Github Copilot Step by Step:

Navigating AI-driven software development

Dr. Gomathi S.

Microsoft

Github Copilot Step by Step

Published with the authorization of Microsoft Corporation by:
Pearson Education, Inc.

ISBN-13: 978-0-13-549304-5
ISBN-10: 0-13-549304-8

Library of Congress Control Number: On File

$PrintCode

Trademarks

Cover: BonkersArt/stock.adobe.com
Figures 1.1, 1.4, 2.1, 2.2, 8.1–8.12, 9.1–9.14, 10.1: © 2025 GitHub, Inc
Figure 1.3: © 2000-2025 JetBrains s.r.o.

Editor-in-Chief

Julie Phifer

Portfolio Manager

Loretta Yates

Acquisitions Editor

Shourav Bose

Development Editor

Songlin Qiu

Managing Editor

Sandra Schroeder

Production Editor

Mary Roth

Copy Editor

Kitty Wilson

Technical Editor

Charles Pluta

Indexer
Timothy Wright

Proofreader
Barbara Mack

Compositor
codeMantra

Contents

Foreword

I still remember the day—December 5, 2022—when I used GitHub Copilot for the first time, and it wrote a complete function before I finished writing the comment. It wasn't just faster; it felt different. I felt like I wasn't coding alone anymore, and that moment changed how I thought about writing software. Not because AI replaced my thinking, but because it helped me to think bigger, build faster, and learn continuously while shipping real code.

In the last few years, I've had the privilege of being at the forefront of the Copilot revolution, advising my teams at Microsoft, integrating AI-powered workflows, and sharing best practices with the global developer community. Fast forward to now: we have a matured, powerful AI coding assistant that has evolved far beyond a tool for autocomplete suggestions. GitHub Copilot transformed into a sophisticated pair programmer that understands context, engages in natural language conversations, generates unit tests, refactors legacy code, and even integrates into our development workflows. It is not just a productivity hack anymore, but a foundational tool in reshaping how millions of developers today write and learn from GitHub Copilot. I highly recommend reading my recent *Business Insider*[*] article about how as a Principal Software Engineering Manager at Microsoft, AI helped me in my day-to-day activities.

* https://www.businessinsider.com/microsoft-manager-ai-reduces-busywork-not-daily-workload-2024-10

Dr. Gomathi, a Microsoft MVP, Microsoft certified trainer (MCT), and passionate AI advocate, has poured her deep technical expertise and teaching experience into every chapter, from setting up Copilot across multiple IDEs to crafting effective prompts, automating tests, avoiding common pitfalls, and preparing for the future of AI-powered development.

This isn't just another tech book. It's comprehensive with hands-on practical knowledge to help those who want to master GitHub Copilot. I'm grateful she wrote this book. And I'm grateful you're reading it.

We're at the beginning of something transformative. AI-powered development is no longer experimental; it's becoming standard practice in every company. Developers who learn to work *with* AI, who understand how to prompt effectively, iterate intelligently, and maintain human oversight, will have a massive advantage in the years ahead. This book is your foundation for that future. Whether you're a student learning to code, a professional looking to boost productivity, or a team leader exploring AI-assisted workflows, the lessons here will serve you well.

There is a truth I've come to embrace in this new AI and agentic landscape:

"AI will not replace your job, but a person who is leveraging AI can."

Start leveraging GitHub Copilot and go write some code. Copilot is waiting—and so is the future.

Happy coding!

Naga Santhosh Reddy Vootukuri
Principal Software Engineering Manager
Microsoft, Azure SQL Server (Cloud+ AI division)

Acknowledgments

This book is the result of not just my work but the collective strength, love, and encouragement of many people who have been part of my journey.

I dedicate this book to my beloved mother, whose love and guidance continue to inspire me every day. With deep gratitude, I honor my father, V. Srinivasan, and my husband, Anantha Krishnan A, for their unwavering support and encouragement through every challenge. To my twin gems, Vishanth and Vishwanth, your boundless joy lights up my world.

I extend heartfelt thanks to my in-laws, Lakshmi and Appadurai, for their constant encouragement, and to Mahalakshmi and Padmanaban, whose warmth resembles that of my parents. I am also grateful to my brother, Viswanathan, and my sisters-in-law, Gomathy and Anantha Kalyani, for their love and support.

Special thanks to the publishing and editorial team who have kept me on track and made this book better at every stage. Thank you to Shourav Bose for guiding me through the entire process, Songlin Qiu for shaping the structure, and Dan Foster for refining the narrative with clarity and precision. My sincere thanks also go to Mary Roth, Content Producer, and Jayaprakash P., Senior Project Manager, for their invaluable assistance and coordination during the production phase, ensuring a smooth and successful completion of this book. I am especially grateful to my technical editor, Charles Pluta, for ensuring accuracy, providing valuable tips, and offering positive feedback throughout this journey.

I am equally indebted to my students, mentors, friends, and the global Microsoft community, who constantly inspire me to share knowledge and keep learning.

Finally, to the readers of this book: Your curiosity and commitment to learning are the true motivation behind my work. May this book help you embrace AI-powered development and achieve your aspirations.

About the author

Dr. Gomathi S. is a Microsoft Most Valuable Professional (MVP), a Microsoft Certified Trainer (MCT) Community Lead, and Microsoft Learn Expert, with more than 14 years of experience in teaching, mentoring, and technology leadership. She has trained thousands of professionals and students globally in Power BI, Business Central, Power Apps, Microsoft Fabric, data science, and AI-driven tools.

As the author of two technical books on Microsoft Dynamics 365 Business Central, she has established herself as a trusted voice in ERP and AI-powered development. Through her YouTube channel *Gom Tech Talks*, global workshops, and community sessions, she continues to inspire professionals, students, and educators alike to embrace technology with confidence.

An accomplished researcher and innovator, Dr. Gomathi holds six Australian and five national patents, has published more than 30 research papers in Scopus and international journals, and serves as a reviewer for leading indexed journals. She is also a record holder in the Asia and India

Book of Records and a recipient of the prestigious Enrique Lima Award (Asia Region, 2024), Young Scientist, and Women Scientist Awards.

Beyond her professional achievements, Dr. Gomathi is a passionate mentor, dedicated to empowering women in ERP and technology, and takes pride in balancing her career with being a mother of twin children.

Introduction

GitHub Copilot is an AI-powered coding assistant designed to help developers write better code faster. Built on OpenAI's Codex and deeply integrated with popular IDEs such as Visual Studio Code, JetBrains, and Neovim, it works alongside you like a virtual pair-programmer. Copilot suggests entire functions, generates unit tests, assists with debugging, and even helps refactor legacy code. It reduces repetitive work, accelerates learning, and makes complex coding tasks more approachable, whether you are building web apps, automating scripts, or exploring data pipelines.

Copilot is more than an autocomplete tool; it is a true productivity partner. You can describe your intent in plain English, and it translates that into working code across multiple programming languages. From writing queries in SQL to creating responsive web layouts in HTML and JavaScript, or developing enterprise logic in Python and C#, Copilot adapts to your project's context and provides intelligent, context-aware suggestions. It helps you focus on the logic and design of your solutions while handling much of the boilerplate code.

Who this book is for

GitHub Copilot Step by Step and other books in the Step by Step series are designed for beginning to intermediate developers. This book is for anyone who wants to understand and use Copilot effectively, regardless of whether you are a student just learning to code, a professional developer seeking to boost productivity, or a team member working in collaborative enterprise environments.

It is especially valuable for:

- Software developers (front-end, back-end, full-stack)
- Data engineers, analysts, and AI/ML practitioners
- Cloud developers and DevOps engineers
- Solution architects and automation engineers
- Students, freelancers, and hobbyists who want to code smarter, not harder

How the book is organized

This book is organized into ten chapters, each building your knowledge of GitHub Copilot in a structured way.

- **Chapter 1** introduces GitHub Copilot, explaining its core features, benefits, limitations, and the responsible AI principles behind it.
- **Chapter 2** walks you through setting up Copilot across popular IDEs such as Visual Studio Code, JetBrains, and GitHub Codespaces, covering account setup, installation, and configuration.
- **Chapter 3** focuses on writing effective prompts, teaching you how to guide Copilot to generate accurate, context-aware code.
- **Chapter 4** shows how to enhance code efficiency with Copilot by automating repetitive tasks, refactoring existing code, and applying best practices for cleaner design.
- **Chapter 5** explores Copilot's role in debugging and troubleshooting, helping you detect errors, suggest fixes, and optimize performance.
- **Chapter 6** demonstrates how to write and automate tests with Copilot, including unit tests and integration testing for different frameworks.
- **Chapter 7** highlights Copilot's value in collaboration by assisting with code reviews, documentation, and shared coding practices.
- **Chapter 8** presents real-world scenarios, showing how Copilot supports workflows in areas like data science, cloud development, and enterprise applications.

- **Chapter 9** identifies common pitfalls developers face when using Copilot and provides strategies to avoid them, ensuring responsible and effective use.
- Finally, **Chapter 10** looks ahead to the future of AI in software development, exploring emerging trends, ethical considerations, and how tools like Copilot will continue to evolve.

This structure supports two types of readers. If you are new to GitHub Copilot or generative AI, the early chapters will help you build a strong foundation with clear explanations, setup instructions, and guided exercises. If you are already familiar with Copilot or prompt engineering, you can skip ahead to the later chapters to focus on advanced use cases, workflow integration, collaboration, and professional practices.

The book has been designed to lead you step by step through the tasks you're most likely to perform with Copilot in your everyday development work. Because generative AI is creative by nature, you will discover new possibilities as you experiment with prompts, explore different coding contexts, and apply Copilot to your own projects. Each topic is self-contained, so you can jump in wherever you need specific skills or reinforcement.

Download the practice files

Before you can complete the exercises in this book, you need to download the book's practice files to your computer. These practice files, and other resources, can be downloaded from the following page:

MicrosoftPressStore.com/gitcopilot/downloads

The following table lists the practice files for this book.

Chapter	File

Chapter	File
Chapter 5: Debugging and Troubleshooting Code with Copilot	buggy_functions.py error_handling_cases.py identify_issues.py performance_review.py

Errata, updates, and support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed at:

MicrosoftPressStore.com/gitcopilot/errata

If you find an error that is not already listed, you can report it to us through the same page.

For additional book support and information, please visit *MicrosoftPressStore.com/Support*.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *support.microsoft.com*.

1

Introduction to GitHub Copilot

In this chapter

- Understand GitHub Copilot and its core functionality
- Explore the AI technology behind GitHub Copilot
- Identify the key benefits of GitHub Copilot for developers
- Learn how GitHub Copilot handles privacy and applies responsible AI
- Recognize the limitations and ethical considerations of GitHub Copilot

Practice files

There are no practice files for this chapter.

Generative AI is reshaping creativity, problem solving, and technology interaction. One of the critical components of innovation is large language models (LLMs), which are AI systems trained on extensive amounts of code, documentation, websites, and technical forums. These models have an understanding of code and provide intelligent suggestions based on the context and intent.

GitHub Copilot is a GitHub product that is integrated with OpenAI's Codex model to bring generative AI into the coding world. It is meant to assist programmers in writing quality code more efficiently by proposing entire lines or blocks of code as they type. Frequently, it suggests function completions, writes documentation, and even generates test cases.

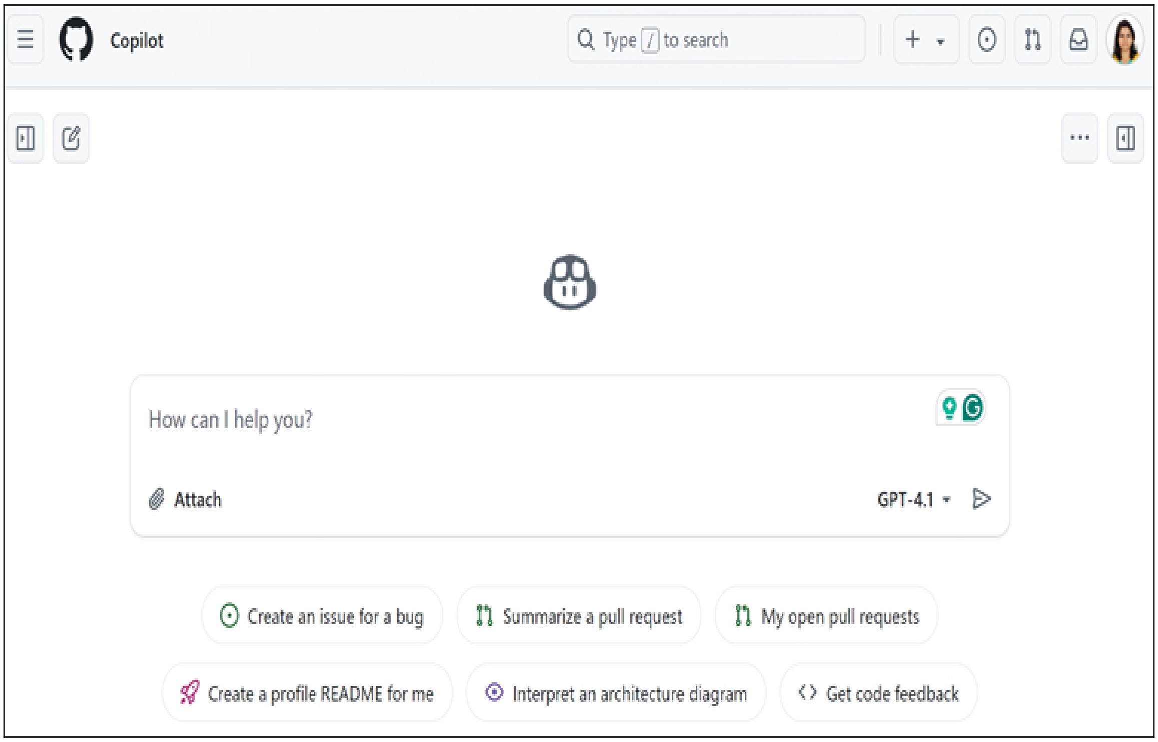
Unlike other AI-assisted coding tools, GitHub Copilot is not meant to replace developers; rather, it serves as an on-demand pair programmer. GitHub Copilot is always available, well read, and incredibly fast. It provides hands-free assistance, whether you’re building a website, writing scripts, or learning a new language. Copilot allows you to maintain focus on logical and creative aspects of your work without losing productivity.

Understand GitHub Copilot and its core functionality

GitHub Copilot is an AI-enabled code completion assistant that is integrated into your development environment. It knows the context of your code and makes suggestions, frequently predicting what you are about to type next. It can:

- Recommend multiple lines of code or even complete functions.
- Provide different ways of solving a given problem.
- Produce unit tests.
- Add documentation or comment lines in code.
- Convert code from one programming language to another.

GitHub Copilot can automatically help with dozens of languages, but it works especially well with Python, JavaScript, TypeScript, Go, C#, and several others.



The GitHub Copilot home page

How does it work?

GitHub Copilot uses Codex, a version of OpenAI's GPT large language model that has been fine-tuned for code generation. It has been trained on publicly available repositories, documentation, and forums like Stack Overflow to learn both syntax and semantics.

When you type a comment or a line of code, Copilot analyzes:

- Your current file.
- Related files in the project.
- Your prompt (code or comment).
- Programming language and framework context.

It then predicts and generates the next best lines of code as suggestions, allowing you to accept, modify, or reject them.

Tip

You can use comments as prompts to guide GitHub Copilot more effectively. For example, by typing the following, you can prompt Copilot to generate a full API call and rendering logic based on context:

```
// fetch user data from API and display
```

Why does it matter?

GitHub Copilot enhances productivity and learning by:

- Reducing boilerplate code and repetitive tasks.
- Speeding up exploration of new libraries and APIs.
- Assisting in learning new languages or frameworks.
- Improving code quality with well-structured suggestions.
- Supporting pair programming and collaborative workflows.

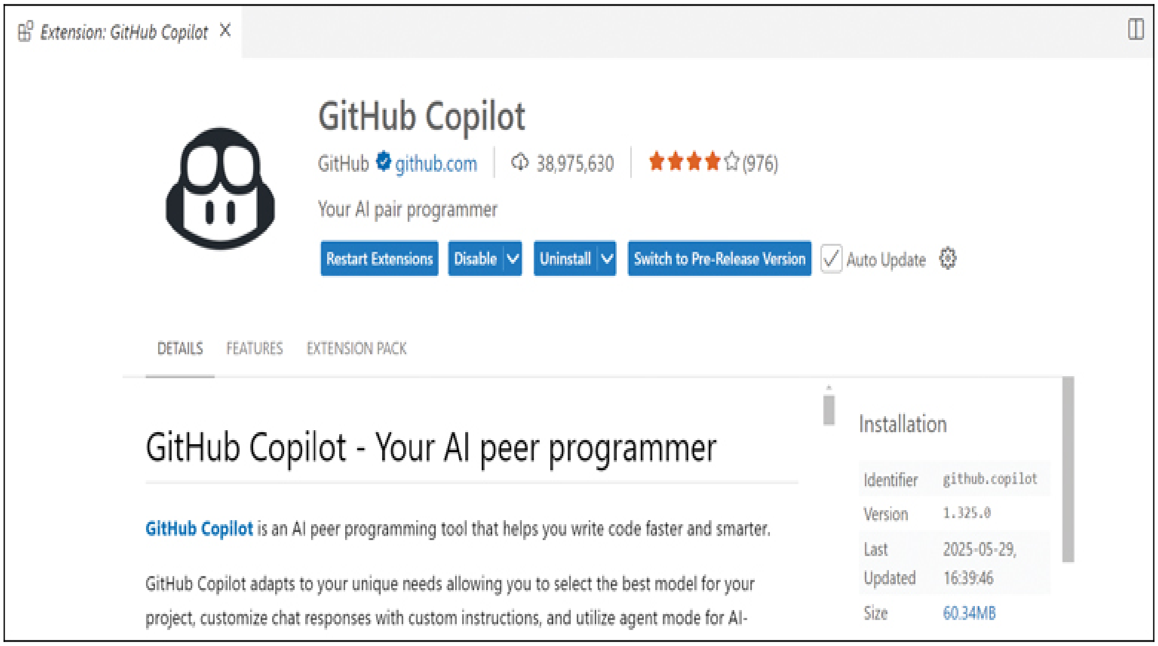
Important

GitHub Copilot is trained on public repositories, which may include code under various licenses. Always review Copilot-generated code for compliance, security, and correctness, especially when using it in commercial or open-source projects.

Where can you use it?

GitHub Copilot is available as an extension or plugin in:

- Visual Studio Code (free and open source): This is the most popular editor for Copilot, with full integration and real-time suggestions.

