and share.

No fluff. No assumptions about what you already have. We're starting from zero and building up to a production-grade setup that'll serve you through this entire book and beyond.

Let's get started.

What We're Building

By the end of this appendix, you'll have:

- An AI-powered code editor (Cursor or VS Code) configured for Python development
- Familiarity with the command-line interface (CLI) on your operating system
- Git and GitHub set up for version control
- Python 3.12+ with a proper virtual environment system (using `uv`)
- Essential Python packages installed (`numpy`, `pandas`, `jupyter`, and more)

- API keys configured for OpenAI, Anthropic Claude, and Hugging Face
- Docker installed for containerization
- Jupyter notebooks ready to use

This is the baseline. Every professional AI engineer has this setup, and now you will too.

Setting Up Your Code Editor

Your code editor is where you'll spend most of your time. We recommend Cursor, which is built on VS Code and adds AI-assisted development features. Think of it as VS Code's cooler, smarter sibling.

Option A: Cursor (Recommended)

Cursor is a fork of VS Code built by Anysphere. It gives you AI assistance right in your editor, which is useful when you're learning AI engineering. You get code completion, explanations, and debugging help, all powered by LLMs.

Follow these steps to install Cursor:

1. Go to `cursor.com`.
2. Download the installer for your operating system.
3. Run the installer.
4. Launch Cursor.

That's it. You're in.

Why Cursor? Because building AI applications while having AI help you code is like having a senior engineer looking over your shoulder. The AI can explain code, suggest fixes, and even help you write entire functions. When you're learning a new domain like AI engineering, that's valuable.

Option B: VS Code (Also Great)

If you prefer vanilla VS Code or already have it configured, that works too. Everything in this book will work in either editor.

Here's how to install VS Code:

1. Go to `code.visualstudio.com`.
2. Download the installer for your operating system.
3. Run the installer.
4. Launch VS Code.

The core development workflow is identical in both editors. Cursor simply adds AI assistance on top. If you need AI help in VS Code, you can always install GitHub Copilot, as we explain next.

Essential Extensions

Regardless of which editor you choose, there are a few extensions that are especially useful for AI dev work. They transform your editor from a text editor into a more capable Python development environment. Here are the extensions you need:

- **GitHub Copilot (VS Code Only):** GitHub Copilot is VS Code's AI-powered assistant for code completion, code explanations, code debugging, and more. If you have Cursor envy, Copilot will fix that.
- **Python:** This is the foundation for Python development. It gives you syntax highlighting, debugging, linting, IntelliSense, and more.
- **Jupyter:** Since we'll be using Jupyter notebooks extensively for prototyping and experimentation, you need this.

To install these:

1. Open your editor.
2. Click the Extensions icon in the left sidebar (it looks like four squares, with one of them askew) or press Ctrl+Shift+X (Cmd+Shift+X on macOS).
3. Type the name of the extension in the search bar.
4. Click the extension in the search results. You'll see many results, so be sure to click the right one. GitHub Copilot is published by GitHub, and both the Python and Jupyter extensions are published by Microsoft.
5. On the extension page that appears, click Install.

Now you're set. Your editor is ready to handle Python and Jupyter notebooks effectively.

Getting to Know Your Terminal

Here's something that surprises a lot of people coming from no-code backgrounds: AI engineering requires comfort with the terminal. You can't build most production AI systems by clicking around GUIs. You need the command line.

Don't worry if you've never used it. We'll get you up to speed.

What's a Terminal?

A terminal is a text-based interface for controlling your computer. Instead of clicking icons, you type commands. It feels old-school because it is, but it's also very useful. AI engineers often work in the terminal as part of their daily workflow.

Getting to Your Terminal

How you access your terminal depends on your OS:

- **macOS:** You have a built-in terminal app called, no surprises here, Terminal. Press Cmd+Space to open Spotlight, type **Terminal**, then press Return. Boom. You're in.
- **Windows:** Right-click Start and then click Terminal. If you don't see Terminal, you can install it: open the Microsoft Store, search for **Windows Terminal**, then install it. Pin it to your taskbar because you'll use it often. (What about WSL? Windows Subsystem for Linux (WSL) is useful for AI engineering because it gives you a Linux environment on Windows. If you're comfortable setting it up, go for it. But it's not required for this book. Windows Terminal will get you there.)
- **Linux:** You're already set. Your distribution comes with a terminal. Press Ctrl+Alt+T to open it.

Understanding the Command Line

When you open your terminal, you see a prompt. It looks something like this:

```
yourname@computer:~$
```

This is where you type commands. The terminal is waiting for instructions. Here are some basic commands everyone should know:

- `pwd`: Print working directory. Shows where you are in your file system.
- `ls` (Mac/Linux) or `dir` (Windows): Lists files in the current directory.
- `cd` *`foldername`*: Change directory. Moves you into a folder.
- `cd ..`: Moves up one directory level.
- `mkdir` *`foldername`*: Makes a new directory.
- `clear` (Mac/Linux) or `cls` (Windows): Clears your terminal screen.

Try them out. Get comfortable moving around. You'll be using the terminal for Git commands, Python environments, package installations, and running your applications.

Git and GitHub: Version Control

Version control is nonnegotiable. You need to track changes to your code, collaborate with others, and deploy to production. Git is the industry standard, and GitHub is where AI engineers share code.

Installing Git

Here's how to get Git installed, based on your OS:

- **macOS:** Open Terminal and type:

```
git --version
```

- If Git isn't installed, macOS will prompt you to install Xcode Command Line Tools, which includes Git. Follow the prompts.
- **Windows:** Go to `git-scm.com`, download the Windows installer, run it, and use the default settings.
- **Linux:** In Ubuntu/Debian, run these commands:

```
sudo apt-get update

sudo apt-get install git
```

- In Fedora, run this command:

```
sudo dnf install git
```

Configuring Your Git Identity

Time to tell Git who you are! This is like setting up your developer ID card. Your Git identity is used to

- Identify you as the author of your commits (adds your name to each change)
- Show up in commit history (your contributions are properly attributed)
- Connect your work to your GitHub profile (when the email address matches one associated with that profile)
- Help other developers know who to contact about changes

At your terminal, run the following commands:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

Be sure to replace the placeholders with your name and email address. Your name can be your real name or a pseudonym, but note that it can be publicly visible in the commit history, so use a name you're comfortable making public.

Your email should match an email address associated with your GitHub account for proper attribution. For work projects, use your work email. For personal projects, use the email linked to your GitHub account.

The preceding commands apply to every repository. If you prefer to set up a different identity for a specific repo, `cd` into the repo directory and then run these commands instead:

```
git config user.name "Your Name"
git config user.email "your.email@example.com"
```

Setting Up GitHub Authentication

GitHub no longer supports password authentication for Git operations. Instead, you can use one of the following authentication methods:

- **Personal Access Token (PAT):** A 40-character string that's a secure alternative credential to a password and is used for authenticating with GitHub at the terminal, with the GitHub API, or with other third-party applications. The token expires after a specified period (we recommend 90 days), after which you must regenerate the token.
- **SSH key:** Uses the Secure Shell (SSH) protocol and a pair of cryptographic keys (public and private) to establish a secure, encrypted connection between your local computer and GitHub's servers. This key never expires.

Pick whichever one you prefer. Setting up SSH access is a bit more involved, but it's a set-it-and-forget-it operation; going the PAT route is easier, but you need to regenerate your token every so often.

Creating a Personal Access Token

Here's how to create a personal access token:

1. Log in to GitHub.
2. Click your profile photo at the top right, then click Settings.
3. In the left sidebar, near the bottom, click Developer Settings.
4. Click Personal Access Tokens ➪ Tokens (Classic).
5. Click Generate New Token ➪ Generate New Token (Classic).
6. Type a name, such as **AI Engineering Dev**.
7. Set the expiration (we recommend 90 days; you can always create a new one).
8. Check every check box (this gives full repository access).
9. Scroll down and click Generate Token.