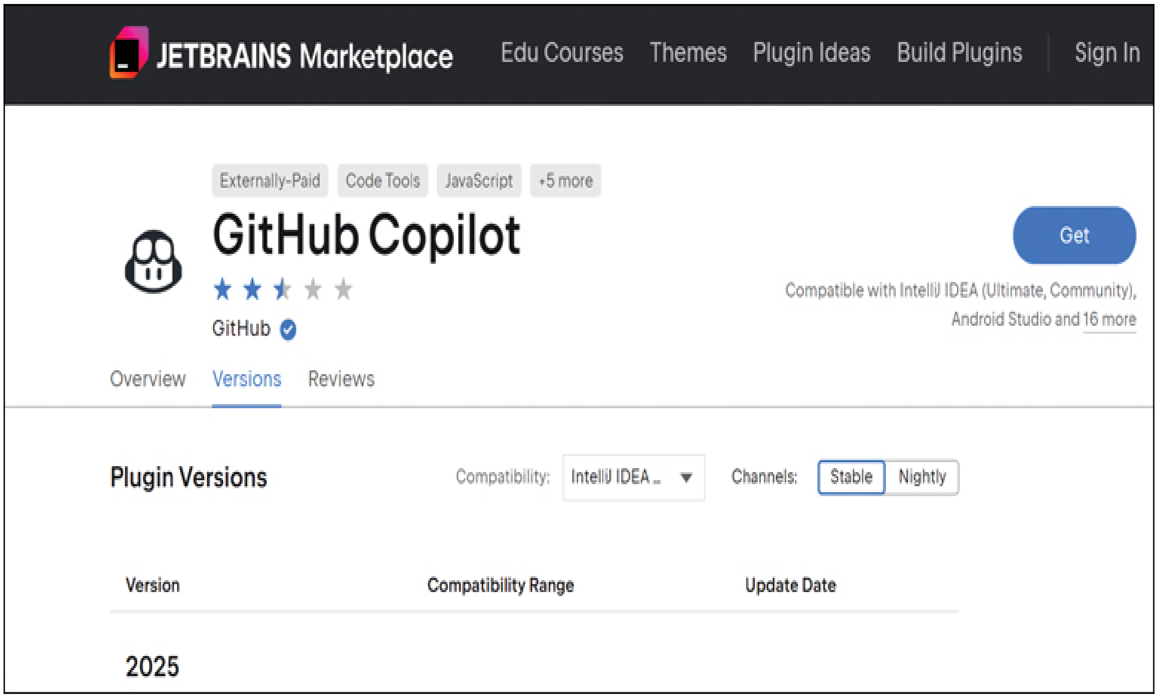


GitHub Copilot in the Visual Studio Code IDE

See Also

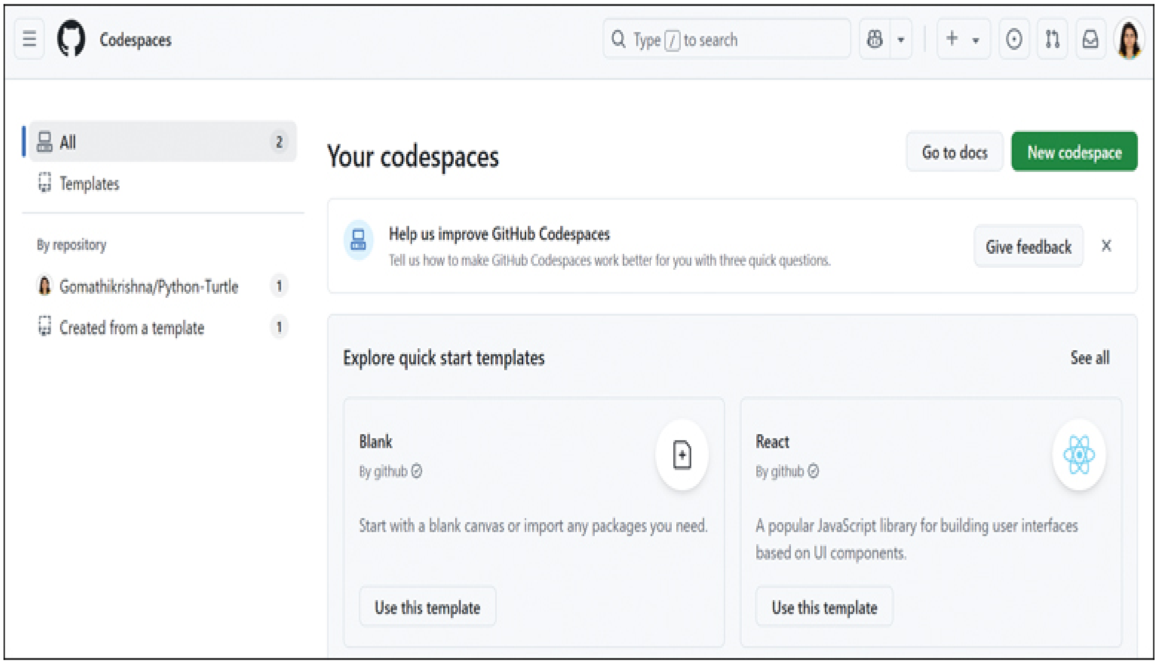
To explore how to set up Copilot in your IDE, go to Chapter 2: Setting up GitHub Copilot for your development workflow.

- JetBrains IDEs: Copilot supports in-line code completion in environments such as IntelliJ IDEA, PyCharm, WebStorm, and more.



GitHub Copilot in the JetBrains IDE Marketplace

- Neovim: Terminal lovers can use Copilot via plugins to get AI-powered suggestions directly in their workflow.
- GitHub Codespaces: Copilot works with this cloud-based environment out of the box, offering a full development environment in the browser.



GitHub Copilot in Codespaces

- Azure Data Studio: This is useful for SQL scripting, data exploration, and integrated Python notebooks.
- Xcode: macOS and iOS developers can use Copilot with Swift, Objective-C, and cross-platform tools.
- Eclipse: Java developers can also take advantage of Copilot's capabilities, using plugins for Eclipse-based environments.

Responsible AI in Copilot

Microsoft and GitHub emphasize responsible AI principles, including:

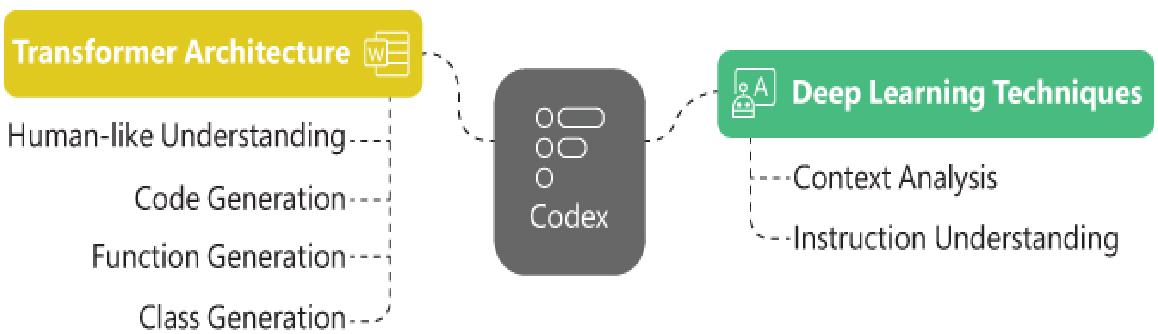
- Data privacy: Copilot doesn't retain your code or use it for training unless explicitly allowed.
- Transparency: You control when and how suggestions appear.
- User agency: You are always in charge; Copilot only assists and never replaces your decisions.

Explore the AI technology behind GitHub Copilot

GitHub Copilot might seem like magic, but behind the scenes, it's powered by cutting-edge artificial intelligence known as Codex, which was developed by OpenAI. Let's explore how this technology works, how it differs from traditional autocomplete tools, and how it continues to evolve.

What is OpenAI Codex?

At the core of GitHub Copilot is OpenAI Codex, a powerful AI model trained on vast amounts of publicly available source code, including repositories from GitHub and natural language text from books, websites, and programming documentation.



The Codex architecture

Codex is capable of:

- Understanding human instructions written in everyday language.
- Generating accurate and relevant code in response to these instructions.
- Supporting a wide range of programming languages (like Python, JavaScript, TypeScript, C#, Go, Ruby, and more).

From prompts to programs: Natural language understanding

Unlike traditional tools that need you to write some code first, Copilot can respond directly to natural language prompts. This means you can describe what you want in plain English, and Copilot will generate the code for it.

See Also

This chapter provides an introductory overview of the subject matter. For detailed guidance on prompt engineering, see Chapter 3: Writing effective prompts for AI-powered coding. For debugging methodologies, see Chapter 5: Debugging and troubleshooting code with Copilot.

Example:

Prompt:

```
// Create a function to sort a list of numbers in ascending o
```

Copilot might suggest:

```
Def sort_numbers(numbers):
    return sorted(numbers)
```

This is possible because Codex understands your intent—not just keywords or syntax. It reads your comment as a request and translates it into working code.

How Copilot is different from IntelliSense

To appreciate how powerful Copilot is, let's compare it to the widely used tool IntelliSense, which is available in Visual Studio and Visual Studio Code. IntelliSense gives suggestions based on what you've typed. It's great at reminding you of method names and completing known variables, but it has limits.

The following table provides a feature-wise comparison between traditional IntelliSense and AI-powered GitHub Copilot, highlighting their input methods, output capabilities, context understanding, creativity, language support, and learning mechanisms:

Feature	IntelliSense	GitHub Copilot
Input type	Partially written code	Natural language or code snippets
Output	Keywords, methods, and parameters	Full code blocks, loops, functions, and classes
Context awareness	Local scope	Multi-line and cross-file context understanding
Creativity	Rule based	AI-driven, context-aware suggestions
Languages supported	Language specific	Multiple languages (Python, JavaScript, Java, C++, etc.)
Learning ability	Static (rule based)	Evolving (trained via large-scale code)

Tip

Copilot doesn't just help with describing what you want in comments; it can write tests, generate documentation, and fix bugs!

How it learns: Machine learning in action

OpenAI Codex uses machine learning, a field of AI where the system improves its performance by analyzing patterns in large datasets. Codex was trained using billions of lines of code written by developers around the world.

Key points about how it works:

- It does not learn directly from your private code.
- It was trained once on publicly available code and is updated over time.
- GitHub and OpenAI continually improve the model to reduce errors, biases, and irrelevant suggestions.

See Also

For deeper insights into how Copilot evolves and adapts with real-world use, see Chapter 10: Exploring the future of AI in software development.

Continuous improvement

GitHub Copilot gets better over time. As more developers use it, GitHub collects anonymized usage data (when allowed) to refine its performance. This helps:

- Reduce hallucinations (incorrect suggestions).
- Improve relevance in specific programming scenarios.
- Expand support for more libraries, frameworks, and niche use cases.

Important

While Copilot is incredibly powerful, it's still a tool—not a human. Always review the code it generates to ensure correctness and security. Additionally, remember that Copilot is trained on publicly available code, which may include code under various licenses. Always verify that any generated code complies with your project's licensing requirements before using it in commercial or open-source work.

Identify the key benefits of GitHub Copilot for developers

Developers today face growing demands for speed, accuracy, and innovation as they are asked to write code more quickly and explore unfamiliar frameworks. GitHub Copilot rises to meet these needs by offering intelligent, context-aware suggestions that fit seamlessly into modern development workflows. This section highlights the key advantages Copilot brings to developers—from reducing repetitive tasks to supporting learning—and showcases practical, real-world scenarios where Copilot can dramatically improve coding efficiency and software quality. Whether you're building new features, cleaning up legacy code, or automating tedious steps, Copilot has use cases that align with every stage of development.

Core capabilities of GitHub Copilot

GitHub Copilot offers a wide range of intelligent features to assist developers at various stages of coding. This section provides a breakdown of its core capabilities.

Autocomplete

What it does:

Copilot automatically completes code statements based on what you're typing.

The autocomplete feature in the Visual Studio Code IDE

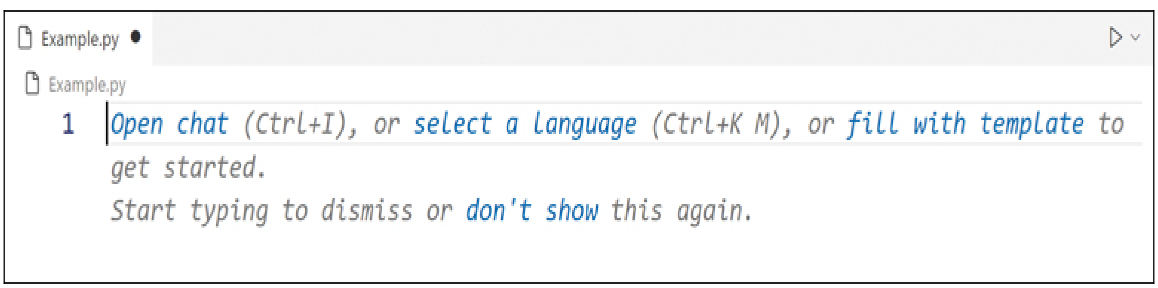
Benefit:

Saves time by reducing keystrokes and increasing development speed.

Code generation

What it does:

Copilot can generate multiple lines of code, functions, or even entire classes based on brief input. Just press Ctrl+I to start interacting with it.



Code Generation in the Visual Studio Code IDE

Example:

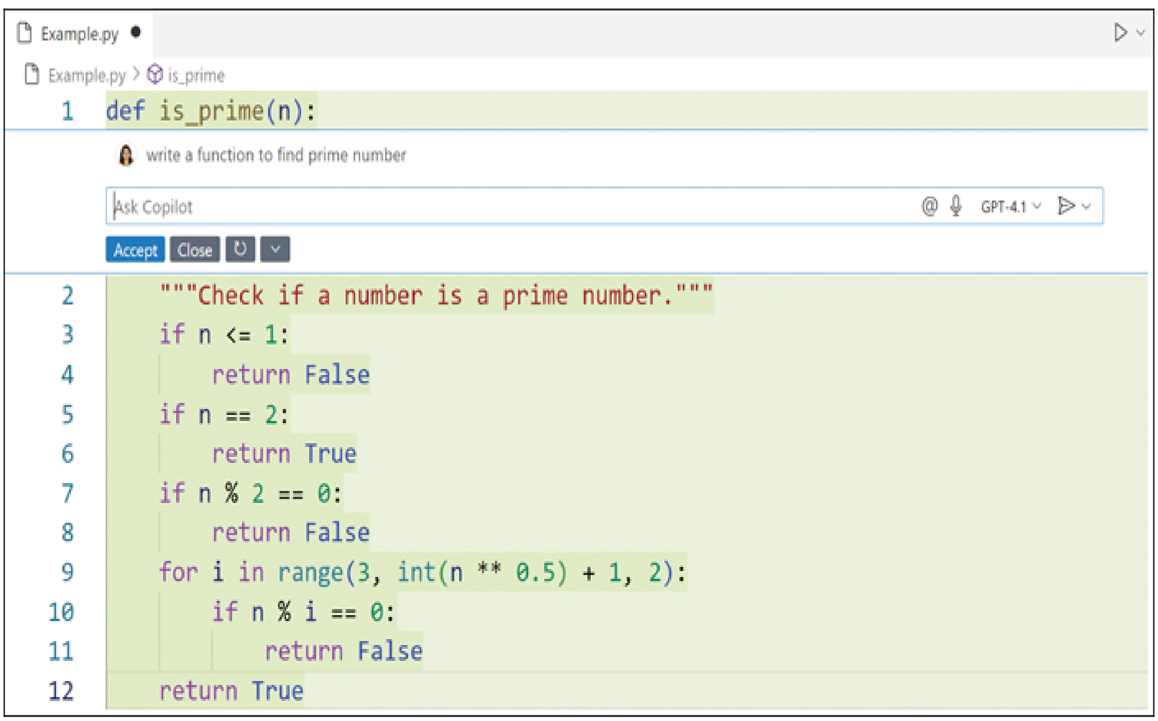
When you start Copilot by pressing Ctrl+I, you see an input bar where you can enter a natural language prompt.

The Copilot prompt in the Visual Studio Code editor

For example, you could enter this prompt:

```
write a function to find prime number
```

Once the prompt is submitted, Copilot generates the appropriate code automatically.



The code generated after entering the prompt "write a function to find prime number"

Benefit:

Helps scaffold projects or functions quickly, without requiring you to manually write boilerplate code.

Comment-to-code

What it does:

Converts natural language comments into executable code.

```python
#sort an array of numbers in ascending order
def sort_array(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

Code generated based on the comments added

Benefit:

Great for beginners and rapid prototyping. You describe what you want, and Copilot writes the code.

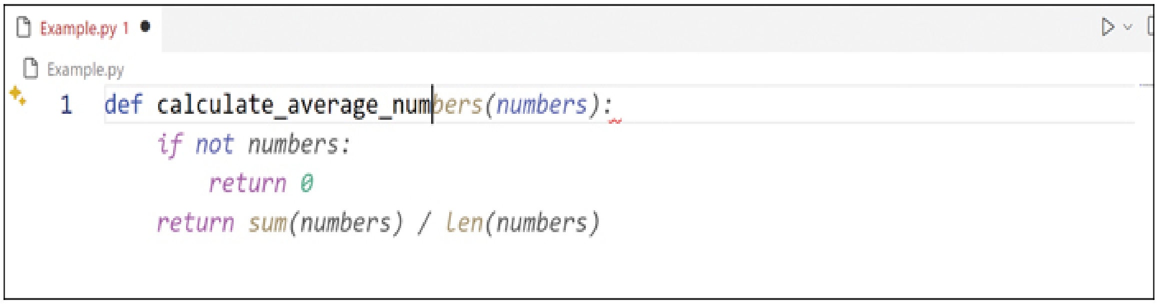
Function completion

What it does:

Automatically completes the body of a partially written function, based on the function name and parameters.

Example:

If you write def calculate_average_numbers(numbers):, Copilot may generate the logic to compute and return the average.



Function completion code in the Visual Studio Code IDE

Benefit:

Saves time on repetitive logic and gives a head start on algorithm design.

Writing tests or documentation

What it does:

Copilot can generate unit tests for your code or create structured documentation.

Example:

For a Python function, you could type the following to prompt Copilot to create unittest or pytest-compatible test cases:

```
#write unit tests can
```

```python
def calculate_average_numbers(numbers):
    if not numbers:
        return 0
    return sum(numbers) / len(numbers)
#write unit tests for the function
import unittest
class TestCalculateAverageNumbers(unittest.TestCase):
    def test_empty_list(self):
        self.assertEqual(calculate_average_numbers([]), 0)

    def test_single_number(self):
        self.assertEqual(calculate_average_numbers([5]), 5)

    def test_multiple_numbers(self):
        self.assertEqual(calculate_average_numbers([1, 2, 3, 4, 5]), 3)
```

A unit test generated for a function

Benefit:

Encourages test-driven development (TDD) and helps maintain high code quality with minimal manual effort.

The following table provides a breakdown of GitHub Copilot's core capabilities, including autocomplete, code generation, comment-to-code conversion, function completion, and automated test/documentation writing, along with a practical example for each capability.

Capability	Description	Example Use Case
Autocomplete	Completes statements or lines	Suggests console.log()
Code generation	Generates code blocks or full features	Writes REST API logic

Capability	Description	Example Use Case
Comment-to-code	Turns comments into functional code	Creates a loop from a description
Function completion	Fills in function body logic	Implements calculate_average()
Tests/docs writing	Writes test cases and documentation	Adds unit tests for a function automatically

Real-world use cases of GitHub Copilot

GitHub Copilot shines in practical coding situations where speed, accuracy, and consistency are crucial. This section presents some key scenarios where developers benefit significantly from using Copilot.

Writing a function from a comment prompt

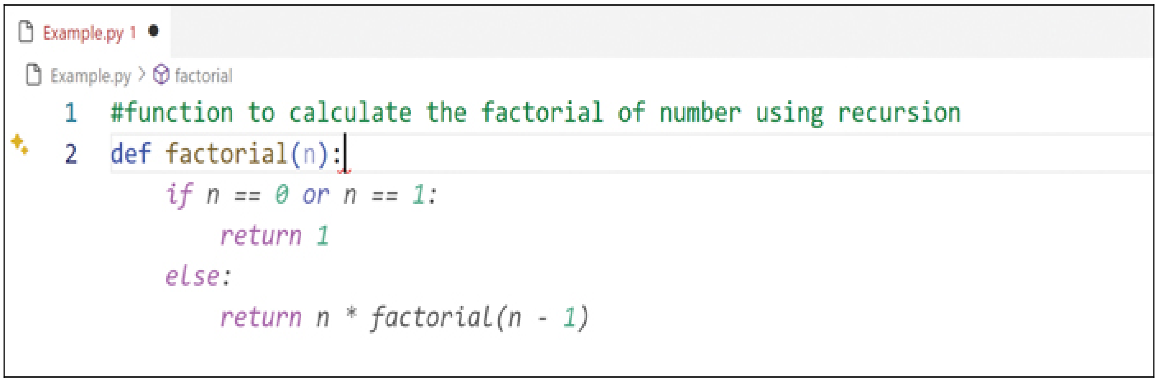
Copilot excels at transforming natural language comments into functional code. You can describe what you want in plain English, and it generates the corresponding function logic.

Example:

Prompt:

```
#function to calculate the factorial of number using recursio
```

Copilot might suggest:



The factorial program generated by GitHub Copilot

This feature is especially useful when prototyping or translating high-level requirements into working code.

> **Tip**
>
> Be specific in your . prompts The more context you provide, the better the output.

Generating tests for a function

Writing unit tests can be time-consuming. Copilot can analyze a function and suggest relevant test cases using popular testing frameworks, such as unittest, pytest, or Jest.

Example:

If you have a function like:

```
def add(a, b):
    return a + b
```

adding a comment like:

```
# Write tests for the add function using unittest
```

can prompt Copilot to generate:

```
import unittest
class TestAddFunction(unittest.TestCase):
    def test_add_positive(self):
        self.assertEqual(add(2, 3), 5)

    def test_add_negative(self):
        self.assertEqual(add(-1, -2), -3)

    def test_add_zero(self):
        self.assertEqual(add(0, 0), 0)
```

Important

Always review test logic and ensure that edge cases are covered.

Refactoring legacy code

Copilot assists in cleaning up and modernizing legacy or poorly written code. By suggesting more efficient structures, improved variable names, or updated syntax (e.g., replacing var with let/const in JavaScript), Copilot makes older codebases easier to maintain.

Example:

Old style:

```
var x = function(a, b) {
    return a + b;
}
```

Copilot might suggest:

```
const add = (a, b) => a + b;
```

Tip

Use prompts like “refactor this function” or “rewrite using modern ES6” to guide Copilot.

Writing SQL queries, Docker files, or YAML configs

Copilot is not limited to traditional programming languages. It supports structured configuration files and scripting formats, too.

SQL example:

Prompt:

```
-- Get the top 3 highest-paid employees from the employees ta
```

Copilot might suggest:

```
SELECT name, salary
FROM employees
ORDER BY salary DESC
LIMIT 3;
```

Dockerfile example:

Prompt:

```
# Dockerfile for a Python Flask app
```

Copilot might suggest:

```
FROM python:3.9
WORKDIR /app
COPY requirements.txt .
```

```
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

YAML config example:

Prompt:

```
# GitHub Actions workflow to run tests on push
```

Copilot might suggest:

```yaml
name: Run Tests
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: 3.9
      - name: Install dependencies
        run: pip install -r requirements.txt
      - name: Run tests
        run: pytest
```

Tip

When writing non-code assets, use structured comments or headers to get precise suggestions.

Bonus: Multi-language support

Copilot supports dozens of programming and scripting languages, which makes it useful for full-stack and polyglot developers.

Some commonly supported languages include:

- JavaScript/TypeScript: Frontend logic, React components, API calls
- Python: Data science, automation scripts, backend APIs
- Java: Enterprise applications, Android development
- C# and C++: Game development, system-level programming
- Go, Ruby, and PHP: Backend services, scripting
- Shell/Bash, and PowerShell: DevOps scripting
- HTML/CSS, and Markdown: UI design and documentation

Important

Support quality varies based on the richness of public code available in a language. Popular languages like JavaScript and Python get the best suggestions.

Why Copilot matters for developers

GitHub Copilot isn't just a code completion tool; it's a powerful AI assistant designed to transform the way developers write, learn, and collaborate on code. Whether you're a junior developer or a seasoned professional, Copilot brings real value to every stage of the software development lifecycle.

How Copilot can boost developers' productivity

Productivity gains: Code faster, build smarter

One of the biggest reasons developers embrace Copilot is the massive boost in productivity it provides. Copilot helps you focus on solving core problems and frees you from writing repetitive boilerplate code.

Examples:

- Copilot can autogenerate REST API routes, database models, or class templates.
- Copilot can suggest loops, error handlers, and common utility functions instantly.
- Copilot can speed up prototyping by filling in code you're about to write.

Important

The less time you spend on boilerplate code, the more time you have for business logic, debugging, and innovation.

Tip

Use Copilot as a code sketcher during hackathons or sprint planning. It's excellent for turning rough ideas into workable code quickly.

Learning aid: Great companion for junior developers

Copilot is like having a mentor by your side 24/7, which is especially helpful for those new to programming. It helps by:

- Completing partially written code.
- Suggesting idiomatic usage for unfamiliar libraries.
- Demonstrating coding patterns in real time.

Example:

You type a function definition like:

```
function validateEmail(email) {
```

Copilot suggests the entire body using regex or proper validation logic, teaching you how validation is typically done.

Important

Copilot reinforces learning through example-based exploration. This helps you understand code in context, not just theory.

Tip

Pair Copilot with documentation or video tutorials. Let it show you how code should look while you learn the "why" behind it.

Creativity boost: See coding from new angles

Developers often fall into familiar coding patterns. Copilot can help break the mold by offering alternative solutions you might not think of.

Use cases:

- Copilot can help you discover a more efficient algorithm or data structure.
- Copilot can suggest built-in methods or library functions.
- Copilot can offer syntax from other languages that solves the same problem differently.

Example:

You write a loop to flatten a nested array. Copilot might suggest using .flatMap() or recursion, offering you a new way to approach the problem.

Tip

Treat Copilot's suggestions as creative code brainstorming. Even if you don't use its suggestion, it might spark a better idea.

Important

Always review suggestions for correctness, performance, and relevance. Copilot is smart, but it's not perfect.

Collaboration and code quality: Better team practices

Copilot is more than a personal assistant. It promotes good practices that benefit the whole team, such as:

- Writing meaningful comments (which drive better suggestions).
- Reducing the chance of bugs by suggesting well-tested patterns.
- Making pair programming more efficient: One codes, and the other supervises Copilot.
- Helping maintain consistency in style and structure.

Tip

Use Copilot to write unit tests for functions you've created.

Important

A well-commented file results in higher-quality suggestions from Copilot. Make documentation a habit.

Where Copilot fits into the developer workflow

GitHub Copilot is designed to seamlessly integrate into a developer's everyday environment. It enhances—not replaces—the traditional development workflow by acting as a highly responsive, context-aware coding assistant.

Embedded in the IDE

GitHub Copilot is embedded directly into popular code editors, making it easy to incorporate into any development setup. It currently supports:

- Visual Studio Code (VS Code): This is the most popular environment for Copilot users, with native support, real-time suggestions, and frequent updates to new features.
- Visual Studio: Available in Visual Studio 2022 and later, Copilot supports full IntelliSense-like assistance for C#, C++, and more in enterprise development scenarios.
- JetBrains IDEs: Copilot offers in-line suggestions and context-aware completions for IDEs such as IntelliJ IDEA, PyCharm, and WebStorm, enhancing Java, Python, and frontend workflows.
- Azure Data Studio: Copilot can assist with writing and optimizing SQL queries, which is especially useful for data engineers and database administrators.
- Xcode: Though not natively integrated, GitHub Copilot can be accessed using community-supported plugins, assisting with Swift and Objective-C in iOS/macOS development.
- Vim/Neovim: For developers who prefer terminal-based workflows, Copilot integrates with Neovim via the copilot.vim plugin, providing lightweight and responsive AI assistance.
- Eclipse: Copilot can be integrated with Eclipse using third-party extensions, allowing Java and enterprise developers to benefit from AI-powered suggestions in their preferred environment.
- GitHub Codespaces: Copilot works in this cloud-based, containerized development environment out of the box, ensuring consistent experiences across devices and platforms.

Tip

Start with VS Code for the best Copilot experience. It has the richest support, frequent updates, and full customization options.

Language and framework support

While this book uses Python for its examples, GitHub Copilot supports a wide variety of programming languages and frameworks. Python is one of the most well-supported languages in Copilot, making it an excellent choice for demonstrating its capabilities.

GitHub Copilot is especially effective in helping with:

- Scripting and automation: From writing small scripts to automating workflows, Copilot accelerates common Python patterns.
- Data science and machine learning: Copilot can assist in writing code with libraries such as NumPy, pandas, scikit-learn, TensorFlow, and PyTorch, including data preprocessing, model training, and evaluation.
- Web development: Copilot can generate boilerplate and application logic—including views, forms, and routes—for frameworks like Django and Flask.
- API development: Copilot can help create RESTful APIs using frameworks like FastAPI or Flask-RESTful.
- Unit testing: Copilot can suggest unit test scaffolding using unittest or pytest, helping ensure code quality and test coverage.

GitHub Copilot supports other major languages in addition to Python, like JavaScript, TypeScript, Go, Ruby, C#, and C++, and it can assist with:

- Writing SQL queries and interacting with databases.
- Calling and handling APIs.
- Writing infrastructure as code (IaC) using tools like Terraform and Kubernetes YAML.
- Generating technical documentation in Markdown.

No matter your use case—from data analysis to backend development—GitHub Copilot is designed to enhance your coding experience in Python and beyond.

Live suggestions as you type

Copilot functions like an intelligent autocomplete engine—but on steroids. As you type, it predicts what you're trying to do based on context and offers one or more suggestions inline or in a pop-up. These suggestions can be short (e.g., a variable name) or long (e.g., an entire function). For example, if you type the beginning of a function, like this:

```
def is_prime(n):
```

Copilot may instantly suggest the entire logic for checking whether n is a prime number. You can scroll through alternatives using keyboard shortcuts (e.g., Alt + [/ Alt +] in VS Code).

> **Important**
>
> Copilot suggestions improve as more context becomes available. Keep your code and comments clean and descriptive.

Accepting, modifying, or rejecting suggestions

You are always in control of what gets written into your codebase. Once Copilot provides a suggestion, you can:

- Accept it as-is by pressing Tab or Enter.
- Modify it to match your coding style, performance considerations, or business logic.
- Reject it entirely and proceed manually or request another alternative using shortcuts.

This iterative process makes Copilot feel like a junior pair programmer that you can guide and correct.

> **Tip**
>
> Use Copilot for a first draft, not a final solution. Always review suggestions for correctness, edge cases, and security.

Pair programming mindset

GitHub Copilot is most effective when used with a pair programming mindset. Think of it as your virtual co-developer who:

- Suggests solutions.
- Offers alternate implementations.
- Helps with repetitive or boilerplate code.

But it still requires your expertise to validate logic, test edge cases, and understand the bigger picture.

Important

Copilot is trained on public code, which may include suboptimal or outdated patterns. You are responsible for ensuring that the final code meets your team's standards.

Tip

Treat Copilot suggestions like pull requests from a junior teammate: Review them thoughtfully and test them thoroughly.

Learn how GitHub Copilot handles privacy and applies responsible AI

As developers embrace GitHub Copilot to accelerate coding, it becomes IMPORTANT to understand how the tool handles sensitive data and aligns with responsible AI practices. This section examines the privacy safeguards that are built into Copilot, the ethical and legal considerations involved in using AI-generated code, and the shared responsibility between the tool and the developer. By exploring these aspects, you'll be better equipped to use

Copilot effectively while upholding professional standards, protecting intellectual property, and ensuring compliance with relevant policies.

Ethical considerations

As powerful as GitHub Copilot is, its use has IMPORTANT ethical and legal implications that every developer must understand. Using AI-assisted coding tools requires more than just technical awareness; it demands professional responsibility and ethical judgment.

Quick guide to common license types

GPL (GNU General Public License): Requires that derivative works are also open sourced under GPL.

MIT License: Permissive; allows reuse with attribution, including in proprietary software.

Apache License 2.0: Similar to MIT but includes explicit patent rights and attribution requirements.

BSD License: Permissive; minimal restrictions, attribution required.

Source code licensing and open-source debates

GitHub Copilot is trained in vast amounts of public code, including open-source repositories hosted on GitHub. This raises key concerns about the ownership and licensing of the code it generates:

- Some of the code suggested by Copilot may closely resemble snippets from repositories that are under specific licenses (e.g., GPL, MIT, Apache).

- Using such code without understanding or honoring the license terms could violate licensing agreements, especially in commercial projects.
- The open-source community is divided: Some welcome Copilot as an innovation, while others see it as exploitation of their contributions without proper attribution.
- Developers must ensure that the code they accept from Copilot is either original or properly licensed for their use. Copilot itself does not guarantee license compliance.

Responsibility: The developer is accountable

While Copilot can assist, a human developer remains fully responsible for any code written or deployed. AI suggestions are based on patterns, not intent or domain understanding.

In the event that AI-generated code results in bugs, security vulnerabilities, or legal issues, it is the developer who bears responsibility, not the tool. Code reviews, testing, and adherence to engineering standards remain essential.

> **Important**
>
> Never deploy code generated by Copilot without a thorough review. The presence of AI support does not reduce your professional liability.

Copilot's usage guidelines and best practices

GitHub has provided the following best practices and usage guidelines for using Copilot ethically and safely:

- Avoid blind trust: Always review, understand, and test suggestions before accepting them.

- Use responsibly: Do not allow Copilot to write entire modules or security-sensitive code without close oversight.
- Be aware of sensitive data: Copilot may unintentionally generate credentials, tokens, or private keys if it has seen similar patterns before. Never use such output as-is.
- Cite or attribute when necessary: If Copilot generates code that resembles known sources, proper attribution should be considered, especially in public or collaborative projects.
- Treat Copilot like a junior developer: Use it as a helpful assistant, but provide close supervision and frequent corrections

Recognize the limitations and ethical considerations of GitHub Copilot

Understanding both the advantages and constraints of GitHub Copilot is key to using it effectively and responsibly.

Benefits

GitHub Copilot brings several advantages that can streamline development and improve coding efficiency. From reducing repetitive tasks to inspiring creative solutions, it serves as a versatile tool that supports developers of all skill levels. In the following subsections, we'll explore how Copilot speeds up common tasks, minimizes context switching, encourages experimentation, and provides valuable learning opportunities.

Speeds up development

Copilot can generate boilerplate code, common patterns, and repetitive logic instantly, saving developers significant time.

Example:

If you type a loop comment like:

```
# loop through a list of users
```

Copilot can instantly generate a for loop structure.

Tip

Use Copilot to handle routine code while focusing your energy on complex, business-critical logic.

Reduces context switching

Instead of searching Stack Overflow or documentation, you can stay within your IDE as Copilot offers suggestions in real time.

Important

This minimizes interruptions and helps maintain flow, especially during deep work sessions.

Enhances creativity

Copilot can offer alternative approaches you might not have considered, such as:

- Different algorithmic implementations.
- Use of less familiar APIs or functions.
- Clean or idiomatic coding styles.

Tip

> Use Copilot for brainstorming solutions or reviewing new ways to approach a problem.

Provides learning support

Junior developers can learn coding patterns, syntax, and common best practices through the suggestions Copilot makes.

Example:

By simply typing a function name or a comment, new developers can observe how error handling or API interactions are structured.

> **Important**
>
> Treat Copilot like an on-demand mentor that is great for learning but whose results need to be verified.

Limitations

While Copilot is a powerful aid, it is not infallible. Its suggestions are generated from patterns in public code, and it lacks true understanding of your project's unique requirements. This means developers must remain vigilant, reviewing AI-generated code for accuracy, security, and compliance. The following subsections outline key limitations and why human judgment is essential when incorporating Copilot's output into production code.

Not always accurate

Copilot predictions are based on patterns, not real understanding. Copilot may:

- Misinterpret your intent.

- Generate nonworking or incorrect code.
- Confuse similarly named functions or libraries.

Important

Always test and debug code generated by Copilot. Never assume that it's perfect.

Lacks business context

Copilot doesn’t know your project’s specific business rules, architecture, or domain knowledge.

Example:

It might suggest deleting a record directly, without checking your company’s data retention policies.

Tip

Don’t rely on Copilot for logic tied to compliance, business processes, or sensitive operations.

Sometimes suggests insecure or outdated code

Because it’s trained on publicly available code, Copilot may:

- Propose insecure coding practices (e.g., no input validation).
- Use deprecated libraries or APIs.
- Copy license-restricted code.

Important

Review suggestions for security vulnerabilities and compliance with modern standards.

Requires human judgment

Copilot is a tool, not a decision maker. A developer must:

- Interpret the relevance of a suggestion.
- Adapt suggestions to their needs.
- Maintain accountability for the final code.

Tip

Use Copilot as a productivity aid—not as a substitute for experience or good software engineering principles.

Skills review

In this chapter, you learned how to:

- Understand what GitHub Copilot is and how it works as an AI-powered coding assistant developed by GitHub and OpenAI.
- Explore the core capabilities of Copilot, including code autocompletion, comment-to-code transformation, function generation, and test creation.
- Learn about the technology behind Copilot, including OpenAI Codex, and how it interprets natural language prompts into code.
- Recognize the importance of Copilot in improving developer productivity, learning, creativity, and collaboration.
- Identify where Copilot fits into your workflow, including integration with Visual Studio Code, JetBrains, Neovim, and GitHub Codespaces.

- Discover real-world use cases such as writing functions, generating tests, refactoring code, and creating SQL queries and configuration files.
- Evaluate the benefits and limitations of using Copilot in daily development, including performance, accuracy, and security awareness.
- Understand ethical considerations, such as source code licensing, developer accountability, and the importance of reviewing AI-generated suggestions.
- Learn the steps to get started with Copilot and understand where and how it can be used effectively.
- Explore how GitHub is embedding responsible AI principles in the design and deployment of Copilot.

Practice tasks

No practice files are necessary to complete the practice tasks in this chapter. Follow the steps below to reinforce the concepts and tools introduced in each section.

Understand GitHub Copilot and its core functionality

To get started with GitHub Copilot, follow these steps:

1. Open your web browser and go to the official GitHub Copilot page: https://github.com/features/copilot.
2. Sign in using your GitHub account credentials. If you don’t have an account, select Sign Up and create one.
3. Click Get Started for Free.

4. Explore the GitHub Copilot Playground for free.

Explore the AI technology behind GitHub Copilot

Open a browser and Visual Studio Code and complete the following tasks:

1. Navigate to https://openai.com/index/introducing-codex/ (Open AI documentation).
2. Read how Codex interprets prompts and generates code from natural language.
3. In VS Code, compare GitHub Copilot with IntelliSense:

   a. Type import math and then . to observe IntelliSense suggestions.

   b. On the next line, write the following prompt and observe Copilot's suggestion:

```
# calculate square root.
```

4. In a new .js file, write a comment to generate an event listener function and review the code that Copilot suggests.
5. Reflect on the differences in behavior between IntelliSense and Copilot.