With your new PAT displayed, click the Copy icon to make a copy. It's vital that you do this immediately because GitHub will never show it to you again. Paste the token somewhere safe for now.

Security note: Treat this token like a password. Don't commit it to a repository. Don't share it in screenshots. If you accidentally expose it, revoke it immediately on GitHub and generate a new one.

The way the PAT works normally is that when you run a `git` command that requires talking to GitHub over the Internet (such as `git clone` or `git fetch`), you get prompted first for your GitHub username:

```
Username for 'https://github.com':
```

Type your username, press Enter or Return, and then GitHub asks for your password:

```
Password for 'https://username@github.com':
```

Your password is actually your PAT, so copy the PAT from wherever you stored it, paste it after the `Password` prompt (for security, you won't see anything appear when you paste it, but you're good to go), and then press Enter or Return.

Wait a minute, we hear you saying, I have to do that *every time* I run a remote `git` command? Well, normally, yeah, but that's not how we roll around here. Instead, you can configure credential storage, which means your OS will store your GitHub username and PAT so you don't have to enter them every time. The command you run depends on your OS:

```
# For macOS (stores in Keychain—your digital vault)
git config --global credential.helper osxkeychain

# For Windows (stores in Credential Manager—your Windows safe)
git config --global credential.helper wincred

# For Linux (stores in memory—your temporary sticky note)
git config --global credential.helper cache
# Note that this is temporary. If you want a permanent secure option
# for Linux, you need to install and configure a credential helper.
```

With that done, the next time you run a remote `git` command, you'll be prompted again for your username and PAT. This time, however, your credentials will be stored, which means you generally won't have to enter them again until your PAT expires or the stored credentials are removed.

Setting Up SSH Access

HTTPS works fine for cloning repos, but SSH gives you passwordless authentication after the initial setup. You authenticate once with your key, and you're good to go. It's more secure, more convenient, and widely used in professional

development workflows. Every serious developer has SSH keys configured. Now you will too.

Here's the workflow: generate an SSH key pair on your machine, add the private key to your SSH agent, and then tell GitHub about your public key. The exact steps depend on your OS, so here we go.

First, if you're using Windows, open the Git Bash app (which was installed when you installed Git).

Step 1: Check for Existing SSH Keys

It's a good idea to first check whether you already have SSH keys on your machine:

```
ls -la ~/.ssh
```

If you see files like `id_ed25519` and `id_ed25519.pub` (or the older `id_rsa` and `id_rsa.pub`), you've already got keys. You can skip to Step 3 if you want to use those existing keys.

If the directory doesn't exist or is empty, no worries. You'll create fresh keys in the next step.

Step 2: Generate a New SSH Key

Run this command, replacing the email with your GitHub email address:

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

The Ed25519 algorithm is the current recommendation because it's fast and secure.

When prompted for a file location, just press Enter or Return to accept the default (`~/.ssh/id_ed25519`).

You'll be asked for a passphrase. This is optional but recommended. It adds an extra layer of security in case someone gains access to your private key file. Pick something memorable, or leave it blank if you prefer convenience over that extra protection.

Step 3: Start the SSH Agent

The SSH agent runs in the background and manages your keys. Start it up:

```
eval "$(ssh-agent -s)"
```

You should see output like `Agent pid 12345`. That means it's running.

Step 4: Configure SSH (macOS Only)

If you're on macOS, you'll want to modify your SSH config file so the agent automatically loads your key and uses the macOS Keychain to store your passphrase.

Check if the config file exists:

```
open ~/.ssh/config
```

If it doesn't exist, create it:

```
touch ~/.ssh/config
```

Add this configuration:

```
Host github.com
    AddKeysToAgent yes
    UseKeychain yes
    IdentityFile ~/.ssh/id_ed25519
```

This tells SSH to automatically add the key to the agent and store your passphrase in the Keychain. Super convenient.

Step 5: Add Your Key to the SSH Agent

Now let's add your private key to the agent.

macOS:

```
ssh-add --apple-use-keychain ~/.ssh/id_ed25519
```

The `--apple-use-keychain` flag stores your passphrase in the macOS Keychain so you don't have to enter it every time.

Windows and Linux:

```
ssh-add ~/.ssh/id_ed25519
```

If you set a passphrase, you'll enter it here.

Step 6: Auto-Start the SSH Agent (Optional but Recommended)

Having to start the SSH agent manually every time you open a new terminal gets old fast. Here's how to automate it.

macOS: If you completed Step 4, you're already covered. The config file handles this for you.

Windows (Git Bash): Open or create the file `~/.bashrc` and add the following:

```
env=~/.ssh/agent.env
```

```
agent_load_env () { test -f "$env" && . "$env" >| /dev/null ; }

agent_start () {
    (umask 077; ssh-agent >| "$env")
    . "$env" >| /dev/null ; }

agent_load_env

# agent_run_state: 0=agent running w/ key; 1=agent w/o key; 2=agent
not running
agent_run_state=$(ssh-add -l >| /dev/null 2>&1; echo $?)

if [ ! "$SSH_AUTH_SOCK" ] || [ $agent_run_state = 2 ]; then
    agent_start
    ssh-add
elif [ "$SSH_AUTH_SOCK" ] && [ $agent_run_state = 1 ]; then
    ssh-add
fi

unset env
```

Linux: Add this to your `~/.bashrc` or `~/.zshrc`:

```
if [ -z "$SSH_AUTH_SOCK" ]; then
    eval "$(ssh-agent -s)"
    ssh-add ~/.ssh/id_ed25519
fi
```

Then reload your shell config:

```
source ~/.bashrc   # or ~/.zshrc for zsh users
```

Step 7: Copy Your Public Key

Time to grab your public key so you can give it to GitHub.

macOS:

```
pbcopy < ~/.ssh/id_ed25519.pub
```

Love that `pbcopy` command. Your key is now on your clipboard.

Windows (Git Bash):

```
cat ~/.ssh/id_ed25519.pub | clip
```

Linux: If you have `xclip` installed**:**

```
xclip -selection clipboard < ~/.ssh/id_ed25519.pub
```

If you have `xsel` instead:

```
xsel --clipboard < ~/.ssh/id_ed25519.pub
```

Don't have either? Install `xclip` with your package manager:

```
# Ubuntu/Debian
sudo apt install xclip

# Fedora
sudo dnf install xclip

# Arch
sudo pacman -S xclip
```

Or just display and copy manually:

```
cat ~/.ssh/id_ed25519.pub
```

Select the output and copy it the old-fashioned way. Works fine.

Step 8: Add the Key to GitHub

1. Go to `github.com` and sign in.
2. Click your profile picture in the top right, then click Settings.
3. In the left sidebar, click SSH and GPG Keys.
4. Click the green New SSH Key button.
5. Give your key a descriptive title (such as MacBook Pro, Ubuntu Workstation, or Work PC).
6. Paste your key into the "Key" field (it's already on your clipboard).
7. Click Add SSH Key.

Done. GitHub now knows about your public key.

Step 9: Test the Connection

Let's make sure everything is working:

```
ssh -T git@github.com
```

You might see a warning about the authenticity of the host the first time. Type **yes** and press Enter or Return to continue.

If everything is set up correctly, you'll see:

```
Hi username! You've successfully authenticated, but GitHub does not
provide shell access.
```

That's exactly what you want. You're in.

Switching a Repository from HTTPS to SSH

Already have repos cloned with HTTPS? No problem. You can switch them to use SSH.

First, check your current remote URL:

```
git remote -v
```

If it shows HTTPS (such as `https://github.com/username/repo.git`), update it to SSH:

```
git remote set-url origin git@github.com:username/repo.git
```

Verify the change:

```
git remote -v
```

Now you should see the SSH URL (`git@github.com:username/repo.git`). All future pushes and pulls will use SSH authentication.

Basic Git Workflow

You'll use these commands constantly:

```
# Clone a repository
git clone https://github.com/username/repo-name.git

# Check status
git status

# Stage changes
git add .

# Commit changes
git commit -m "Description of changes"

# Push to GitHub
git push

# Pull latest changes
git pull

# Create a new branch
git checkout -b branch-name

# Switch branches
git checkout branch-name
```

We'll dive deeper into Git workflows in the book. For now, just get comfortable with the basics.