Identify the key benefits of GitHub Copilot for developers

Open Visual Studio Code and a new project folder and complete the following tasks:

1. Use Copilot to write boilerplate code, such as a class definition (class Student:) or API call (# fetch data from URL).

2. Add a new function (e.g., def calculate_average(scores):) and let Copilot complete the logic.
3. Write the following prompt and accept the tests that Copilot generates:

```
# write unit tests for calculate_average
```

4. Observe how Copilot helps with reducing manual effort, suggesting naming conventions, and filling in logic.
5. Make a list of the top three time-saving suggestions you received from Copilot in this exercise.

Learn how GitHub Copilot handles privacy and applies responsible AI

Open GitHub Copilot settings in VS Code and complete the following tasks:

1. Open GitHub Copilot's privacy documentation by visiting https://docs.github.com/en/site-policy/privacy-policies/github-general-privacy-statement.
2. Read about how Copilot handles user data and when code is retained for model improvement.
3. Consider and write down one scenario in your own development work where privacy could be a concern (e.g., working with medical or financial data).
4. Try disabling Copilot temporarily and then re-enable it to understand your control options.

Recognize the limitations and ethical considerations of GitHub Copilot

Open Visual Studio Code and complete the following tasks:

1. Type the following comment in HTML and observe the generated suggestion:

```
# login form
```

2. Evaluate the output:

   a. Does it include proper security measures (e.g., input validation)?

   b. Is the logic sound?

   c. Open a new JavaScript file and type the following prompt and then look for whether outdated methods or insecure patterns are suggested:

```
// connect to a database
```

3. Open GitHub's guidelines on responsible AI by visiting https://learn.microsoft.com/en-us/training/modules/responsible-ai-with-github-copilot/ and review the ethics best practices.

4. Discuss or journal: Would you feel comfortable using Copilot to write production code? Why or why not?

5. Try prompting Copilot with a vague instruction (e.g., # handle user data) and analyze whether the result reflects secure and ethical practices.

2

Setting up GitHub Copilot for your development workflow

In this chapter

- Set up your GitHub account and select a Copilot plan
- Install Copilot in your development environment
- Configure permissions and personal settings
- Test Copilot with sample prompts across languages
- Troubleshoot installation issues and common errors
- Apply best practices for using Copilot effectively
- Explore Copilot’s functionality in online and offline modes

Practice files

There are no practice files for this chapter.

Getting started with GitHub Copilot begins with setting up the right environment for your development needs. Whether you're a beginner exploring AI-powered coding for the first time or a seasoned developer integrating Copilot into an existing workflow, the setup process lays the foundation for a smooth and productive experience. This chapter walks you through everything you need to get Copilot up and running—from choosing a suitable subscription plan to installing Copilot in popular IDEs like Visual Studio Code, JetBrains IDEs, and GitHub Codespaces.

You'll also learn how to configure Copilot to suit your coding preferences, verify that it's working properly, and write your first AI-assisted code. The chapter includes tips on troubleshooting common issues and using best practices to ensure a successful onboarding experience. Whether you're working online or offline, you'll gain a clear understanding of how to optimize your setup to make the most of what Copilot has to offer.

Set up your GitHub account and select a Copilot plan

Before you begin using GitHub Copilot, you need to set up a GitHub account and choose an appropriate subscription plan that fits your development needs. Whether you're a student trying Copilot for the first time or a professional exploring AI-powered development, this section guides you through the account creation process, explores available pricing tiers, and explains eligibility for free access to premium features. With your account and plan in place, you'll be ready to integrate Copilot into your development environment.

Prerequisites and account setup

Before you can start coding with GitHub Copilot, there are a few key steps to complete. This section guides you through setting up your GitHub account, choosing the right plan for your needs, and understanding the differences between free trials and paid subscriptions.

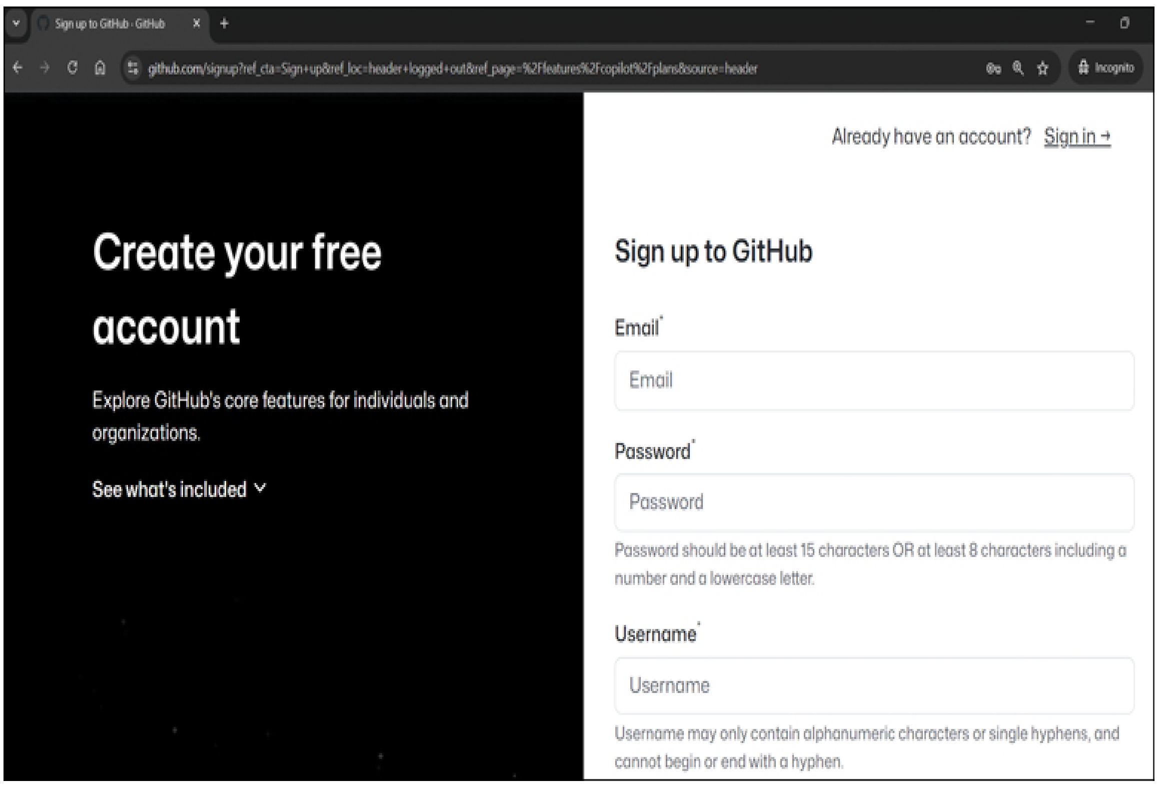
See Also

The next section walks you through installing GitHub Copilot in your preferred development environment—whether on the desktop or in the cloud.

Creating a GitHub account

GitHub Copilot requires a GitHub account. If you don't have one already, follow these steps:

1. Visit https://github.com.
2. Click on Sign Up.
3. Enter your email address, password, and username.



The GitHub free account sign-up page

4. Choose whether to receive updates and offers and solve the CAPTCHA.
5. Verify your email address by clicking the link GitHub sends to your inbox.

6. Complete your profile and preferences. (This is optional but recommended for new developers.)

Once you're registered, you'll have access to all of GitHub's features, including repositories, issue tracking, project management tools, and, of course, GitHub Copilot.

> **Tip**
>
> You can use the same GitHub account across multiple devices and IDEs. If you're a VS Code user on both desktop and the web, your settings will sync automatically when you're logged in.

> **Important**
>
> While GitHub Copilot supports multiple IDEs, all examples and demonstrations in this book are provided using Visual Studio Code (VS Code). This ensures consistency and ease of replication for most developers, as VS Code is free and widely adopted and has seamless Copilot integration.

GitHub Copilot plans: Free vs. paid

GitHub Copilot now offers three distinct subscription tiers to suit individual learners, developers, and power users.

The following table details the key differences between GitHub Copilot's Free, Pro, and Pro+ plans, including pricing, feature access, and AI model availability. Please note that features and availability may change over time. For the most up-to-date plan comparison, visit the official GitHub Copilot plans page: https://docs.github.com/en/copilot/get-started/plans-for-github-copilot.

Feature	Free	Pro (Most Popular)	Pro+
Cost	$0 USD	$10/month or $100/year	$39/month or $390/year
Best for	Beginners, casual users	Most developers, students, individual pros	Advanced users, power coders, AI tinkerers
Agent/chat access	50 requests/month	Unlimited agent mode and chats with GPT-4o	Access to all models, including GPT-4.5
Completions	2,000/month	Unlimited code completions	30 times more premium requests than the Free plan
Models available	Claude 3.5, Sonnet, GPT-4o	All in the Free plan plus Claude 3.7 Sonnet, o1, and more	All in the Pro plan plus and GPT-4.5
Premium model usage	Limited	6 times more requests for latest models than with the Free plan	30 times more requests than with the Free plan; upgradable
Free trial	Always free	30-day free trial available	No free trial

Tip

Students, teachers, and verified maintainers of popular open-source projects can access the Pro plan for free under GitHub Education benefits.

Which plan should you choose?

Which GitHub Copilot plan is right for you depends on your development goals and frequency of use, as well as the level of AI capabilities you need. Here's a quick guide to help you decide:

- Choose the Free plan if you're just exploring Copilot, experimenting, or coding occasionally.
- Upgrade to Pro if you're coding frequently and need more completions, advanced model access, and unlimited usage.
- Go for Pro+ if you want full model access, including GPT-4.5, and you work with large-scale or high-performance AI tasks.

> **Important**
>
> The Pro+ plan is best for enterprise developers and advanced individual users who want maximum AI capability with minimal restrictions. It offers significantly more compute and model access.

Getting free access to GitHub Copilot Pro as a student, teacher, or open-source maintainer

GitHub Copilot Pro is a premium AI coding assistant that brings additional features like enhanced suggestions, chat support (in some editors), and priority access to infrastructure. While it's generally available via paid subscription, GitHub offers free access to Copilot Pro for a select group of eligible users, including verified students, teachers, and open-source maintainers, in recognition of their contribution to the tech community.

How to qualify for free access to GitHub Copilot Pro

The following table outlines the eligibility criteria for students, educators, and open-source maintainers to access GitHub Copilot Pro for free.

User category	How to qualify
Verified student	Students can apply at https://education.github.com/students. Proof of academic enrollment (such as a student email address or documents) is required.
Verified teacher	Educators can apply at https://education.github.com/teachers. Verification of teaching status is required.
Open-source maintainer	Maintainers of widely used public repositories may qualify. GitHub automatically evaluates repository popularity and contribution metrics. You can check your eligibility at https://github.com/settings/copilot.

Tip

GitHub reevaluates eligibility monthly. So, if your student or maintainer status changes, your access might be revoked unless reverified.

What if you're not eligible?

If you don't meet the free access criteria, you still have two solid options:

- Try GitHub Copilot Pro free for 30 days: Anyone can sign up for a one-time 30-day free trial. After the trial ends, you'll need to subscribe to a paid plan to continue using Pro features.
- Use the GitHub Copilot Free plan: While more limited in functionality, the free Copilot experience still offers AI coding

suggestions within select IDEs. It’s a good starting point if you're exploring.

How to access Copilot Pro for free (if eligible)

Follow these steps to activate your free Copilot Pro access:

1. Sign in to GitHub by going to github.com and logging in with your GitHub credentials.
2. Go to your Copilot settings. In the upper-right corner, click your profile picture and select Your Copilot from the dropdown.
3. Check for eligibility. If you qualify for free access, you’ll see the message “GitHub Copilot Pro” informing you of your eligibility.
4. Activate access by clicking the Get Access to GitHub Copilot button and then configure your usage preferences (e.g., telemetry settings, suggestion behavior), and click Save and Complete.

> **Important**
>
> You must be signed in to the same GitHub account that has been verified as a student, teacher, or maintainer account. The system will not recognize eligibility across multiple accounts.

Install Copilot in your development environment

GitHub Copilot is compatible with a wide range of integrated development environments (IDEs), enabling developers to integrate AI-assisted coding seamlessly into their existing workflows. From web-based editors to powerful desktop environments, Copilot enhances productivity across languages and platforms. The following table lists the supported IDEs and briefly describes each of them, highlighting how Copilot functions within these environments. The sections after the table provide further information on the IDEs.

IDE Name	Description
GitHub Codespaces	Supports Copilot in browser via GitHub.dev or Codespaces for quick editing.
Visual Studio Code	Provides full-featured support for Copilot, including inline suggestions and Copilot Chat.
Visual Studio	Works with Visual Studio 2022+ for .NET, C#, VB.NET, and enterprise development.
VS Code for Web	Supports browser-based use of Copilot with Provides VS Code (GitHub.dev) for light development.
JetBrains IDEs	Includes IntelliJ, PyCharm, WebStorm, etc. and supports Copilot via official plugin.
Neovim	Provides an advanced text editor with Copilot using community plugins.
Azure Data Studio	Enables SQL/database development through use of Copilot with extensions.
Eclipse	Enables the Java IDE with Copilot support via a plugin. (This is less seamless than VS Code.)

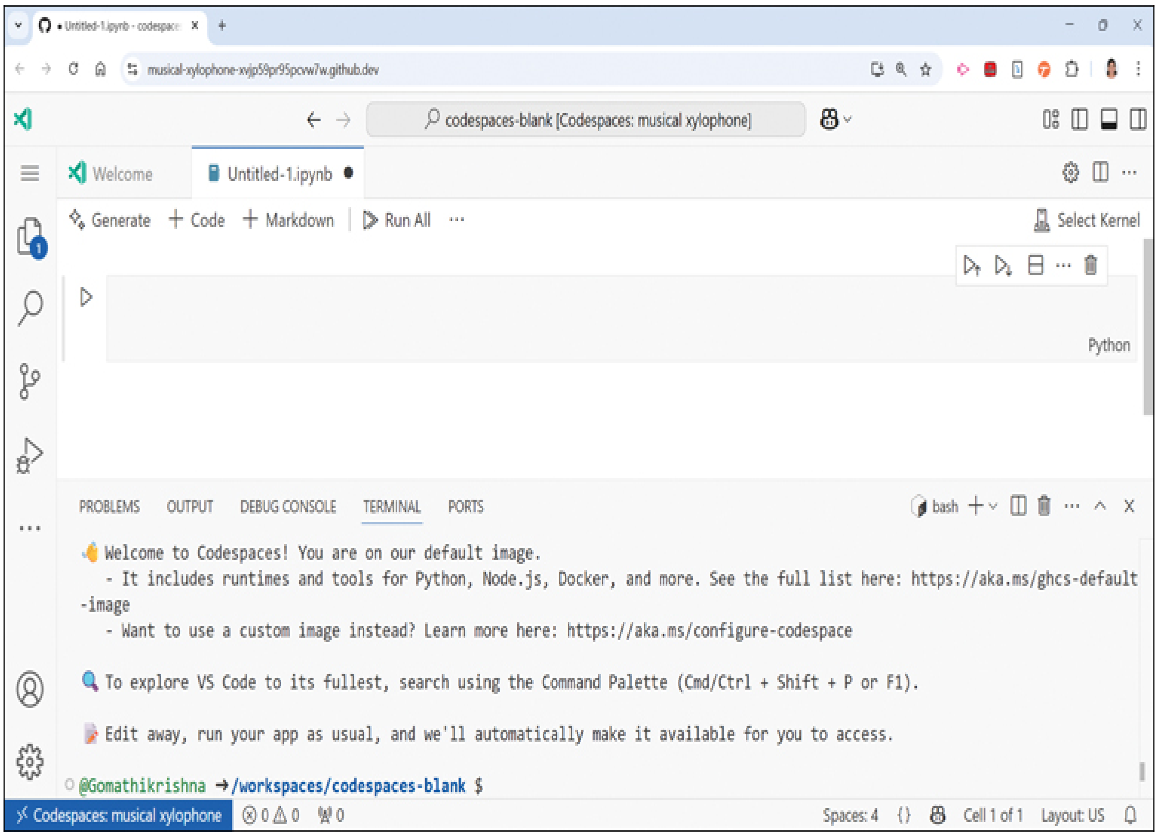
GitHub Codespaces

Installation required: No

Access method: Cloud based

How to use:

1. Create or sign in with a GitHub account that has an active Copilot subscription or trial.
2. Open https://github.com/codespaces In your browser.
3. Launch or create a Codespace. Copilot will already be integrated and available in this cloud-hosted development environment.



The GitHub Codespaces environment, displaying a Jupyter Notebook interface with Python kernel and welcome terminal instructions

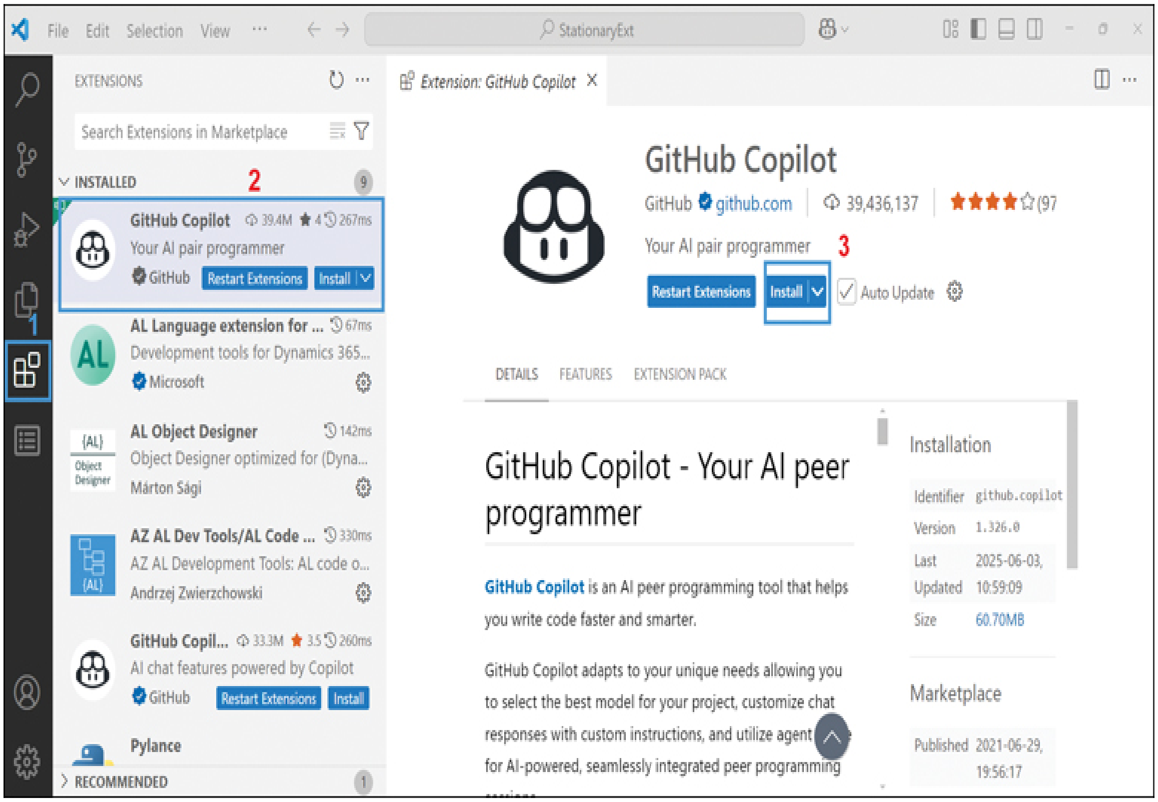
Visual Studio Code (VS Code)

Installation required: Yes

Access method: Desktop app

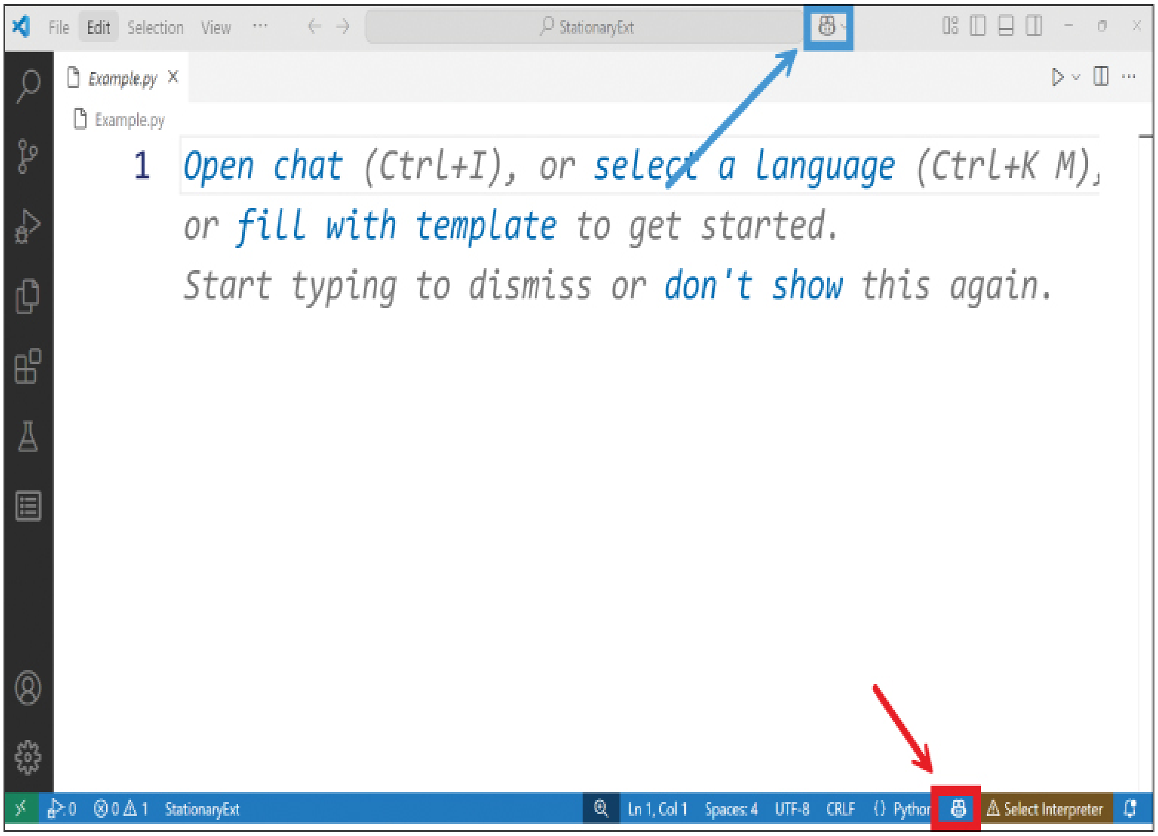
How to use:

1. Download and install VS Code from https://code.visualstudio.com.
2. Open VS Code and select the Extensions panel.
3. Search for GitHub Copilot.
4. Click Install as shown in the following image



The GitHub Copilot Extensions panel in Visual Studio Code, highlighting the Install button, extension details, and version information

5. Sign in with your GitHub account to activate Copilot.
6. Begin coding. Copilot suggests completions as you type.



Visual Studio Code interface, showing the GitHub Copilot icons highlighted in both the top menu and bottom status bar, indicating ways to launch the AI assistant

Visual Studio (2022 and later)

Installation required: Yes

Access method: Desktop app

How to use:

1. Install Visual Studio from https://visualstudio.microsoft.com.
2. Go to Extensions, search for GitHub Copilot, and install it.
3. Sign in with your GitHub account to enable Copilot.

4. Start coding in supported languages (e.g., C#, VB.NET). Copilot provides suggestions.

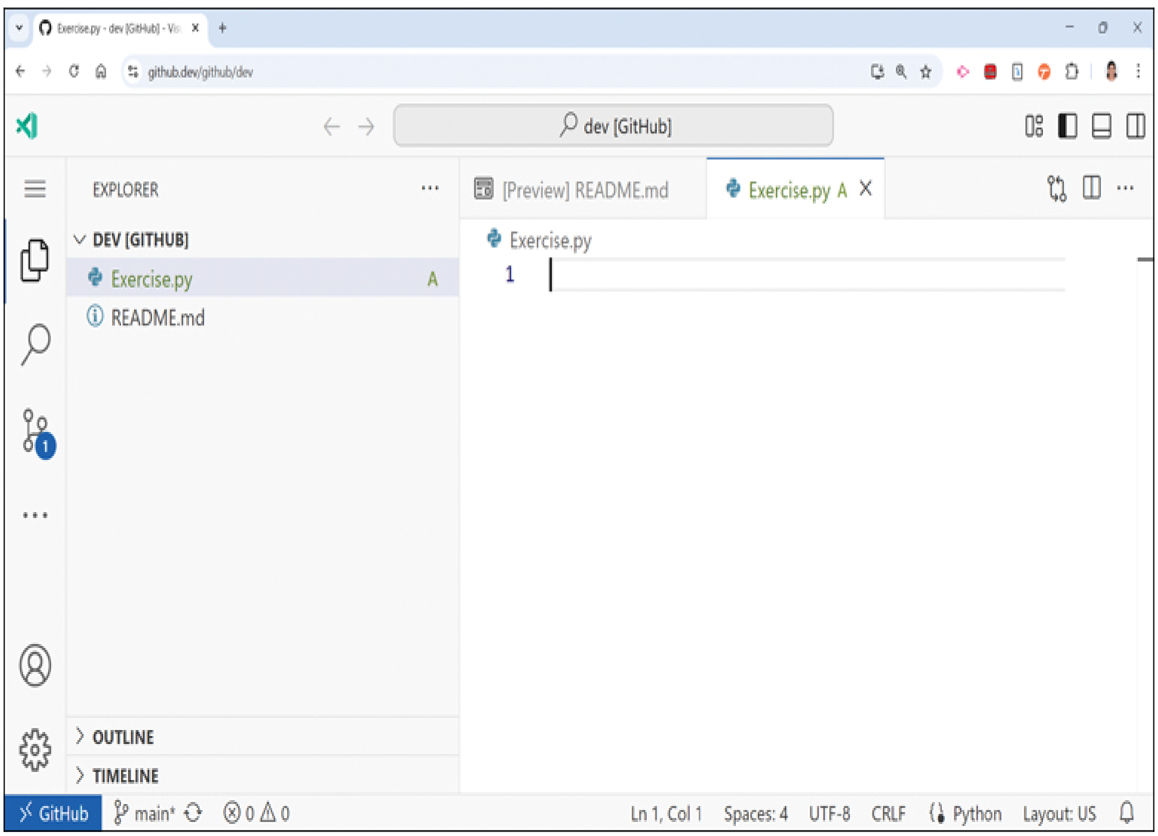
VS Code for Web (GitHub.dev)

Installation required: No

Access method: Cloud based

How to use:

1. Go to https://github.dev and create or sign in with a GitHub account.
2. Navigate to any GitHub repository



The Visual Studio Code browser-based interface at github.dev, with a Python file open in the editor and the file explorer displaying the project structure

JetBrains IDEs (IntelliJ IDEA, PyCharm, WebStorm, etc.)

Installation required: Yes

Access method: Desktop app

How to use:

1. Download the JetBrains IDE of your choice from https://www.jetbrains.com.
2. Go to Settings > Plugins, search for GitHub Copilot, and click Install.
3. Log in with your GitHub account to activate Copilot. Copilot suggestions appear inline as you write code.

Neovim

Installation required: Yes

Access method: Terminal/desktop

OS: Linux

How to use:

1. Download and install Neovim from https://neovim.io/
2. Install the github/copilot.vim plugin using a plugin manager (such as vim-plug).
3. Launch Neovim and authenticate Copilot through terminal login. Start coding; Copilot will suggest completions inline as you type.

Azure Data Studio

Installation required: Yes

Access method: Cloud based

How to use:

1. Download Azure Data Studio from https://learn.microsoft.com/en-us/azure-data-studio/.
2. Install the Azure Data Studio in Visual Studio Code.
3. Go to Extensions, search for GitHub Copilot, and click Install.
4. Sign in with your GitHub account.
5. Begin working with SQL. Copilot provides AI assistance.

Eclipse

Installation required: Yes

Access method: Desktop app

How to use:

1. Download Eclipse from https://www.eclipse.org.
2. Go to Help > Eclipse Marketplace, search for GitHub Copilot, and install the plugin.
3. Log in with your GitHub account.
4. Start coding. Copilot provides AI-powered suggestions for Java and other supported languages.

Configure permissions and personal settings

Once GitHub Copilot is installed and running in your IDE, you can tailor its behavior to match your development style and project needs. GitHub offers a variety of customization options that help improve usability, reduce distractions, and maintain control over your development environment. This section walks you through the key settings you can adjust, including how suggestions are presented, how to control Copilot on a per-language basis, and how your data is handled, how to change suggestion behavior (e.g., number of suggestions, inline vs. in a panel), and how to use keyboard shortcuts with GitHub Copilot.

GitHub Copilot offers a set of intuitive keyboard shortcuts that are designed to streamline and accelerate your coding workflow. These shortcuts allow you to interact with Copilot's AI-generated suggestions more efficiently, helping you accept, reject, explore alternatives, and manage how suggestions appear in your editor. Knowing these shortcuts enhances productivity and also reduces the friction between ideation and implementation.

The shortcuts may vary slightly depending on the IDE you're using (such as VS Code, JetBrains IDEs, or Neovim), but most of the commonly used shortcuts are available in Visual Studio Code, which is the IDE used throughout this book.

> **Important**
>
> The ability to cycle through suggestions using Ctrl+] or Ctrl+[may no longer be available in the latest VS Code Copilot extension. Instead, you can manually trigger suggestions using Alt+/.

Let’s say you're writing a function, and Copilot offers a suggestion. To accept it, all you need to do is press the Tab key. If the suggestion isn’t helpful, press Esc to dismiss it.

To boost productivity, GitHub Copilot includes several keyboard shortcuts. The following table lists and describes the most useful shortcuts.

Action	Keyboard Shortcut	Notes
Accept current suggestion	Tab	Inserts the AI-generated code suggestion inline.

Action	Keyboard Shortcut	Notes
Reject current suggestion	Esc	Dismisses the current suggestion.
Trigger Copilot manually (inline)	Alt+/	Useful when Copilot doesn’t auto-suggest or when starting a new line.
Show Copilot Labs feature list	Ctrl+Shift+Alt+E	Opens experimental tools and enhancements from Copilot Labs (if enabled).
Accept suggestion word-by-word	Ctrl+→	Accepts the suggestion one word at a time.
Show multiple suggestions (panel view)	Ctrl+Enter	Opens a list of alternative completions (if supported).
Toggle Copilot sidebar	Ctrl+Shift+A	Opens or closes the Copilot sidebar in VS Code.
Show next Copilot suggestion	Alt+]	Cycles to the next suggestion (only if cycling mode is available).
Show previous Copilot suggestion	Alt+[	Cycles to the previous suggestion.

Tip

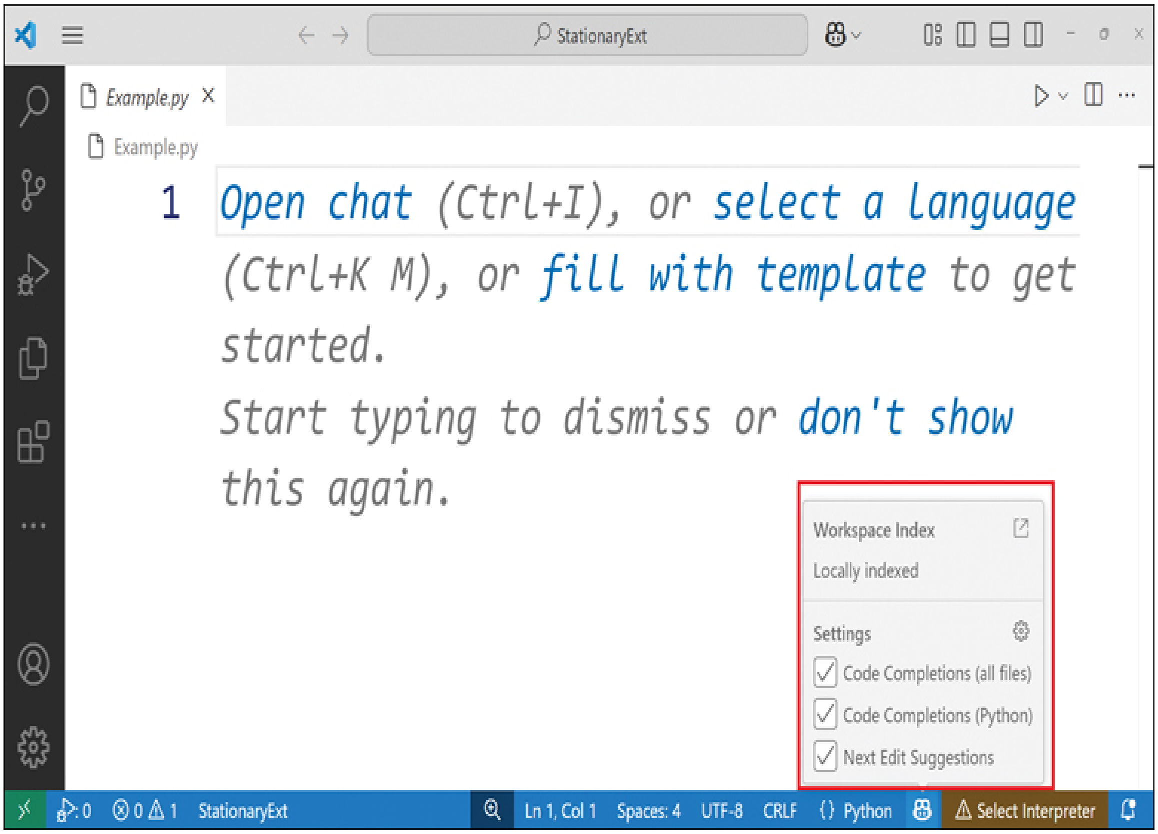
You can customize or view these keyboard shortcuts in Visual Studio Code by navigating to File > Preferences > Keyboard Shortcuts and searching for Copilot. Keeping these shortcuts at your fingertips ensures a smoother, faster, and more productive coding experience with GitHub Copilot.

Exploring the GitHub Copilot interface in VS Code

GitHub Copilot's integration with Visual Studio Code isn't just about code suggestions. VS Code with GitHub Copilot provides a complete interface that empowers you to control when, how, and where suggestions appear. Next, we will look at the key elements of the interface along with practical use cases.

Copilot in the status bar: Real-time control over code suggestions

The Copilot status bar icon, located at the bottom right of the VS Code window (next to the language selector), gives you access to quick toggles and settings. Let’s look at the options available when you click the status bar icon and consider some use cases.



Visual Studio Code, showing a Copilot-related settings pop-up, including code completion and next edit suggestions, with workspace indexing set to locally indexed

Locally Indexed (Workspace Index)

What it does: Tells you whether the Copilot suggestions are enhanced with your current project files.

Use case: When you're working in a large codebase like a Django project, local indexing helps Copilot provide more context-aware suggestions, such as referencing existing models, views, or utility functions.

Example:

```
def get_user_email(user_id):
```

Copilot suggests:

return User.objects.get(id=user_id).email

````
```
````

Because your project is indexed, Copilot knows you're using Django ORM.

Code Completions (All Files)

What it does: Enables Copilot suggestions across all file types.

Use case: You're working in a full-stack project with both Python and JavaScript files, and you want Copilot to help in both.

Example: In app.js, you begin typing a fetch() call, and Copilot suggests a full API call with error handling.

Code Completions (Python)

Use case: You're preparing for a data science assignment and want Copilot's help specifically in .py files but not in README files.

Example:

```
import pandas as pd
df = pd.read_csv(
```

Copilot suggests available CSV files in the project folder.

Next Edit Suggestions

What it does: Suggests what to do next after a code change.

Use case: You just wrote a new class, and Copilot immediately recommends test cases or docstrings.

Example:

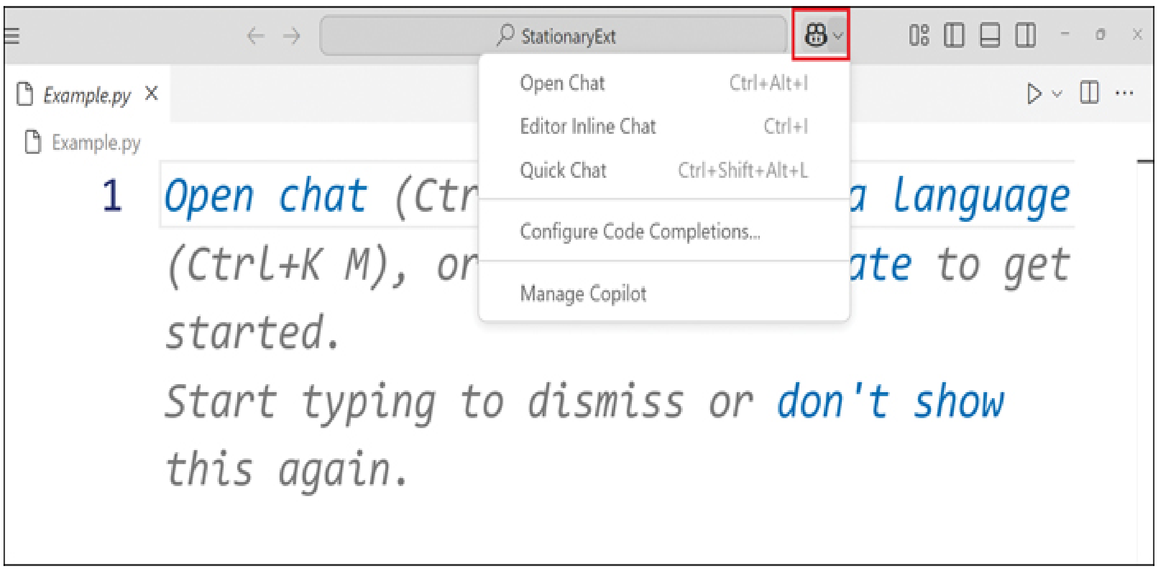
```
class InvoiceProcessor:
    def process(self):
        pass
```

Copilot suggests:
```

```
# def test_invoice_processor():
#     processor = InvoiceProcessor()
#     ...
```

Top bar Copilot menu: Interact, chat, and configure

At the top right of VS Code (next to the split editor and layout icons) is the Copilot icon, which you can click to open the GitHub Copilot menu. Next, we'll look at the options available in this menu.