Essential .gitignore

You don't want certain files getting pushed to your repo. To tell Git what to avoid, create a `.gitignore` file in your project root and add the following to it:

```
# Environment
.env
.venv/
venv/
env/

# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python

# Jupyter
.ipynb_checkpoints/
*.ipynb_checkpoints

# IDE
.vscode/
.idea/
*.swp
*.swo

# OS
.DS_Store
Thumbs.db

# Data (if large)
data/raw/*
data/processed/*
!data/raw/.gitkeep
!data/processed/.gitkeep
```

This prevents you from committing sensitive files or large data files to Git.

Setting Up Python with uv

Python is the language of AI engineering. You need Python 3.12 or later, and you need a proper way to manage dependencies and virtual environments.

We're using `uv`, a blazing-fast Python package installer and environment manager built in Rust. It's like `pip` and `venv` on steroids.

Installing uv

First, let's get `uv` onto your machine:

macOS and Linux:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

After installation completes, close and reopen your terminal, then verify the installation:

```
uv --version
```

Windows:

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install
.ps1 | iex"
```

Close and reopen your terminal, then verify:

```
uv --version
```

Creating Your First Virtual Environment

Virtual environments isolate your project's dependencies. This prevents conflicts between projects. You might have one project using LangChain 1.0.0 and another using LangChain 1.1.0. Virtual environments keep them separate.

Creating a Virtual Environment with Python 3.12

```
uv venv .venv -p 3.12
```

This creates a folder called `.venv` in your current directory that contains an isolated Python environment.

Activating the Virtual Environment

macOS/Linux:

```
source .venv/bin/activate
```

Windows (PowerShell):

```
.venv\Scripts\Activate.ps1
```

Windows (Command Prompt):

```
.venv\Scripts\activate.bat
```

You'll see `(.venv)` appear at the start of your terminal prompt. That means you're in the virtual environment.

Deactivating When You're Done

```
deactivate
```

Installing Essential Packages

With your virtual environment activated, install the core packages you'll need:

```
uv pip install numpy pandas matplotlib jupyter openai anthropic
huggingface_hub langchain langchain-openai python-dotenv
```

What you just installed:

- **numpy:** Scientific computing and numerical operations
- **pandas:** Data manipulation and analysis
- **matplotlib:** Data visualization and plotting
- **jupyter:** Interactive notebooks for experimentation
- **openai:** Official OpenAI Python client
- **anthropic:** Official Anthropic/Claude Python client
- **huggingface_hub:** Tools for interacting with Hugging Face models and datasets
- **langchain:** The LangChain LLM orchestration framework
- **langchain-openai:** LangChain integration with OpenAI
- **python-dotenv:** Reads the contents of `.env`, which is where you store secrets such as your OpenAI API key

This is your baseline toolkit. We'll add more packages as needed throughout the book.

Configuring Your Editor to Use the Virtual Environment

Now that you have a virtual environment, tell your editor to use it.

In Cursor or VS Code, follow these steps:

1. Open your project folder in the editor.
2. Press Cmd+Shift+P (macOS) or Ctrl+Shift+P (Windows/Linux).
3. Type **Python: Select Interpreter**.
4. Choose the interpreter from your `.venv` folder. It'll look something like `.venv/bin/python` or `.venv\Scripts\python.exe`.

Now your editor will use the packages installed in your virtual environment. Code completion will work. Imports will resolve. Everything is connected.

Getting API Keys for LLM Providers

To build AI applications, you need programmatic access to large language models (LLMs). That means you need to get an API key from each of the major providers.

OpenAI API Key

OpenAI gives you access to the latest models through its API. Here's how to create an API key:

1. Go to `platform.openai.com`.
2. Sign up or log in.
3. Click your profile icon (top right), then click Your profile.
4. Click Projects, click Create, type a project name, and then click Create.
5. Click API Keys.
6. Click Create New Secret Key.
7. Type a name, such as **AI Engineering Dev**.
8. Use the Project list to select the project you created in Step 4.
9. Click Create Secret Key.
10. Copy the key immediately (you won't see it again), then refer to the Storing API Keys Securely section to store it.

Important: OpenAI isn't free. You need to add a payment method (Billing ➪ Payment Methods) and credits to your account. Start with $5–$10. That's enough for development and learning. Monitor your usage in the dashboard.

Best practice: Set spending limits. Go to Billing ➪ Usage Limits and set a monthly cap. This prevents surprise bills if you accidentally create an infinite loop that repeatedly calls the API.

Anthropic API Key

Anthropic's Claude models are powerful for reasoning and analysis tasks and are widely considered (at least as we write this) the best models for generating code. Here's how to create an Anthropic API key:

1. Go to `console.anthropic.com`.
2. Sign up or log in.
3. Click Get API Key.
4. Click Create Key.

5. Name your key, then click Add.
6. Copy the key right away (Anthropic won't show it to you again), then check out the Storing API Keys Securely section to store it.

Important: Anthropic requires a minimum $5 credit purchase before you can use the API. It's prepaid, not billed monthly. Go to Billing, then click Buy Credits.

Hugging Face Access Token

Hugging Face hosts thousands of open-source models and datasets. You'll use it frequently. Here's how to create an access token:

1. Go to `huggingface.co`.
2. Sign up or log in.
3. Click Settings in the left sidebar.
4. Click Access Tokens.
5. Click Create New Token.
6. Give it a name and click Write Access (you'll need this for deploying to Spaces).
7. Click Create Token.
8. When the key appears, copy it, then check out the Storing API Keys Securely section to store it.

Storing API Keys Securely

Never hardcode API keys in your code, and never commit them to Git. Use environment variables instead.

Storing Local Environment Variables

For local development, you can create a file called `.env` in your project root, then add your keys:

```
OPENAI_API_KEY=sk-proj-...
ANTHROPIC_API_KEY=sk-ant-...
HF_TOKEN=hf_...
```

Critical: As we mentioned earlier (refer to the "Essential .gitignore" section), it's vital to add `.env` to your `.gitignore` file so it never gets committed. If you haven't done so already, you can run the following command:

```
echo ".env" >> .gitignore
```

Now you can load any environment variable in your Python code. For example, the following code loads the `OPENAI_API_KEY` environment variable:

```
from dotenv import load_dotenv
import os

load_dotenv()

openai_key = os.getenv("OPENAI_API_KEY")
```

Storing System Environment Variables

To enable access to your keys system-wide, you can store them as system environment variables.

macOS/Linux:
Add them to your shell configuration file (`~/.bashrc`, `~/.zshrc`, etc.):

```
export OPENAI_API_KEY="sk-proj-..."
export ANTHROPIC_API_KEY="sk-ant-..."
export HF_TOKEN="hf_..."
```

Then reload your shell:

```
source ~/.bashrc  # or ~/.zshrc
```

Windows (PowerShell):

```
[Environment]::SetEnvironmentVariable("OPENAI_API_KEY", "sk-proj-...",
"User")
[Environment]::SetEnvironmentVariable("ANTHROPIC_API_KEY", "sk-ant-...",
"User")
[Environment]::SetEnvironmentVariable("HF_TOKEN", "hf_...", "User")
```

Restart your terminal for these changes to take effect.

Setting Up Docker for Containerization

Docker lets you package your application with all its dependencies into a container. This makes deployment consistent and reliable. You build it once, and it runs everywhere.

Installing Docker Desktop

macOS:

1. Go to `docker.com/products/docker-desktop`.
2. Click Download Docker Desktop, then click which version you need (Intel or Apple Silicon).
3. Open the downloaded `.dmg` file.
4. Drag Docker to Applications.
5. Launch Docker Desktop.
6. Follow the setup wizard.
7. Docker will ask for permissions—grant them.

Windows:

1. Go to `docker.com/products/docker-desktop`.
2. Click Download Docker Desktop, then click which version you need for your PC.
3. Run the installer. *Important*: When asked, make sure Use WSL 2 Instead of Hyper-V is checked (WSL 2 is faster and more modern).
4. When the installation is complete, click Close and Restart.
5. Follow the prompts to configure Docker Desktop.

Windows-specific note: If you're not an admin, you need to be added to the `docker-users` group:

1. Press Windows+R to open the Run dialog, type **lusrmgr.msc**, then click OK.
2. Click Groups.
3. Right-click docker-users and then click Add to Group.
4. Click Add, type your user name, then click OK.
5. Sign out and back in.

Linux (Ubuntu/Debian):