Visual Studio Code with the GitHub Copilot menu expanded, showing multiple interaction and configuration options for using the AI assistant

Open Chat (Ctrl+Alt+I)

Use case: You need help understanding a block of unfamiliar code or generating a function on the fly.

Sample prompts:

```
"What does this regex do?"
"Generate a function to validate email format."
```

Benefit: Think of this as your coding assistant; it is conversational, context-aware, and fast.

Editor Inline Chat (Ctrl + I)

Use case: You're editing a class and want to ask for a method without opening the sidebar.

Example: Highlight a method and then type the following prompt:

```
"Convert this function to async and add error handling."
```

Benefit: Perfect for in-place refactoring or micro-improvements.

Quick Chat (Ctrl+Shift+Alt+L)

Use case: You're writing a loop and are stuck. Use Quick Chat for a one-line prompt without switching focus.

Sample prompt:

```
"Generate a for-loop to iterate over a list of files and prin
```

Configure Code Completions

Use case: You're working on a sensitive codebase and want to limit suggestions to only certain file types.

> **Tip**
>
> You can toggle completions for file types like .ipynb, .json, and .sql.

Manage Copilot

Use case: You need to access your Copilot account, disable telemetry, switch to Copilot for Business, or set up organization-wide preferences. Or you need to customize how Copilot handles suggestions, integrates with other extensions, or operates offline.

Tip

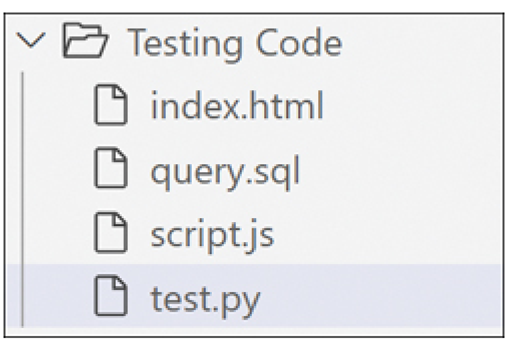
Combine chat, inline chat, and indexing for best results. Chat gives high-level contextual help, inline chat offers precision edits, and indexing ensures that suggestions are project aware.

Test Copilot with sample prompts across languages

This section guides you through the process of verifying GitHub Copilot's installation by writing your first prompt and observing suggestions in multiple programming languages. It also covers how to use keyboard shortcuts to control suggestions.

Step 1: Creating a file for each language

To enable Copilot to understand the language context, you must first create a file with the correct extension in Visual Studio Code.



Folder structure in Visual Studio Code, displaying HTML, SQL, JavaScript, and Python files under the Testing Code directory

To verify Copilot functionality across programming languages, you can start by creating test files. The following table shows the file extension and steps to create them in Visual Studio Code.

Language	File extension	Steps to create a file
Python	.py	File > New File > Save As > test.py
JavaScript	.js	File > New File > Save As > script.js
SQL	.sql	File > New File > Save As > query.sql
HTML	.html	File > New File > Save As > index.html

Tip

The file extension tells Copilot which programming language you're working in, enabling it to generate accurate code.

Step 2: Writing a sample function with a comment

Once your file is created and open, you can write a natural language comment that describes the functionality you want. Copilot will respond with code suggestions in real time. Let's start with the Python file test.py.

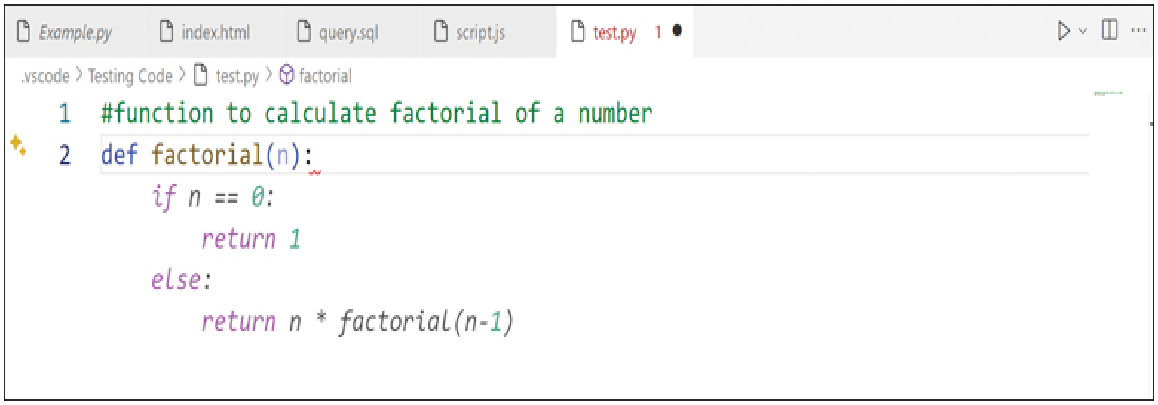
Example:

```
#function to calculate factorial of a number
```

Copilot suggestion:

```
def factorial(n):
    if n < 0:
```

```
        return "Invalid input"
    elif n == 0 or n == 1:
        return 1
    else:
        result = 1
        for i in range(2, n + 1):
            result *= i
        return result
```



Python code generated by GitHub Copilot, based on a comment to calculate the factorial of a number, demonstrating AI-assisted coding with recursion

Step 3: Testing suggestions in multiple languages

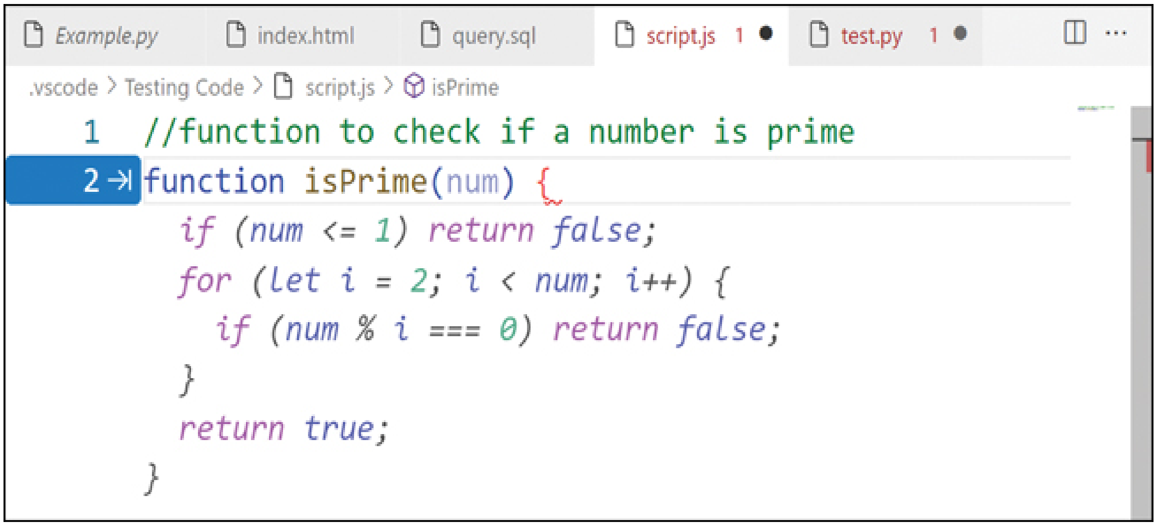
You can try the same idea with other files to verify that Copilot works across languages.

JavaScript (script.js)

Prompt:

```
//function to check if a number is prime
```

Suggested output:



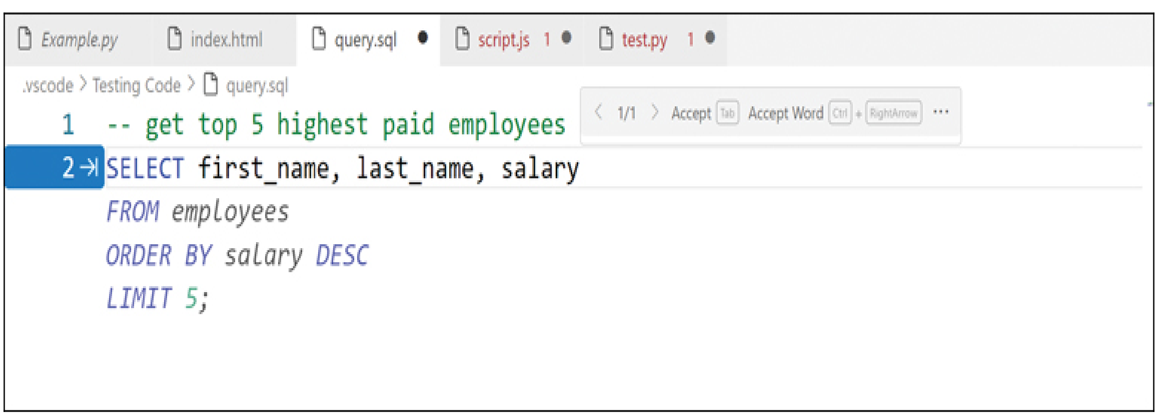
JavaScript function generated by GitHub Copilot from a comment to determine if a number is prime, highlighting AI-assisted code generation with syntax coloring

SQL (query.sql)

Prompt:

```
-- get top 5 highest paid employees
```

Expected output:



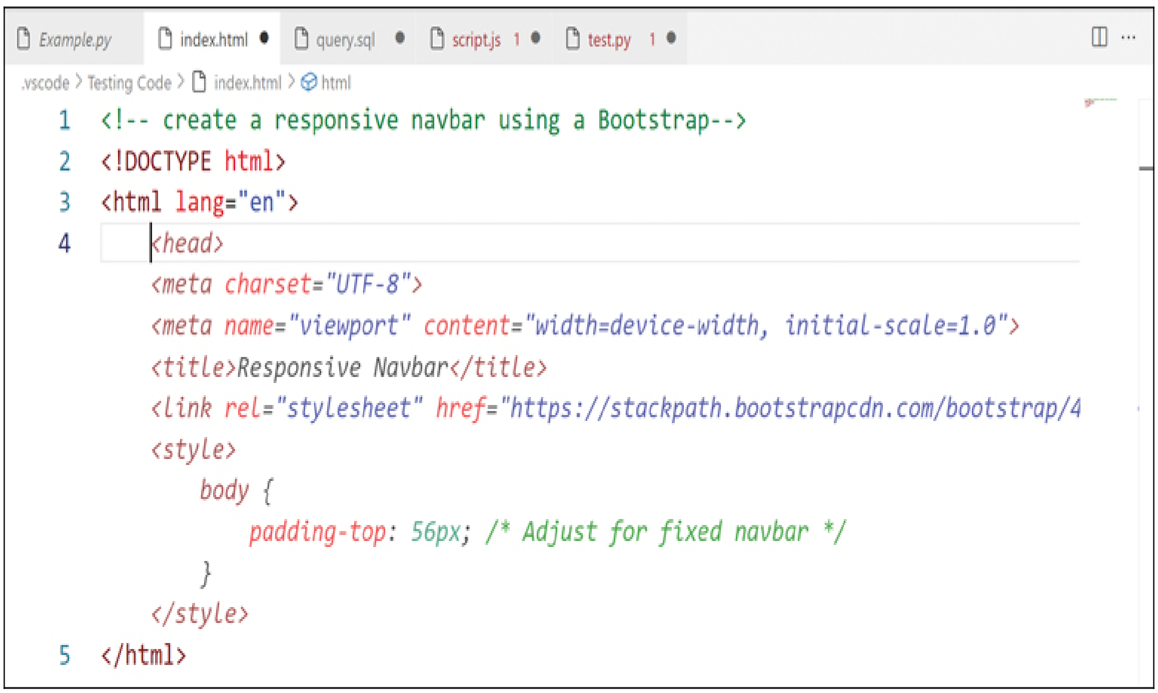
SQL query generated by GitHub Copilot to select the top five highest-paid employees from an employee table, with Copilot's in-line *suggestion options visible*

HTML (index.html)

Prompt:

```
<!-- create a responsive navbar using a Bootstrap -->
```

Suggested output:



HTML boilerplate generated by GitHub Copilot based on a comment to create a responsive navbar using a bootstrap, illustrating AI assistance in web development

Troubleshoot installation issues and common errors

Even with a streamlined setup process, you may occasionally encounter issues when installing or activating GitHub Copilot. This section outlines common problems and provides solutions to help you get up and running quickly.

Copilot extension not appearing in the IDE

Problem:

You've installed GitHub Copilot, but the extension doesn't show up in Visual Studio Code or another IDE.

Solutions:

- Ensure that you are using a supported version of the IDE.
- Restart the IDE and check the Extensions panel again.
- Manually trigger Copilot by pressing Ctrl+Shift+P and then search for "Copilot."

Authentication or sign-in issues

Problem:

You're prompted to sign in repeatedly, or GitHub Copilot refuses to authenticate.

Solutions:

- Make sure you are signed in with a GitHub account that has an active Copilot subscription.
- Open the Command Palette (Ctrl+Shift+P) and run GitHub: Sign Out. Then run GitHub: Sign In again.
- Check your network connection and firewall settings (especially in enterprise environments).

Copilot suggestions not appearing

Problem:

You’re writing code, but Copilot does not provide any suggestions.

Solutions:

- Ensure that you are in a supported file type (.py, .js, .html, etc.).
- Press Alt+/ (or Ctrl+Enter) to manually trigger suggestions.
- Check to see if the Copilot icon in the status bar shows Disabled. If it does, re-enable Copilot through the status bar menu.
- Check to see if the Copilot icon in the status bar shows Offline. If it does, reconnect to the internet.

Extension installation fails or is blocked

Problem:

The GitHub Copilot extension fails to install or update.

Solutions:

- Run VS Code as Administrator (Windows) or with elevated permissions (macOS/Linux).
- Try uninstalling and reinstalling the extension.
- Check if your network proxy or antivirus is blocking extension downloads.

Copilot Chat or Labs features not available

Problem:

You can use Copilot suggestions, but advanced features like Chat or Labs don’t appear.

Solutions:

- Verify that you are on the Pro or Pro+ plan as some features require higher-tier subscriptions.
- Make sure the GitHub Copilot Chat extension (which is separate from the core Copilot extension) is installed.
- Check experimental feature toggles in your IDE settings.

Slow or laggy suggestions

Problem:

Suggestions take a long time to load, or typing feels delayed.

Solutions:

- Disable other heavy extensions to improve IDE performance.
- Switch to a lightweight file or reduce the number of open tabs.
- Ensure that you have a stable internet connection, as Copilot fetches suggestions from the cloud.

> **Tip**
>
> Use the GitHub Copilot status icon in the bottom-right corner of VS Code to get real-time updates on Copilot's connectivity and activity.

> **See Also**
>
> For additional help, refer to the official GitHub Copilot troubleshooting guide: https://docs.github.com/en/copilot/how-tos/troubleshoot/troubleshooting-common-issues-with-github-copilot.

Apply best practices for using Copilot effectively

As you begin your journey with GitHub Copilot, understanding how to work strategically and thoughtfully with the tool is key to maximizing its value. Copilot is not just a code-completion assistant; it’s a generative AI model that interprets your comments, code structure, and intent to suggest possible implementations. While it can save time and reduce repetitive coding tasks, it's not perfect or infallible.

The initial days of using Copilot are critical for developing productive habits and setting the right expectations. It's important to learn how to interpret the suggestions Copilot provides, how to effectively prompt it using comments, and when to trust or modify its output. Without proper guidance, developers may either over-rely on suggestions or fail to utilize the tool to its full potential.

This section offers a practical set of best practices tailored for first-time users, ensuring that your early experiences with GitHub Copilot are effective, secure, and aligned with your development goals. Whether you're a student, a professional, or an open-source contributor, these practices will help you build confidence and mastery in using AI-assisted development tools.

When to accept, modify, or reject suggestions

Copilot generates code based on context, which means its suggestions may not always be accurate, secure, or optimal. Knowing when to accept, modify, or reject suggestions is a vital part of the learning curve.

Accept suggestions when:

- The code output is syntactically and logically corrected for your needs.
- The suggestion matches your intended structure or logic.
- You’ve reviewed the suggestion and verified that it meets any security or performance standards.

Modify suggestions when:

- The structure is correct, but the logic needs refinement.
- Variable names, formatting, or comments need to follow your project's conventions.
- The code is almost correct but requires context-specific adjustments.

Reject suggestions when:

- The code is irrelevant, incorrect, or inefficient.
- The code does not align with the best practices (e.g., security, architecture).
- Copilot introduces patterns you do not want junior developers or your team to follow.

> **Important**
>
> Always review suggestions critically, especially in production-grade or security-sensitive applications.

Using comments effectively to prompt Copilot

Copilot heavily relies on natural language comments to understand what you intend to write. Writing clear, descriptive comments before you start to code can significantly improve the relevance of suggestions.

Consider the following tips for writing effective prompts.

Be specific

A specific prompt like this:

```
# Calculate the factorial of a number using recursion
```

yields better results than this:

```
# Factorial
```

Include structure

If you know the structure you want, include it. For example:

```
# Create a Python function that accepts a list of integers an
```

Break down logic

Writing step-by-step logic in comments helps Copilot understand complex tasks. Here's an example:

```
# Step 1: Check if the list is not empty
# Step 2: Filter elements greater than 10
# Step 3: Return the filtered list
```

Use docstrings

Starting with a function name and a docstring helps Copilot infer functionality.

See Also

Chapter 3: Writing effective prompts for AI-powered coding dives deeper into writing clear and effective prompts to get the most out of Copilot's AI suggestions.

Setting realistic expectations as you onboard

Copilot is not a silver bullet. It's a coding assistant, not a replacement for human judgment or experience.

Keep in mind these key points:

- Copilot doesn't understand your business logic. Copilot can't fully grasp your project's domain context or long-term architecture. Always cross-check logic and performance.
- Quality depends on context. The more context you provide—like descriptive comments, file structure, or function names—the better the suggestions.
- Trial and error is part of the process. Don't expect perfect results immediately. Use Copilot iteratively: Try, observe, and modify.
- Monitor code quality and style. Copilot may use patterns or syntax styles that differ from your team's guidelines. Use linters, formatters, and manual reviews to maintain consistency.

The importance of security and compliance checks

Never assume that Copilot-generated code is safe for production. Always run security scans and perform audits as needed.

The following table summarizes common best practices and common pitfalls when using GitHub Copilot effectively.

Do	**Don't**
Write clear and specific comments	Use vague prompts like // do something
Review suggestions before accepting them	Blindly copy and paste Copilot code

Do	Don't
Customize suggestions to fit project standards	Assume that Copilot understands business logic
Use Copilot to speed up the creation of boilerplate code	Use Copilot to replace deliberate design decisions
Start with simpler use cases	Jump straight into complex systems with Copilot

Explore Copilot's functionality in online and offline modes

GitHub Copilot is a cloud-powered AI tool that depends heavily on access to online resources for its real-time suggestions. However, developers today work in a mix of environments—some entirely cloud based (like GitHub Codespaces) and others locally installed and sometimes even offline. Understanding how Copilot behaves in both scenarios is essential to plan your workflow, avoid surprises, and maximize efficiency.

GitHub Codespaces integration

GitHub Codespaces is a cloud-hosted development environment that runs entirely in the browser or from Visual Studio Code. It allows you to spin up container-based dev environments with all your tools, dependencies, and extensions—on demand. Copilot integrates natively and seamlessly into GitHub Codespaces, making it ideal for developers who want to:

- Start coding instantly without local setup.
- Work from anywhere with consistent environments.
- Get real-time Copilot suggestions with minimal latency.

Use case example

A student working on a group project can open Codespaces directly from the GitHub repo, write a comment like:

```
// function to calculate prime numbers in JavaScript
```

and get an instant, intelligent suggestion from Copilot without any local setup. All suggestions are served from GitHub's cloud infrastructure.

> **Tip**
>
> GitHub Codespaces ensures that Copilot remains active as long as internet access is available, since the environment is inherently online.

How Copilot works without an internet connection (limitations)

GitHub Copilot requires an active internet connection for generating suggestions. This is because Copilot doesn't run its model locally; it sends your prompt (e.g., comments or code context) to the GitHub Copilot servers, which return relevant completions. However, it is available in offline mode, which can be useful if you temporarily lose connectivity (such as on a plane or in a restricted network environment) and still need to keep working with your existing code.

These are the key limitations in offline mode:

- No suggestions: If your machine goes offline, Copilot cannot provide new completions or suggestions. The Copilot icon typically shows a warning or "offline" message in such cases.
- No updates or learning: Copilot cannot fetch new context or respond to changes in real time.

- Cached data is limited: Even if you had previous suggestions, they are not stored locally in a meaningful way for reuse.
- Feature access is blocked: Features like Labs, feedback submission, and telemetry are inaccessible offline.

Use case caution

If you're coding on a plane or in a remote area with no internet, Copilot won't function. Developers in such scenarios should prepare code templates, snippets, or reference materials in advance, or they can temporarily rely on traditional IDE features like IntelliSense.

GitHub Copilot mode comparison

The following table compares GitHub Copilot's availability and performance across different development environments, both online and offline.

Mode	Copilot availability	Performance	Ideal for
GitHub Codespaces (online)	Full support	Fast, cloud based	Quick setup, remote collaboration, cloud-first teams
Local IDE (online)	Full support	Slight latency	Personal projects, enterprise workstations
Offline (local)	Not available	No suggestions	Not recommended for Copilot usage

Important

If you're concerned about data privacy when working online, you can configure Copilot’s telemetry and data-sharing settings under your IDE's GitHub Copilot preferences.

See Also

If you’re curious about how Copilot compares across real-world projects and environments, check out Chapter 8, which offers in-depth use cases and developer workflows.

Skills review

In this chapter, you learned how to:

- Set up a GitHub account and choose the appropriate Copilot subscription plan.
- Understand the differences between the Free, Pro, and Pro+ tiers of GitHub Copilot.
- Install GitHub Copilot in supported IDEs, including Visual Studio Code, JetBrains IDEs, and GitHub Codespaces.
- Configure Copilot settings to match your development style and file preferences.
- Use keyboard shortcuts and interface features to control Copilot’s suggestions and behavior.
- Test your setup by writing prompts in different languages, like Python, JavaScript, SQL, and HTML.
- Troubleshoot common installation and activation issues with GitHub Copilot.
- Apply best practices for prompting Copilot effectively and securely.
- Understand how Copilot functions differently in online and offline environments, including GitHub Codespaces integration.

Practice tasks

No practice files are necessary to complete the practice tasks in this chapter. Follow the steps below to reinforce the concepts and tools introduced in each section.

Set up your GitHub account and select a Copilot plan

Create your GitHub account and explore available Copilot plans:

1. Open your browser and go to https://github.com.
2. Select Sign Up and enter your email, password, and username.
3. Complete the CAPTCHA and email verification steps.
4. Navigate to https://github.com/features/copilot.
5. Compare the Free, Pro, and Pro+ subscription options.
6. Select Start Free Trial if applicable or subscribe to the plan that suits your needs.

Install Copilot in your development environment

Set up GitHub Copilot in your preferred IDE:

1. Open https://code.visualstudio.com and install Visual Studio Code.
2. Launch VS Code and open the Extensions panel (Ctrl+Shift+X).
3. Search for GitHub Copilot and then click Install.
4. Sign in with your GitHub account when prompted.

5. Verify that the Copilot icon appears in the bottom-right corner of the IDE.
6. Repeat steps 2–5 in another supported IDE, like JetBrains or Codespaces, if desired.

Configure permissions and personal settings

Customize Copilot behavior in VS Code:

1. In VS Code, go to File > Preferences > Settings.
2. Search for Copilot to view all available settings.
3. Adjust the suggestion frequency, panel view, and language preferences.
4. Open Keyboard Shortcuts and test:

   a. Alt+/ to trigger a suggestion.

   b. Tab to accept a suggestion.

   c. Ctrl+Enter for multiple options.

5. Explore the Copilot sidebar and Labs features (if available).

Test Copilot with sample prompts across languages

Try prompts in Python, JavaScript, SQL, and HTML:

2. Create the following new files in VS Code:

   a. test.py

   b. script.js

c. query.sql

d. index.html

3. In each file, write a comment that describes a function or layout:

a. In test.py, try this prompt:

```
# function to calculate factorial
```

b. In script.js, try this prompt:

```
// check if number is prime
```

c. In query.sql, try this prompt:

```
-- select top 5 salaries
```

d. In index.html, add this prompt:

```
<!-- responsive navbar using Bootstrap -->
```

4. Observe Copilot’s suggestions and accept or modify them.

Troubleshoot installation issues and common errors

Simulate and resolve common Copilot setup issues:

1. Disconnect from the internet and confirm that Copilot goes offline.
2. Reconnect and restart VS Code.

3. Open the Command Palette (Ctrl+Shift+P) and run GitHub: Sign Out and then GitHub: Sign In.
4. Try triggering suggestions in an unsupported file type (e.g., a .txt file) to observe limitations.
5. Visit your GitHub Copilot dashboard to confirm your subscription status.

Apply best practices for using Copilot effectively

Learn to prompt Copilot for optimal code suggestions:

1. In a Python file, write vague and specific comments to compare results:

   ```
   # function
   ```

   vs.

   ```
   # function to reverse a list
   ```

2. Try adding a step-by-step comment block:

   ```
   # Step 1: Filter even numbers
   # Step 2: Square them
   ```

3. Accept and test suggestions, modifying variable names and logic.
4. Create a docstring-based prompt and observe Copilot's understanding.
5. Write insecure code and observe whether Copilot flags or reinforces bad practices.

Explore Copilot’s functionality in online and offline modes

Compare Copilot behavior in cloud vs. local setups:

1. Open GitHub Codespaces and confirm that Copilot provides suggestions immediately as you start typing code.
2. Use inline comments in Codespaces to test real-time completions.
3. Switch to local VS Code and simulate working offline.
4. Try using Copilot Chat (if on the Pro plan) and note what the chat experience is like.
5. Review telemetry settings in your GitHub Copilot preferences.
6. Reflect on how cloud-first environments differ from local development.

3

Writing effective prompts for AI-powered coding

In this chapter

- Understand what makes a prompt for AI code generation effective
- Use comment-based, function signature, and docstring prompts
- Refine or rewrite prompts to improve suggestion quality
- Apply prompt templates for algorithms, tests, refactoring, and more
- Customize prompts for specific languages, libraries, and frameworks
- Identify and fix vague and inefficient prompts
- Build an iterative workflow to continuously improve prompt results

Practice files

There are no practice files for this chapter.

GitHub Copilot is a powerful coding assistant, but its usefulness depends largely on how well you communicate with it. This chapter focuses on the art and science of writing effective prompts to guide Copilot in generating accurate, efficient, and relevant code. Whether you're working on a new feature, debugging, or simply looking for a quick script, the right prompt can make all the difference. You'll learn how to structure prompts using comments, function headers, and docstrings, as well as how to refine suggestions when they don't initially meet your needs. This chapter also provides reusable prompt templates for common coding scenarios and highlights ways to avoid common pitfalls that can confuse the AI. Through practical tips and real-world examples, you'll gain the skills you need to collaborate with Copilot as a productive coding partner.

Understand what makes a prompt for AI code generation effective

GitHub Copilot is like a helpful coding partner—but it can't read your mind. The more clearly you describe what you want, the more useful the code it provides will be. In this section, we'll focus on how to write clear and specific prompts in Python and show you with examples how small changes can make a big difference.

Think of it like giving instructions to a junior developer. If your instructions are vague, you'll get unpredictable results. But when you're clear and detailed, the outcome is exactly what you expect.

What makes a prompt effective?

An effective prompt is:

- Clear: It tells Copilot what you want in simple, understandable language.
- Specific: It describes the task, expected behavior, inputs/outputs, and, optionally, the language or libraries involved.
- Contextual: It aligns with the surrounding code or your current development setup, giving Copilot useful clues.

Compare vague vs. effective prompts

When you give Copilot a vague prompt, you're essentially tossing it a half-finished thought and hoping it can read your mind. Something like "handle data" or "do the thing" might make sense to you in the moment, but the AI sees a puzzle with most of the pieces missing.

Vague prompt:

```
# calculate tax
```

This prompt is simply too open ended. Copilot might guess that you want to apply some kind of tax, but it won't know:

- What's the tax rate?
- What are the input and output?
- Which country's tax rule is in play?
- Is it for a product or income?

Improved prompt:

```
# Calculate a 10% sales tax for a given product price and ret
```

```
def calculate_total_price(price):
```

This tells Copilot:

- The tax rate (10%)
- The domain (sales tax)
- The input (product price)
- What the function should return (final price, including tax)

Copilot can now complete the code so it looks like this:

```python
def calculate_total_price(price):
    tax = price * 0.10
    total = price + tax
    return total
```

This code result is short, readable, and accurate because the prompt gave it just the right amount of context.

Prompt writing tips for Python

Here are a few simple strategies to make your prompts more effective.

1. State the goal clearly

Tell Copilot exactly what the code should do.

Vague prompt:

```python
# process data
```

Improved prompt:

```python
# Process a list of student scores and return the average
```

2. Mention input and output

By mentioning the input and output, you can help Copilot understand how to structure the function.

Effective prompt:

```python
# Given a list of numbers, return to a new list with only eve
```

3. Specify language or tools, if needed

If your task requires a specific language, library, or tool, include that detail so Copilot tailors the output accordingly.

Effective prompt:

```
# Using Python, create a dictionary from two lists: keys and
```

4. Use natural, complete sentences in comments

You don’t need fancy terms in comments. Just write them as if you're explaining to a complete beginner, using full sentences.

Pro tip

When Copilot’s suggestions are off, ask yourself:

- Did I give it enough detail?
- Does it know what I want as output?
- Should I break this into smaller steps?

Think of prompts as instructions in plain English that describe what your Python function should do.

Tip

Write prompts as if you're giving instructions to a junior developer. Clear, specific, goal-oriented language works best.

Important

Avoid vague words like "process," "analyze," or "clean" unless followed by a specific action or context.

Sample prompt

How would you write an effective prompt to count how many times each word appears in a paragraph?

```
."
```

Prompt:

```
# Given a paragraph of text, count the frequency of each word
```

```
def count_word_frequency(text):
```

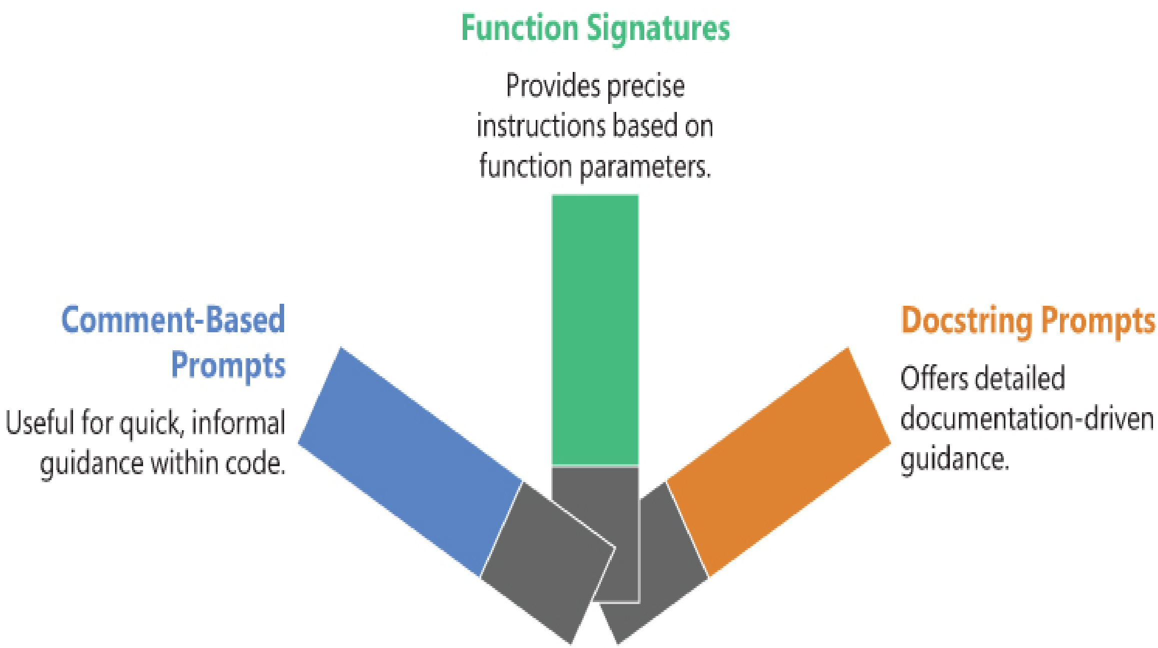
Copilot can now generate useful Python code that works to solve that problem.

See Also

Chapter 4: Enhancing code efficiency with AI assistance, for techniques on automating and optimizing prompt-driven coding tasks.

Use comment-based, function signature, and docstring prompts

GitHub Copilot understands a variety of prompt styles, especially in Python. In this section, we'll explore three main types of prompts you can use to guide Copilot: comment-based prompts, function signatures, and docstring prompts. Each one helps Copilot understand what you want in a slightly different way. When used well, they can greatly improve the accuracy and usefulness of Copilot's suggestions.



GitHub Copilot prompt types: comment-based prompts for informal guidance, function signatures for precise parameter instructions, and docstring prompts for documentation-driven help

Comment-based prompts

Comment-based prompts are plain-language comments written just above your code. You can use these prompts to describe (in a single line or short paragraph) what you want the code to do.

Effective prompt:

```
# Convert a temperature from Celsius to Fahrenheit
```

```
def convert_temperature(celsius):
```

Copilot-suggested code:

```
def convert_temperature(celsius):
    return (celsius * 9/5) + 32
```

Tip

Use comments when your task can be described in a sentence. This is quick and works well for simple logic.

Function signature prompts

When you write a prompt using just the function name and parameters, Copilot looks at the names and guesses the purpose. This method is faster than using comment-based prompts, but it works best when your function and variable names are meaningful.

Here is an example of a prompt you could write in the Visual Studio code IDE or any preferred IDE where GitHub Copilot is configured.

Effective prompt:

```
def calculate_area_of_circle(radius):
```

Copilot-suggested code:

```
import math
def calculate_area_of_circle(radius):
    return math.pi * radius * radius
```

Important

Function names must be descriptive. Copilot may misinterpret vague or generic function names, such as process_data().

Docstring prompts

A docstring is a multi-line comment inside a function. To write a docstring, you use triple quotes (““““) and explain the purpose, inputs, and outputs in more detail.

Effective prompt:

```
def get_even_numbers(numbers):
  """
  Given a list of integers, return a new list containing only
  """
```

Copilot-suggested code:

```
def get_even_numbers(numbers):
    """
    Given a list of integers, return a new list containing only t

even numbers.
    """
    return [num for num in numbers if num % 2 == 0]
```

> **Tip**
>
> Use docstrings when you want to explain the logic more clearly, especially for complex functions or when collaborating with others.

Prompt style comparison

The following table compares different prompt styles to help guide Copilot’s suggestions effectively.

Prompt style	Best for	Example
Comment-based prompts	Quick, readable task descriptions	# Calculate discount for a shopping cart
Function signature prompts	Simple logic with clear naming	def sort_students_by_grade(students):
Docstring-based prompts	More detail, ideal for documentation	Describing inputs, outputs, and behavior

See Also

Chapter 5, Debugging and troubleshooting code with Copilot, for more on using prompt styles to generate and refine bug fixes.

Refine or rewrite prompts to improve suggestion quality

Even a small change in how you write a prompt can lead to better code suggestions from GitHub Copilot. In this section, you'll learn how to recognize vague and weak prompts and how to refine or rewrite them to get more useful, accurate, and relevant code completions, especially in Python.

Why refining prompts matters

Copilot is trained to look for clues. If your prompt is too short, too broad, or unclear, Copilot will struggle to understand your goal. But if you give it just a bit more guidance—such as expected input types or desired behavior—Copilot is likely to produce exactly what you need.

> **Tip**
>
> Break complex prompts into smaller subtasks. Copilot performs best when each prompt focuses on one clear goal.

Case study: Writing a sorting function

Let’s look at refining prompts with a case study–style example.

Initial (weak) prompt:

```
# sort a list
```

```
def sort_list(lst):
```

Copilot-suggested code:

```
def sort_list(lst):
    return sorted(lst)
```

This code works, but what if you wanted descending order or case-insensitive string sorting? The initial prompt is too vague. Copilot needs to know more, such as what kind of sorting (ascending or descending) you want and whether there are any custom rules to consider.

You might want to revise the initial prompt to sort strings alphabetically, using a case-insensitive sort.

Refined prompt 1:

```
# Sort a list of strings in alphabetical order, ignoring case
```

```
def sort_list_case_insensitive(lst):
```

Copilot-suggested code:

```
def sort_list_case_insensitive(lst):
    return sorted(lst, key=lambda x: x.lower())
```

Important

Even one missing detail (like input format or output type) can lead to incomplete or incorrect suggestions.

Now let's say you want to sort a list of dictionaries by key.

Refined prompt 2:

```
# Sort a list of student dictionaries by the 'grade' key in d
```

```
def sort_students_by_grade(students):
```

Copilot-suggested code:

```
def sort_students_by_grade(students):
    return sorted(students, key=lambda x: x['grade'], reverse=Tru
```

Techniques to refine prompts

Here are some ways to improve a prompt if Copilot's first response isn't what you expected.

Add context

Copilot performs best when you provide extra details or background, giving it the context needed to generate accurate results.

Vague prompt:

```
# find primes
```

Improved prompt:

```
# Find all prime numbers up to a given number n using the Sie
```

Break complex tasks into smaller parts

Large or abstract tasks can confuse Copilot. To help Copilot give you the results you're looking for, give it steps.

Vague prompt:

```
# create a chatbot
```

Improved prompt:

```
# Step 1: Tokenize user input
# Step 2: Match keywords
# Step 3: Respond with predefined answers
```

Explain the expected input/output

Copilot performs best when it knows what to expect.

Effective prompt:

```
# Given a list of integers, return a list of their squares
```

```
def square_numbers(nums):
```

Weak vs. refined prompts

Refining a prompt is about adding just enough clarity and context to guide Copilot. The following table provides some examples.