```
# Update package index
sudo apt-get update

# Install prerequisites
sudo apt-get install ca-certificates curl gnupg lsb-release
```

```
# Add Docker's official GPG key
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg
--dearmor -o /etc/apt/keyrings/docker.gpg

# Set up the repository
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/
docker.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker
.list > /dev/null

# Install Docker Engine
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
docker-compose-plugin

# Start Docker
sudo systemctl start docker
sudo systemctl enable docker

# Add your user to docker group (so you don't need sudo)
sudo usermod -aG docker $USER
```

Log out and back in for the group changes to take effect.

Verifying Docker Installation

Run these commands to check that Docker was installed properly:

```
docker --version
docker compose version
```

Try running a test container:

```
docker run hello-world
```

If you see `Hello from Docker!` followed by a long message, the installation is working.

Docker Basics

You don't need to be a Docker expert for this book, but here are the essentials:

- **Dockerfile:** A text file with instructions for building your container image. Think of it as a recipe.
- **Image:** A snapshot of your application with all dependencies, built from a Dockerfile.

- **Container:** A running instance of an image.
- **Common commands:**

```
# Build an image
docker build -t myapp .

# Run a container
docker run -p 8000:8000 myapp

# List running containers
docker ps

# Stop a container
docker stop container_id

# List images
docker images

# Remove an image
docker rmi image_id
```

We'll cover Docker in depth when we get to deployment. For now, just having it installed is enough.

Jupyter Notebooks

Jupyter notebooks are essential for AI engineering. They let you write code in cells, run them independently, and see results inline. Perfect for experimentation and data exploration.

Installing and Running Jupyter

You already installed Jupyter with `uv` earlier, but let's make sure everything's working.

With your virtual environment activated, run this command:

```
jupyter notebook
```

This launches Jupyter in your browser, usually at `http://localhost:8888/tree`. You'll see a file explorer showing your current directory.

To create a new notebook:

1. Click New (top right).
2. Click Python 3.

Just like that, you're now in a notebook. Each rectangle is a "cell" where you write code. Here are some basic operations to get you started:

- **Run a cell:** Press Shift+Enter.
- **Add a cell below:** Press B.
- **Add a cell above:** Press A.
- **Delete a cell:** Press D twice.
- **Change to Markdown:** Press M.
- **Change to Code:** Press Y.

Try it:

```
import numpy as np
import pandas as pd

print("Jupyter is working!")
```

Run the cell. If it prints `Jupyter is working!`, you're set.

JupyterLab (Optional but Recommended)

JupyterLab is the next-generation interface for Jupyter. It's more powerful and flexible.

Install it:

```
uv pip install jupyterlab
```

Launch it:

```
jupyter lab
```

Same concepts, a better interface. You can have multiple notebooks, terminals, and text editors open side by side.

Project Structure—Organizing Your Work

Good project structure makes your life easier. Here's a template for AI engineering projects:

```
project-name/
├── .env                    # API keys (never commit!)
├── .gitignore              # Files to ignore in Git
├── README.md               # Project documentation
├── pyproject.toml          # Project dependencies (uv format)
├── requirements.txt        # Alternative dependencies format
```

```
├── data/                # Data files
│   ├── raw/
│   └── processed/
├── notebooks/           # Jupyter notebooks
│   └── exploration.ipynb
├── src/                 # Source code
│   ├── __init__.py
│   ├── agents/
│   ├── retrieval/
│   └── utils.py
├── tests/               # Test files
│   └── test_agents.py
└── .venv/               # Virtual environment (never commit!)
```

Your First Integration Test

Run the following in a Jupyter notebook or Python file to verify that everything's connected:

```
import os
from dotenv import load_dotenv
from openai import OpenAI

# Load environment variables
load_dotenv()

# Initialize client
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

# Simple test
response = client.responses.create(
    model="gpt-5.6-luna",
    input="Say 'Setup complete!'",
)

print(response.output_text)
```

If this prints `Setup complete!` (or something similar), you're ready to build. Great job!

APPENDIX

B

Twelve Projects to Build, Ship, and Share

Okay, you're about to spend weeks leveling up your AI engineering skills, and the best way to learn is by building something *you* actually care about. ResearchBot is our main example throughout the book—but you should absolutely build your own thing.

Not sure what to build? No stress. We've got you covered with 12 solid project ideas that'll work great as you follow along chapter by chapter. Each one starts simple but has plenty of room to grow into a full production system by the end.

Pick one that resonates with you, or use these as inspiration to come up with something completely different. The goal is to have a project you're genuinely excited to work on. That excitement is fuel.

MeetingMinder: Your Intelligent Meeting Summarizer

MeetingMinder transforms messy meeting transcripts into structured, actionable summaries. Upload a transcript from Zoom, Teams, or Google Meet, and get back an executive summary, key decisions, action items with owners, discussion topics, and follow-up questions. No more "Wait, what did we decide?" moments.

Why it works for this book: You'll add document retrieval for company context, multi-agent systems for analyzing different aspects, memory for tracking recurring themes, and production deployment.

Starter prompt:

```
You are MeetingMinder, an AI assistant that transforms meeting transcripts
into
structured, actionable summaries. When someone uploads a meeting transcript,
analyze it
and generate a professional summary containing the following: an executive
summary (2–3
sentences capturing the main purpose and outcomes), key decisions (a
bulleted list of
concrete decisions made), action items (specific tasks with owners if
mentioned in the
transcript), discussion topics (main themes covered), and follow-up
questions (anything
that needs clarification or wasn't fully resolved). Present this
information in a
clean, well-organized format that's easy to scan. Be concise but
thorough—focus on
extracting what matters for people who need to know what happened and what
comes next.
```

CodeCoach: Your Pair Programming Partner

For the developers in the room. CodeCoach reviews your code, explains patterns, suggests improvements, and helps you debug when you're stuck at 2:00 a.m. wondering why nothing works.

Why it works for this book: You'll build retrieval over documentation and coding standards, add memory for project context, and create agents that can analyze code, suggest refactors, and explain error messages.

Starter prompt:

```
You are CodeCoach, a supportive pair programming partner. Your job
is to help
developers write better code by reviewing their work, explaining
concepts,
suggesting improvements, and helping debug issues. Be constructive, not
critical. Explain the "why" behind your suggestions. When
reviewing code,
consider readability, maintainability, and performance. If you see
multiple
ways to solve a problem, present the trade-offs so the developer
can decide.
```

MealMentor: Your Kitchen Companion

Love cooking? Hate figuring out what to make for dinner? MealMentor helps with recipes, meal planning, ingredient substitutions, and making the most of what's already in your fridge.

Why it works for this book: You'll add retrieval over recipe databases, build memory for dietary preferences and past meals, and create agents that can plan weekly menus and generate shopping lists.

Starter prompt:

```
You are MealMentor, a friendly kitchen companion. Your job is to help
users
plan meals, find recipes, suggest ingredient substitutions, and make
cooking
less stressful. Consider dietary restrictions, skill level, and
available
time when making suggestions. Be practical—not everyone has specialty
ingredients on hand. When suggesting recipes, focus on clear
instructions
and helpful tips. Make cooking feel approachable, not intimidating.
```

LitReview Lab: Your Research Literature Assistant

This one's a great fit for academics, grad students, or anyone who needs to make sense of a pile of papers. LitReview Lab helps you explore research topics, summarize findings, and spot patterns across sources.

Why it works for this book: You'll build retrieval over academic papers and research databases, add memory for ongoing research threads, and create multi-agent teams where one agent finds sources while another synthesizes and validates findings.

Starter prompt:

```
You are LitReview Lab, a research literature assistant. Your job is to
help
users explore academic topics, summarize key findings from papers, and
identify connections between sources. Be thorough but accessible—not
everyone
reads academic writing daily. When summarizing research, highlight
the main
contributions, methods, and limitations. Point out where sources agree,
disagree, or leave gaps worth exploring. Always note when findings might
be
outdated or contested.
```

CareerNavigator: Your Professional Growth Guide

This one's for anyone thinking about their next career move, prepping for interviews, or trying to level up professionally. CareerNavigator helps with résumés, interview prep, skill development, and navigating the job market.

Why it works for this book: You'll build retrieval over job postings and industry resources, add memory for career goals and application history, and create agents that can tailor résumés and run mock interviews.

Starter prompt:

```
You are CareerNavigator, a professional growth guide. Your job is
to help
users advance their careers by assisting with résumés, cover letters,
interview preparation, and professional development planning. Be
encouraging
but realistic. Give specific, actionable advice rather than
generic tips.
When helping with résumés, focus on impact and clarity. When
prepping for
interviews, help users articulate their experience compellingly.
```

ComplianceGuide: Your Regulatory Intelligence Assistant

If you work in a regulated industry—healthcare, finance, data privacy, or another highly regulated field—you know the pain of keeping up with requirements. ComplianceGuide helps you understand policies, extract obligations, and figure out what you actually need to do.

Why it works for this book: You'll build retrieval over policy documents and regulatory texts, add memory for your organization's compliance status, and create agents that can assess risk and recommend controls.

Starter prompt:

```
You are ComplianceGuide, a regulatory intelligence assistant. Your job
is to
help users understand policies, regulations, and compliance requirements
in
their domain. Break down dense legal language into clear obligations and
action items. When analyzing requirements, identify what's
mandatory versus
recommended, flag potential risks, and suggest practical ways to address
gaps. Be precise with citations—compliance details matter.
```

TravelGuide: Your Trip Planning Assistant

For the wanderers. TravelGuide helps you research destinations, plan itineraries, find hidden gems, and organize all the logistics that make trips actually happen.

Why it works for this book: You'll add retrieval over travel guides and reviews, build memory for travel preferences and past trips, and create multi-agent systems where one agent researches destinations while another handles logistics.

Starter prompt:

```
You are TravelGuide, a knowledgeable trip planning assistant. Your
job is to
help users discover destinations, plan itineraries, and handle travel
logistics. Balance popular attractions with local favorites.
Consider budget,
travel style, and time constraints when making recommendations. Be
specific
with practical details like timing, transportation, and booking tips.
Help
make the planning process as enjoyable as the trip itself.
```

CodeStudio: Your API Integration Assistant

Different from CodeCoach—this one's specifically about working with APIs. CodeStudio helps you understand documentation, generate working integration code, and build tests to make sure everything works as expected.

Why it works for this book: You'll build retrieval over API documentation and code examples, add memory for your project's existing integrations, and create agents that can scaffold code and propose test suites.

Starter prompt:

```
You are CodeStudio, an API integration assistant. Your job is to help
developers understand APIs and generate working code that uses them
correctly.
When given documentation, extract the key endpoints, parameters, and
patterns.
Produce clean, well-commented code scaffolds that developers can build
on.
Suggest basic tests to verify that the integration works. Focus on
practical,
runnable examples rather than theoretical explanations.
```

FinanceHelper: Your Money Management Partner

Personal finance can be overwhelming. FinanceHelper breaks down concepts, helps with budgeting, and provides guidance on financial decisions—without judgment.

Why it works for this book: You'll build retrieval over financial resources and tax documents, add memory for financial goals and transaction patterns, and create agents that analyze spending and suggest optimizations.

Starter prompt:

```
You are FinanceHelper, a supportive money management partner. Your
job is to
help users understand financial concepts, create budgets, and make
informed
decisions about their money. Be clear and jargon-free. Never judge
spending
habits—focus on helping users achieve their own goals. When explaining
financial concepts, use concrete examples. Be upfront that you're
providing
general guidance, not professional financial advice.
```

ExperimentDesigner: Your ML Research Planning Assistant

For the ML practitioners and researchers. ExperimentDesigner helps you turn vague research questions into concrete, testable experiments with clear metrics and reproducible plans.

Why it works for this book: You'll build retrieval over ML papers and methodology guides, add memory for your ongoing research projects, and create agents that can propose experiments, review feasibility, and generate reproducibility checklists.

Starter prompt:

```
You are ExperimentDesigner, an ML research planning assistant. Your
job is to
help researchers turn ideas into testable experiments. When given a
research
question, propose clear hypotheses, appropriate metrics, and realistic
baselines. Consider what could go wrong—confounding variables, data
issues,
and computational constraints. Aim for reproducible plans that another
researcher
could follow. Be honest about uncertainty and trade-offs.
```

GraphBuilder: Your Knowledge Extraction Assistant

This one's for anyone working with unstructured text who needs to extract structure. GraphBuilder helps you identify entities, relationships, and patterns that you can query later.

Why it works for this book: You'll build retrieval over document collections, add memory for evolving knowledge schemas, and create agents that extract entities, normalize relationships, and validate consistency.

Starter prompt:

```
You are GraphBuilder, a knowledge extraction assistant. Your job is to
help
users identify structure in unstructured text—entities, relationships,
and
how concepts connect. When analyzing documents, extract the key players
and
their relationships clearly. Suggest how information could be organized
for
querying later. Be explicit about what's stated directly versus what
you're
inferring. Note where the source is ambiguous or incomplete.
```

IdeaBridge: Your Cross-Domain Innovation Assistant

This is the wild card. IdeaBridge helps you find unexpected connections between different fields—borrowing ideas from biology to improve your algorithms or applying engineering principles to organizational design.

Why it works for this book: You'll build retrieval across multiple domain-specific document collections, add memory for promising cross-domain threads, and create agents that can identify parallels, synthesize ideas, and assess feasibility.

Starter prompt:

```
You are IdeaBridge, a cross-domain innovation assistant. Your job is to
help
users find unexpected connections between different fields—such as AI
and
biology or engineering and art. When exploring a concept, look for
parallel
ideas in other domains that might offer fresh perspectives or
techniques.
Propose concrete ways insights from one field could apply to another. Be
creative but grounded—wild ideas are great, but they should be
buildable.
```

Pick Your Path

Any of these projects will work well as you progress through the book. The patterns you'll learn apply across all of them:

- **Chapters 1–2:** Get your basic assistant working with RAG for document retrieval.
- **Chapters 3–4:** Transform it into a proper agent with tools and reasoning.
- **Chapters 5–8:** Add memory, multi-agent coordination, and deep research capabilities.
- **Chapters 9–11:** Build serious evaluation pipelines to measure and improve performance.
- **Chapters 12–14:** Wire everything together into a full-stack application.
- **Chapters 15–18:** Deploy to production with proper guardrails and optimization.

Pick the project that gets you excited to sit down and code. That's the goal. And if none of these appeal to you, invent your own. Maybe you want a DungeonMaster for running tabletop games, a GardenAdvisor for your plants, or a FitnessCoach for your workouts. The framework is the same—start simple, add retrieval, make it agentic, ship it, and share it.

Index

Numerics

12-Factor Agent methodology, 51–53
  configuration and dependencies in, 52
  disposability in, 52
  logs in, 52–53
  state management in, 52
12-Factor App methodology, 51

A

accessibility, AI application, 331–332
Accessible Rich Internet Applications (ARIA) attributes, 331–332
adaptive retriever, 313–315, 320–321
advanced retrieval pattern, 178–179
adversarial collaboration, 138
adversarial test cases, 264
  ambiguous questions, 264
  misleading questions, 264
  out-of-scope questions, 264
  unanswerable questions, 264
agent context vs RAG context, 78–79
agent decisions, rolling back, 235–236
agent engineering, 69–70
agent harness, 69–70, 97
agentic RAG, 95–128
  ambiguity and uncertainty in, 110–111
  architecture of, 103–106
  definition of, 96–97
  dynamic retrieval strategies in, 107–108
  need for, 97
  query planning and analysis in, 106–107
  result synthesis in, 109–110
agentic workflows, 136
agent loop, 7, 61, 64–67
agent memory, 8
  architecture of, 173–179
  consolidation of, 186–187
  formation strategies for, 181–184
  integration patterns for, 187–189
  need for, 171–172
  persistent, 172
  vs RAG, 173
  storage and retrieval of, 184–187
  types of, 179–181
agent personalities, 138
agent reasoning, exposing, 90–92
agents
  actions of, 66
  challenges of, 8
  core functions of, 64–67
  definition of, 62
  need for, 64

observation in, 66
perception in, 64–65
planning in, 65–66
role of, 7–8
termination of, 67
vs workflows, 62–63
AI behavior, transparency about, 331
AI Engineer Challenge (repository), 15, 18–19
AI engineering
definition of, 1–2
vs machine learning engineering, 1–2
reference guides, 114, 116
vs software engineering, 2
AI engineering stack, 2–3
API and inference layer, 2
application layer, 3
foundation models, 2
orchestration layer, 3
AI engineers
key skills and tools for, 8–9
role of, 2
skills for prototyping and production, 6–7
ambiguous queries, detecting, 110
analysis node, 104, 118–119
ANN. *See* approximate nearest neighbors (ANN)
answer relevance (RAGAS metric), 259–260
Anthropic, 133
Anysphere, 356
`/api/flashcard` endpoint, 124
API keys
Anthropic, creating, 370–371
Hugging Face, 371
OpenAI, creating, 370
storing securely, 371–372
approximate nearest neighbors (ANN), 39–40, 43–44
ARIA attributes. *See* Accessible Rich Internet Applications (ARIA) attributes
async programming, LLM API calls and, 8
authentication, user, 326, 335–336, 350–351

B

backend development, LLM applications and, 8
background job processing, 326, 328
background memory extraction, 182–184, 193
`BackgroundTasks` parameter, 328
backtracking, 234–236, 246
agent decisions, rolling back, 235–236
alternative paths, backtracking and, 236
dead ends, detecting, 234–235
backward chaining, 216–217, 238
behavior adaptation, 189
Best Matching 25 (BM25), 291–294, 305
`BM25Retriever` class, 292
bi-encoders, 294
BM25. *See* Best Matching 25 (BM25)
brute force search, 39–40

C

cache hit rates, 197, 207
caching, 193–198
cache keys, hashing for, 196
content-addressed, 193–195
cost of redundant generation, 193
database-backed, 194–195
generated content, 195–198
performance improvements, measuring, 197
when to use, 197–198
Card Generator agent, 151, 153–154, 237, 240
CareerNavigator, 382
chain-of-thought prompting, 12
chain-of-thought reasoning, 4, 227
chat interfaces, frontend, 329, 345
Chroma, 38

chunking, 44–49
  best practices for, 48
  embedding-based boundary detection in, 300
  fixed-size, 45, 55
  importance of, 44
  metadata and, 48–49
  overlap in, 47–48
  by section, 47
  semantic, 45–47, 291, 300–301, 305–307, 322, 339, 341–342
  structure-aware, 300–301
CI/CD integration, evaluation and, 270
circuit breaking, 334
clarifying questions, 110
CLI. *See* command-line interface (CLI)
CodeCoach, 380
code generation workflows, 147–148
CodeStudio, 383
Cohere embedding models, 37
Cohere reranking API, 295–296, 309–311
cold-start problem, test data and, 257
command-line interface (CLI), 355, 358
complexity assessment, 107
ComplianceGuide, 382
confidence scoring, 104–105, 109–110
conflicting information, handling, 109
conflict resolution, 145
content-addressed caching, 193–195, 203–207
content creation workflows, 148
content generation tools, 324
context compression, deep agents and, 219–220
context engineering, 12–13, 32–34, 78–80
context isolation, 132
context overload, 79–80
context precision (RAGAS metric), 259–261, 304, 316, 322
context problem, 31
context recall (RAGAS metric), 259–260, 304, 316, 322
context rot, 131–132
contextual compression, 301, 303
context windows, 4–5, 12, 32–33, 78–80, 132, 172, 217
continuous deployment, 18
contrastive learning, 37
coordinator agent, deep agent, 221
cosine similarity, 41–42, 44, 55
coverage verification, test set, 265
`create_agent` API, 72–73, 84, 87–89, 95, 97–100
credential storage, Git, 361
cross-encoders, 294, 296
cross-register vocabulary mismatch, 290
curriculum generation, AI, 338
`CurriculumModal.tsx` component, 249
Curriculum Planner agent, 237–246, 249
Cursor, 16–18, 20, 356–357, 369
custom builds vs frameworks, 73
customer support
  automation, 324
  chatbots, 130
custom evaluators, educational content, 275–278

D

database design, agent applications, 335–336
  conversation history in, 336
  document storage in, 335
  generated content in, 336
  users and sessions in, 335
  workspaces in, 335
dead ends, detecting, 234–235
deduplication
  of retrieved documents, 104, 108, 120
  of test cases, 265
  of uploaded documents, 340

deep agents
backtracking and recovery in, 234–236
context management in, 217–220
definition of, 211–213
four key elements of, 213–214
multistep reasoning in, 227–230
need for, 213
planning and task decomposition in, 214–217
self-reflection in, 230–234
vs shallow agents, 212–213
subagent spawning in, 221–227
delegation
error handling in, 226
hybrid patterns of, 225
parallel vs sequential, 224–225
results, collecting and integrating, 227
dense vector retrieval, 289–290
`detect_stuck` function, 146–147
development environment, setup of, 355–377
API keys, 370–372
code editor, 356–357
Docker, 372–375
Git and GitHub, 359–367
integration test, 377
Jupyter notebooks, 375–376
project structure, 376–377
Python with `uv`, 367–369
terminal, 357–358
dev-prod parity, 189
direct answer node, 105, 122
distance metrics, 41–44
Docker, 372–375
basics of, 374–375
installing, 373–374
docstrings, for tools, 75, 87
document loaders, LangChain, 84, 86
document upload and indexing, 339–343
dot product, 43–44
dynamic retrieval strategies, 107–108

E

Ed25519 algorithm, for SSH keys, 362
edges (LangGraph), 98, 101
conditional, 101–102, 105
editor extensions, essential, 357
embeddings, 9, 35–37, 49
definition of, 35–36
dimensionality of, 37
meaning captured by, 37
models, 36–37, 49, 51
emergent capabilities, 4
`.env` file, 371
environment variables, 21, 28, 52, 371–372
episodic memory, 180–181
error handling
agent integration and, 334–335
multi-agent systems and, 145–146
error messages, AI application, 332
Euclidean distance, 43–44
evaluation. *See also* RAGAS (RAG Assessment) framework
automated metrics for, 15
dashboard, 282–285, 287–288
importance of, 9, 14, 255–256
infrastructure for, 15
LangSmith for, 268–270
monitoring and, 9
node, 104, 120–121
nondeterminism and, 256
systematic, 15
test data for, 256–258
test datasets for, 15
thresholds for, 267
tracker, 266
evaluation datasets
adversarial set, 269
comprehensive set, 269
golden set, 268
organizing, 268–269
regression set, 268
StudyBuddy, building for, 272–275
exact match failure, dense retrieval, 290

ExperimentDesigner, 384
exponential backoff, 334–335

F
FAISS, 38
faithfulness (RAGAS metric), 256, 258–261, 267
FastAPI, 8, 17, 20, 22–24, 165–166, 327–329
feedback loops, 10, 13, 67, 247, 249, 324
few-shot prompting, 11
file size limits, document upload, 339
FinanceHelper, 384
fine-tuning, 113
flashcards, 115, 124
  minimum information principle for, 343
  program scoping of, 343–344
  quality evaluator, 276
focus areas, StudyBuddy sidebar, 249
`FocusAreas.tsx` component, 249
"Forget RAG, the Future is RAG Fusion" (Raudaschl), 298
forward planning, 216
foundation models, 2
frameworks vs custom builds, 73
full-stack applications
  agent integration in, 333–335
  architecture of, 325–327
  backend architecture of, 327–329
  database design for, 335–336
  frontend components of, 329–330
  industry use cases for, 323–325
  state management in, 326, 332–333
  user experience in, 331–332
function calling, 73–74, 76–77

G
generation node, 104–105, 121–122
`get` method, 175–176
Git, 16, 359–367
  basic workflow of, 366
  identity, configuring, 359–360
  installing, 359
GitHub, 16, 18–19, 359–366
  authentication for, 360–366
GitHub Copilot, 357
`.gitignore` file, 367, 371
goals, evaluating progress toward, 231
Goodhart's Law, 267–268
GPT-3, 3
GraphBuilder, 385
guardrails, agent, 69

H
hallucinations, 4, 9, 258–259
handoffs, 139–140
  structured, 140
  tool-based, 139
health checks, backend, 329
Hierarchical Navigable Small World (HNSW), 40, 43, 80
hierarchical pattern (multi-agent), 135–136
hierarchical retrieval, 302–303
high-dimensional similarity, curse of, 290
HNSW. *See* Hierarchical Navigable Small World (HNSW)
Homebrew, 190, 199
hot path memory formation, 181–182, 184
HTTPS, switching to SSH keys, 366
Hugging Face
  access token for, 371
  cross-encoder models from, 296
human-in-the-loop approach, 338
hybrid memory strategies, 184
hybrid search, 290–294, 305
  in practice, 293–294
  Reciprocal Rank Fusion (RRF), 292–293

I
IdeaBridge, 385
`identify_weak_areas()` function, 283

improvement loop, metrics-driven, 266
in-context learning, 3–4
indexing strategies, vector database, 39–41
infinite loops, preventing, 104–105, 135
injection at start, 187
integration test, first, 377
inverse document frequency, 291
Inverted File Index (IVF), 40, 43
iterative development, LLM applications and, 13
IVF. *See* Inverted File Index (IVF)

J

JupyterLab, 376
Jupyter notebooks, 5, 6, 13, 375–376

K

KaTeX, math rendering with, 345
keyword matching vs semantic similarity, 36
k-nearest neighbors (kNN), 43
knowledge graph approach, RAGAS testset, 261, 270
knowledge management, RAG-powered, 324

L

LangChain, 3, 9, 61, 70–73
  component architecture of, 70–71
  composability of, 70
  framework lock-in and, 70–72
  Qdrant integration with, 82–83
  tools, defining in, 74–75
LangGraph, 3, 9, 95, 97, 99–103, 105, 114–115
  core concepts of, 100–101
  vs `create_agent`, 99–100
  first graph, building, 101–103
  parallel execution in, 148
  shared state with, 140
  visualization in, 103
LangSmith, 9, 15, 96, 111–112, 126
  baseline evaluation with, 278–280
  datasets in, 268–269
  evaluations, running in, 269
  experiments, comparing in, 269–270
  feedback collection in, 112
  results, viewing in, 288
  traces in, 111–112, 126
LangSmith Studio, 148–150, 167–168
  chat mode in, 149
  graph mode in, 149
  time-travel debugging in, 150
  v2 of, 112, 149
Language Models Are Few-Shot Learners (OpenAI paper), 3–4
large language models (LLMs)
  application stack, 34–35
  capabilities of, 3–4
  limitations of, 4–5
  probabilistic nature of, 2
latency, multi-agent systems and, 130–131, 134
learning and training applications, 324
Learning Coordinator (supervisor), 151, 156–159
learning goals, task decomposition for, 238–240
learning history tracking, 198
learning insights, tutoring and, 248–249
learning programs, StudyBuddy, 337–339, 345–346
LitReview Lab, 381
Llama, 112–113
`llama3.2:3b` model, 113–114
LlamaIndex, 3
LLMs. *See* large language models (LLMs)
loading states, AI application, 331
`localStorage`, client-side state and, 332
long-running operations, progress updates for, 334

M
maintainability, multi-agent systems and, 130
manager agents, 137–138
`MarkdownHeaderTextSplitter` class, 300
Massive Text Embedding Benchmark (MTEB), 37
maximum iteration limits, 67, 69
MealMentor, 381
MeetingMinder, 379–380
memory consolidation, 186–187
`Memory` model, 174, 192
`MemoryStore` class, 175–176, 178, 192
memory store pattern, 175–176
memory tools, 176–177, 182
metacognition, agent, 230
metadata
  filtering, 80, 83
  memory storage and, 184–185
metrics-driven development, 266–268
minimum information principle, 343
Mistral, 112
mobile-responsive design, 330
MTEB. *See* Massive Text Embedding Benchmark (MTEB)
multi-agent systems
  agent roles in, 137–139
  architectures of, 8, 133–137
  case for, 130
  context optimization in, 131–133
  coordination in, 139–147
  debugging, 148–150
  decision framework for, 131
  error handling in, 145–146
  need for, 129–131
  testing, 166–167
  use cases of, 147–148
  when to avoid, 130–131
multi-source retrieval, 108
multistep reasoning, 227–230
  reasonable stopping conditions, 228–230
  sequential reasoning chains, 227–228
multi-tenancy, schema design for, 335

N
namespaces, 174, 176–178, 203
needle-in-the-haystack problem, 32
Neon, 193
Next.js, 16–17, 24–25, 345
node functions, 140–143
Node.js, 17, 25
nodes (LangGraph), 98, 100–101, 104–105
`nomic-embed-text` model, 113–114
nondeterminism, AI system evaluation and, 256
`npm`, 17, 25, 27

O
observability, 9, 96, 111–112
offline scenarios, handling, 333
Ollama, 113–114
on-demand retrieval, 187–188
OpenAI API
  account for, 16
  embeddings, 53, 56
  key, setting, 21
open-source models, 112–114
  advantages of, 112–113
  embedding models, 37
  trade-offs of, 113
optimistic updates, 333
orchestration frameworks, 3, 9

P
parallel execution, multi-agent, 130, 148
parallel research patterns, 148
PATs. *See* personal access tokens (PATs)
persistence, testing, 209–210
personal access tokens (PATs), 360–361
pgvector, 39
Pinecone, 38
pipeline pattern, 136–137
Plan-and-Execute pattern, 63
playgrounds, for prompts, 6, 13
polling, flashcard generation, 347
PostgreSQL, 38–39, 189–193, 199, 209

connecting to, 192
local setup of, 190
schema design for, 190–192
POSTGRES_URL environment variable, 192–193, 199
proactive memory, 188–189
procedural memory, 180–181, 189
productionizing phase, LLM applications and, 6
progress dashboard, learning, 346
progressive disclosure, 331
project structure, AI engineering, 376–377
prompt engineering, 4, 11–12
prototyping
best practices for, 11–13
phase of, 5–6
psql, 190–191, 209
put method, 175
Pydantic models, 140, 215
pyproject.toml file, 17, 20
Python, 8, 367–369
python-dotenv library, 20

Q

Qdrant, 38–39, 80–86, 94
architecture of, 80
collections in, 80, 83–84
in-memory mode of, 81, 84–86
LangChain integration with, 82–83
program collections in, 337, 339, 341, 350
server mode of, 81–82
Qdrant Cloud, 81, 94, 353–354
Quality Checker agent, 151, 154–156
quality thresholds, evaluation, 267
query analysis, 104, 106–107
query decomposition, 106–107
query-document mismatch, 96
query drift, 297–298
query expansion, 291, 296–298, 305
LLM-based, 297
risks and alternatives of, 297–298
query reformulation, 97, 108
queue management, background jobs and, 329
Qwen, 112

R

RAG. *See* retrieval-augmented generation (RAG)
RAGAS (RAG Assessment) framework, 258–261, 270–272, 304, 315–316, 322
basic evaluation with, 260
four core metrics of, 258–259
results, interpreting, 260–261
testset generation with, 261
Raudaschl, Adrian H., 298
React, 17, 24
React hooks, 330, 332, 345
react-markdown library, 345
ReAct pattern, 7, 63, 67–68
implementation of, 67–68
traces in, 63, 68
reasoning models, 68
recency weighting, 185–186
Reciprocal Rank Fusion (RRF), 292–293, 298–299, 308
weighted, 292–293, 308
reflection loop, 232–234
Reflexion, 63
relevance filtering, retrieved documents, 301–303
remember tool, 177
report generation teams, 147
repositories
cloning, 18–19
forking, 18
requirements.txt file, 20
reranking, 290–291, 294–296, 305
two-stage architecture of, 294–296
when to use, 296
result synthesis, 109–110
retrieval
dense, blind spots of, 290
upgrading, decision framework for, 290–291, 304–305

retrieval-augmented generation (RAG), 2, 9, 31–60. *See also* agentic RAG; embeddings
  agentic, 96–97
  indexing phase of, 33, 48–49
  LLM application stack for, 34–35
  vs memory, 173
  pipeline for, 48–51, 54–57
  pitfalls of, 51
  query phase of, 33, 49–51
  RAG-Fusion, 291, 298–299, 305, 311–313
  as solution pattern, 33–34
  static, limitations of, 96
retrieval node, 104, 119–120
retrieval precision evaluator, 277
retrievers
  adaptive, 313–315, 320–321
  comparison tool for, 317–321
  controlling variables in comparing, 303–304
  LangChain, 83–84
  segment analysis of, 304
  systematic comparison of, 303–305
retrieve-then-generate approach, 96, 103
retry logic
  agent integration and, 334–335
  tool failures and, 69, 78
role confusion, 138–139
routing functions, 101–102, 105, 122–123, 143, 162–163
RRF. *See* Reciprocal Rank Fusion (RRF)

S

Scheduler agent, 170, 198
`search_by_content` method, 185
`search_materials` tool, 86–87, 89
`search` method, 175–176, 178
Secure Shell (SSH) keys, 360–366
selective information retention, deep agents and, 218–219
self-reflection, agent, 230–234
`SemanticChunker` class, 300
semantic chunking, 45–47, 291, 300–301, 305–307, 322, 339, 341–342
semantic memory, 179–181
semantic search, 9, 34, 36, 185
semantic similarity vs keyword matching, 36
sequential reasoning chains, 227–228
Server-Sent Events (SSE), 327–328, 330, 348–350
service layer pattern, 334
shallow agents vs deep agents, 212–213
shared state, 140, 159–160
Sherlock Holmes stories (document set), 84, 87, 91
similarity search, 41–44
`SimpleVectorDB` class, 42, 54–55
SM-2 algorithm, 170, 198, 237
source attribution, 104, 109
spaced repetition, 170, 198, 209–210, 237, 252, 343
sparse search, BM25 and, 291
speed-accuracy trade-off, 38, 40–41, 44
SQLAlchemy, 173, 175, 178, 189, 192, 200
SQLite, 173, 189, 192
SSE. *See* Server-Sent Events (SSE)
SSH. *See* Secure Shell (SSH) keys
state (LangGraph), 98, 100, 103–104
`StateGraph` class, 101–102, 105, 144–145
statelessness, of LLMs, 171–172
state machines, 97–99
state management
  client-side, 332
  full-stack layers and, 326, 332–333
  server-side, 332
  synchronization of, 333
strategy adjustment, agent, 231–232
`StreamingResponse` class, 327–328
streaming responses, 329–330, 348–350
struggle areas, StudyBuddy tracking of, 247–249

stuck loops, detecting and breaking, 146–147
StudyBuddy app
adaptive retriever in, 313–315
agent team for, 151–156
backend setup for, 20–24
backtracking in, 246–247
baseline evaluation of, 278–280
Cohere reranking in, 309–311
context across sessions in, 243–246
curriculum planning in, 237–240
custom evaluators in, 275–278
deploying to Vercel, 27–29
document upload and indexing in, 339–343
evaluation API endpoint for, 280–282
evaluation dashboard for, 282–285
evaluations, running for, 315–317
flashcard caching in, 203–207
flashcards in, 115, 124, 343–345
frontend of, 24–27, 249
full-stack interface of, 345–348
hybrid search in, 307–309
knowledge base of, 114, 116–117
learning programs in, 337–339
memory-connected learning modes in, 247–249
persistence in, 199, 209–210
progress checkpoints in, 241–242
RAGAS setup for, 270–272
RAG-Fusion in, 311–313
RAG status polling in, 58–59
retrieval comparison tool in, 317–320
semantic chunking in, 306–307
source attribution in, 53
streaming responses in, 348–350
supervisor for, 156–159
testing, 166
user-scoped data isolation in, 350–351
v1, building, 15–29
v2, building, 53–60
v3, building, 84–94
v4, building, 114–128
v5, building, 150–170
v6, building, 198–210
v7, building, 237–254
v8, building, 270–288
v9, building, 305–322
v10, building, 336–354
subagent lifecycles, 222–224
`SubagentManager` class, 222–223
subagent spawning, 221–227
summarization, of context, 219–220, 243
supervisor arbitration, 145
supervisor pattern, 133–134, 150
supervisor prompting, 137–138
swarm pattern, 134–135
synonym expansion, 298
synthetic data, 257–258, 270
advantages of, 257
limitations of, 257
quality validation of, 257–258
system prompts, 11, 23, 87–88

T

task decomposition, 130, 213–216, 238
temperature, agent settings for, 153–154, 156
Tenacity (Python library), 334–335
term frequency, BM25 scoring and, 291
terminal, basics of, 357–358
test cases
adversarial, 264
difficulty-based generation of, 262
generating, 262–264
question types in, 263
test data
challenge of creating, 256–257
quality and coverage of, 264–265
synthetic, 257–258
`TestsetGenerator` class, 261
`text-embedding-3-large` model, 36
`text-embedding-3-small` model, 36–37, 56
text splitters, LangChain, 84, 86
theory of mind, 4

three-layer architecture, full-stack, 325–327
  agent layer, 326
  backend layer, 326
  frontend layer, 325
timeout handling, LLM requests and, 329
time-travel debugging, 112, 126, 150
token economics, 133
tokens, 5, 32
tools
  defining, in LangChain, 74–75
  descriptions of, 75–76
  error handling in, 77–78
  parameter extraction for, 76–77
  selection of, 76
  working of, 74
`@traceable` decorator, 111
TravelGuide, 383
Tree of Thoughts, 63
Tutor agent, 151–153
tutoring quality evaluator, 275
`TypedDict`, 100

U

uncertainty, acknowledging, 110–111
unit tests, for evaluation system, 285–286
user intent, understanding, 106
user-scoped data isolation, 350–351
`uv` (package manager), 17, 20–21, 23, 367–369, 375
`uvicorn`, 23–24

V

vector databases, 9, 37–39
  architecture of, 38
  custom, building, 39
  need for, 39
  popular solutions, 38–39
Vercel, 16, 18, 27–29, 59–60, 93–94, 126–127, 169, 208–209, 253–254, 352–354
`vercel.json` file, 28–29
Vercel Postgres, 192–193, 199, 209
vibe checks, 10–11, 14, 256
vibe coding, 16, 18, 24
virtual environments, Python, 17, 21, 368–369
vocabulary mismatch, 290–291, 296, 305
VS Code, 356–357, 369

W

Weaviate, 38
WebSockets, 326–328, 348
Windows Subsystem for Linux (WSL), 358
Windows Terminal, 358
worker agents, 138
workflows vs agents, 62–63
WSL. *See* Windows Subsystem for Linux (WSL)

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.