Weak prompt	Refined prompt
# process data	# Clean a list of strings by removing whitespace and converting to lowercase
# analyze text	# Count the number of words in a string, ignoring punctuation
def func(x):	def is_palindrome(text):

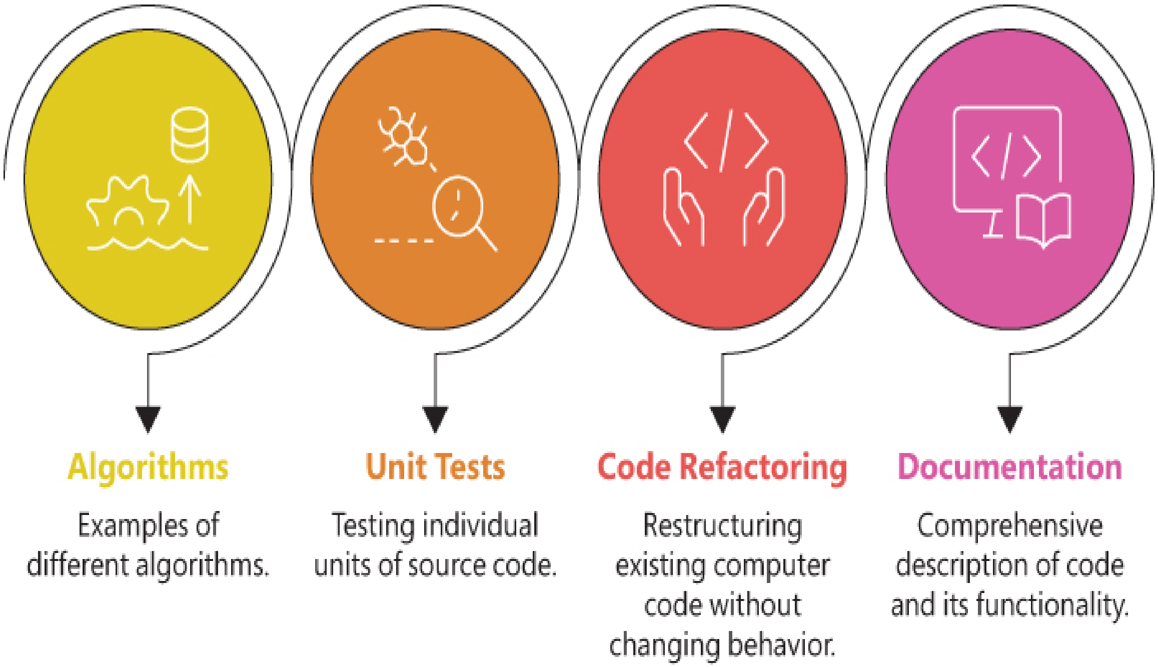
Apply prompt templates for algorithms, tests, refactoring, and more

Once you understand how to write effective prompts, the next step is to reuse structured prompt templates for different coding tasks. These templates act like formulas: You fill in the blanks, and Copilot generates what you need.

In this section, we'll cover four practical types of templates using Python examples:

- Algorithms
- Unit tests
- Code refactoring

- Documentation



Software engineering domains where GitHub Copilot can assist

Algorithm prompt template

When you need to implement logic or solve a common problem, using a structured description helps Copilot generate correct and efficient code.

Template:

```
# Problem: [Explain the algorithm goal]
# Input: [Describe the input format]
# Output: [Describe the expected output]
def function_name(parameters):
```

Example: Binary search

Prompt:

```
# Problem: Perform binary search on a sorted list to find the
# Input: A sorted list of integers and a target integer
# Output: The index of the target if found, otherwise -1
def binary_search(arr, target):
```

Copilot-suggested code:

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Unit test prompt template

You can prompt Copilot to write tests using a format that makes the intent very clear.

Template:

```
# Write unit tests using unittest for the function [function_
# The function takes [inputs] and returns [expected outputs]
```

Copilot-suggested code:

```
import unittest
class TestFunction(unittest.TestCase):
```

Example: Sum of two numbers

Prompt:

```
# Write unit tests using unittest for the function add_number
# The function returns the sum of two numbers
```

Copilot-suggested code:

```
import unittest
class TestAddNumbers(unittest.TestCase):
Copilot will produce:
def add_numbers(a, b):
    return a + b

class TestAddNumbers(unittest.TestCase):
    def test_positive_numbers(self):
        self.assertEqual(add_numbers(3, 4), 7)

    def test_negative_numbers(self):
        self.assertEqual(add_numbers(-2, -5), -7)
```

Important

Always review AI-generated code from templates for correctness. Copilot may make logical or boundary-related errors.

Refactoring prompt template

If you want Copilot to improve or simplify code, prompt it with the phrase “Refactor this code to…” and specify the goal.

Template:

```
# Refactor this code to [improve performance / readability /
```

Example: Improved readability

Prompt:

```
# Refactor this code to improve readability
```

```
def multiply_list(numbers):
    result = 1
    for i in range(len(numbers)):
        result *= numbers[i]
    return result
```

Copilot-suggested code:

```
def multiply_list(numbers):
    result = 1
    for num in numbers:
        result *= num
    return result
```

Documentation prompt template

You can ask Copilot to generate documentation for existing functions.

Template:

```
# Add docstring to this function
```

```
def function_name(parameters):
    [function body]
```

Example: Added docstring

Prompt:

```
# Add docstring to this function
```

```
def is_even(n):
    return n % 2 == 0
```

Copilot-suggested code:

```
def is_even(n):
    """
    Check if a number is even.

    Parameters:
    n (int): The number to check.

    Returns:
    bool: True if the number is even, False otherwise.
    """
    return n % 2 == 0
```

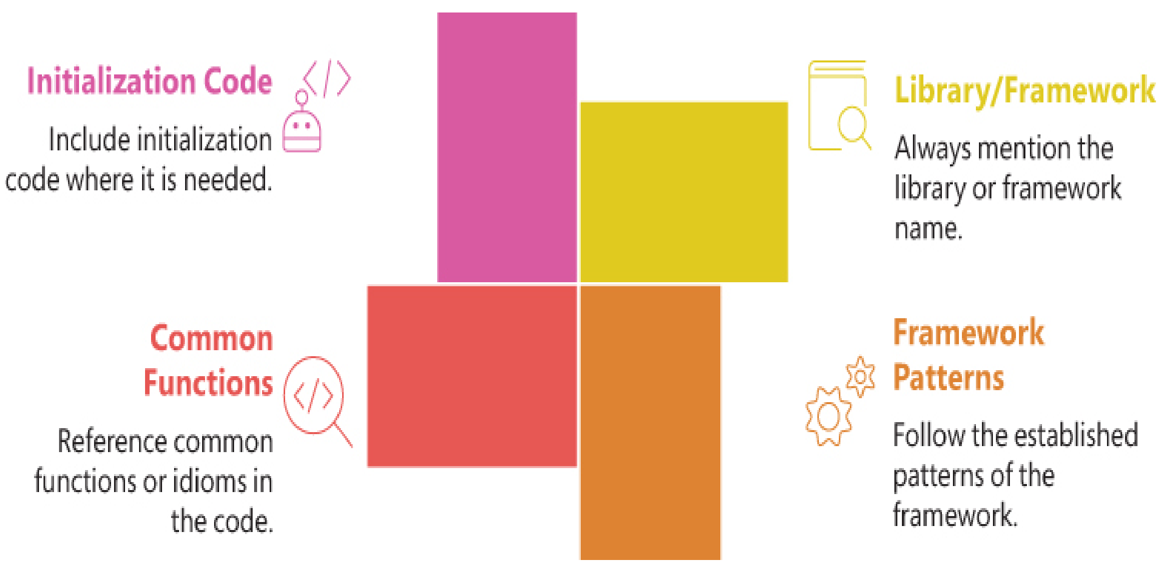
Prompt templates at a glance

Using prompt templates saves time and helps you guide Copilot more effectively—especially when repeating tasks across a large project. The following table summarizes the reusable prompt templates for common programming tasks.

Task	Prompt pattern
Algorithm	Problem/input/output comment block
Unit test	# Write unit tests for function X
Refactoring	# Refactor this code to...
Documentation	# Add docstring to this function

Customize prompts for specific languages, libraries, and frameworks

When you’re working with GitHub Copilot, general prompts often work well. But when you're using specific tools—like Python libraries or frameworks—it’s important to mention them explicitly in your prompts. This specificity helps Copilot tailor its suggestions to fit the conventions, functions, and syntax of those tools.



Key elements developers should include in code for clarity and consistency

Let’s explore how you can adapt your prompts for better results with Python libraries and frameworks.

Mention the library or framework name

By indicating exactly what library or framework you're using, you can help Copilot improve its accuracy. Copilot adjusts its completions to match the APIs, style, and patterns of that tool.

Example: Using NumPy

Weak prompt:

```
# Create a 2D array of zeros with shape (3, 3)
```

Copilot-suggested code:

```
array = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Effective prompt:

```
# Use NumPy to create a 2D array of zeros with shape (3, 3)
```

```
import numpy as np
Copilot will generate:
array = np.zeros((3, 3))
```

Follow the patterns of the framework

Frameworks have their own coding structures. If you're using Flask for a web app or Pandas for data manipulation, showing Copilot an example aligned with that structure helps it continue correctly.

Example: Flask route

Effective prompt:

```
# Create a Flask route that returns "Hello, World!" at the ro
from flask import Flask
app = Flask(__name__)
```

Copilot-suggested code:

```
@app.route('/')
def home():
    return "Hello, World!"
```

Reference common functions or idioms

Using prompts that follow the way people usually work with a library helps Copilot identify what you're expecting.

Example: Pandas data cleaning

Effective prompt:

```
# Using pandas, drop rows with missing values from the DataFr
import pandas as pd
```

Copilot-suggested code:

```
df = df.dropna()
```

Include initialization code where needed

Some libraries require initialization. Copilot performs best if you include setup steps in the prompt.

Example: Matplotlib plotting

```
# Plot a line graph of x and y using matplotlib
```

```
import matplotlib.pyplot as plt
x = [1, 2, 3]
y = [4, 5, 6]
```

Copilot-suggested code:

```
plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Graph')
plt.show()
```

Tip

Mention the specific library (e.g., NumPy, Pandas) in your prompt to guide Copilot toward relevant, optimized code.

Important

If your project uses a specific version or API, verify that Copilot's suggestions match the correct syntax and behavior.

Templates for specific libraries and frameworks at a glance

Use the following templates to tailor prompts for specific libraries and frameworks.

Goal	Template
NumPy arrays	# Use NumPy to [do something specific]
Pandas cleaning	# Using pandas, [describe task]
Flask route	# Create a Flask route to [describe endpoint behavior]
Matplotlib graph	# Plot a [type] chart using matplotlib

Identify and fix vague and inefficient prompts

As you've seen, when you're using GitHub Copilot, not every prompt gets you the result you expect. The problem is often that the prompt is too vague or ambiguous, or it lacks context. In this section, you'll learn how to recognize ineffective prompts and revise them to get better, more precise code suggestions.

Why vague prompts fail

If your prompt is too general or unclear, Copilot doesn't have enough information to generate useful code. It might make incorrect assumptions, use a different library than intended, or produce boilerplate code that doesn't fit your use case.

Ineffective prompt:

```
# Create a function
```

This could mean anything—Copilot has no idea what the function is supposed to do.

Improved prompt:

```
# Create a function to check if a number is prime
```

Now Copilot has a clear task and will likely generate a relevant function.

Recognizing inefficiency in prompts

Sometimes a prompt is technically clear, but the code Copilot generates is overly verbose, slow, or doesn't follow best practices. This happens when the prompt lacks specificity in performance goals, data size, or language features.

Inefficient prompt:

```
# Using a loop, count how many times each element appears in
```

This will likely lead to a basic loop-based implementation.

Copilot-suggested code:

```
def count_frequencies(lst):
    freq = {}
    for item in lst:
        if item in freq:
            freq[item] += 1
        else:
            freq[item] = 1
    return freq
```

Improved prompt:

```
# Count frequency of elements in a list using collections.Cou
```

Copilot-suggested code:

```
from collections import Counter
frequencies = Counter(my_list)
```

This version is more concise and readable, and it takes advantage of standard library optimizations.

You're now guiding Copilot to use an optimized Pythonic solution.

> **Important**
>
> Generic prompts may lead Copilot to suggest outdated methods or inefficient loops. It is important to refine your language to specify libraries or idioms.

Fixing prompts: Before and after

The following table presents examples of vague prompts, explains why they are ineffective, and offers improved versions that provide clearer guidance to GitHub Copilot.

Vague prompt	Problem	Improved prompt
# Sort a list	No sort order specified	# Sort a list of integers in descending order

Vague prompt	Problem	Improved prompt
# Read a file	File type/goal not clear	# Read a CSV file using pandas
# Clean data	Too general	# Remove null values from a DataFrame using pandas
# Analyze numbers	No goal provided	# Find the mean and standard deviation of a list of numbers

Add context when needed

If your prompt depends on earlier code, make sure to include that code in the prompt. Copilot works best when it can “see” surrounding context.

Prompt without context:

```
# Add error handling
```

Copilot doesn’t know what code you want to protect.

Prompt with context:

```
def divide(a, b):
    return a / b
```

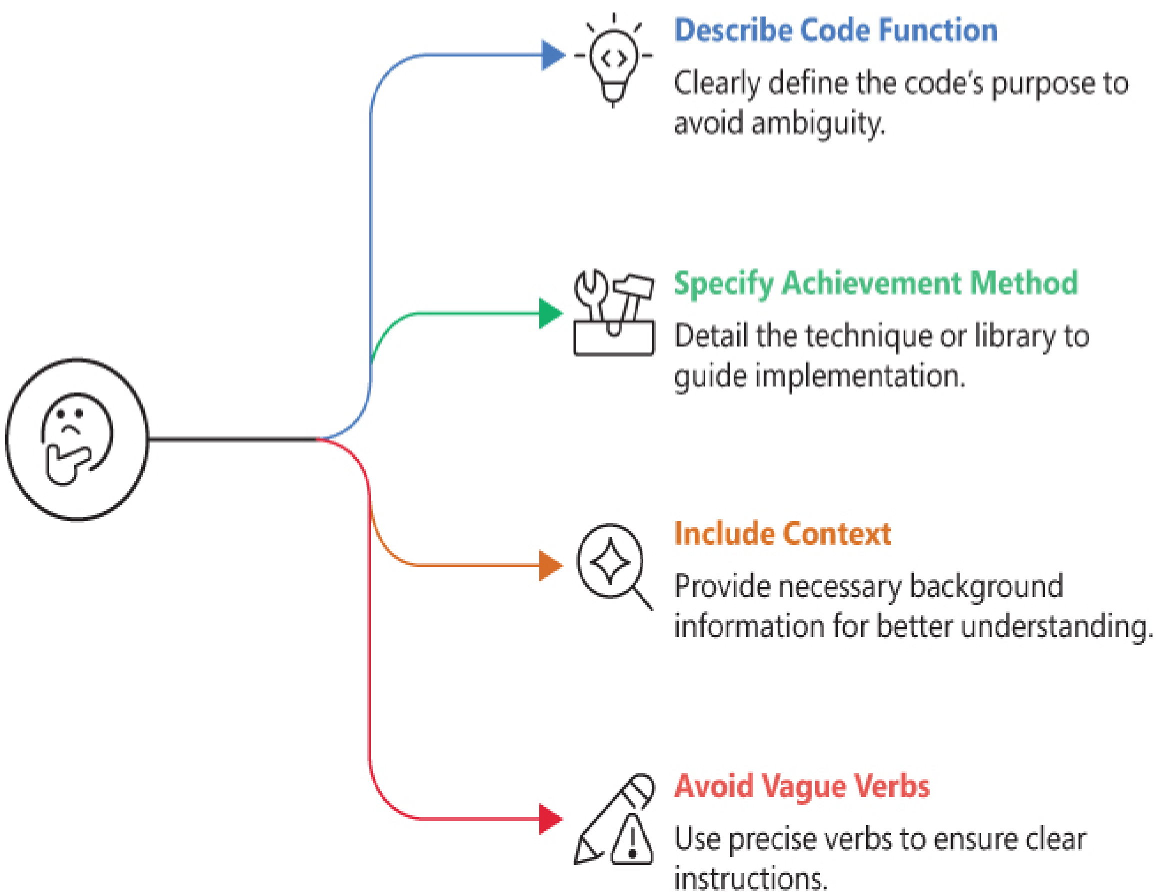
```
# Add error handling to catch ZeroDivisionError
```

Now Copilot can accurately wrap the function in a try/except block.

Prompt quality checks

Use this checklist to improve prompt clarity:

- Did I clearly describe what the code should do?
- Did I specify how the result should be achieved (library, technique)?
- Did I include context, if needed?
- Did I avoid vague verbs like "do," "make," and "clean" without details?



Strategies to improve code prompt clarity

Build an iterative workflow to continuously improve prompt results

Writing a great prompt doesn't always happen on the first try. That's why it's important to treat prompting as an iterative process—a cycle of writing, reviewing, refining, and repeating until you get the result you want.

This section guides you through building a practical, feedback-driven workflow to get better and faster results from GitHub Copilot, especially when working with Python code.

Start small and build up

A good approach is to break tasks into manageable pieces and let Copilot guide you step by step. Don't try to ask Copilot for an entire solution at once—especially for a complex task. Instead, start with a basic, clear prompt.

Initial prompt:

```
# Define a function to calculate factorial of a number
```

Once Copilot gives a valid response, you can expand your request with a follow-up prompt.

Follow-up prompt:

```
# Modify the factorial function to handle negative numbers an
```

Test and evaluate the output

When Copilot gives a suggestion, read it critically:

- Does the code work as intended?
- Is it readable and maintainable?

- Does it use the best Python practices?
- Are there edge cases it doesn't handle?
- If it’s not quite right, move to the next step and refine your prompt.

Important

Skipping evaluation of Copilot's first suggestion can result in subtle bugs. Always refine and retest based on the outcome of each prompt.

Refine based on what you get

If the result of a prompt is too generic or is incorrect, tweak your prompt. Add details, clarify the requirements, or suggest the method you want.

Initial prompt:

```
# Extract emails from a string
```

If the result misses edge cases or uses split(), revise it.

Improved prompt:

```
# Extract valid email addresses from a string using regex
```

This level of detail leads to more accurate and production-ready code.

Use comments and structure for context

When working in an actual Python script, Copilot benefits from seeing the file’s structure. Function headers, docstrings, and imports all help the AI “understand” your intent.

Effective prompt:

```
import re
def extract_emails(text):
  """
  Extract all valid email addresses from the input text.
  """
# Use regex to find email patterns
```

Even if your prompt is just a comment, placing it within a function or structured code block improves the results.

Maintain a prompt log

When working on longer projects, it's helpful to keep a prompt log, which is a record of what you asked Copilot and how it responded. This lets you:

- Track what works best.
- Reuse effective prompts.
- Avoid repeating mistakes.

A simple table or markdown file in your repository works well.

Experiment and compare

If you're unsure which approach is best, try multiple prompt variations and compare the outputs. Then test both results and choose the one that meets your needs. This is especially helpful when using new libraries or unfamiliar tasks.

Prompt 1:

```
# Sort a list of tuples by the second element
```

Prompt 2:

```
# Sort a list of (name, score) tuples by score in descending
```

See Also

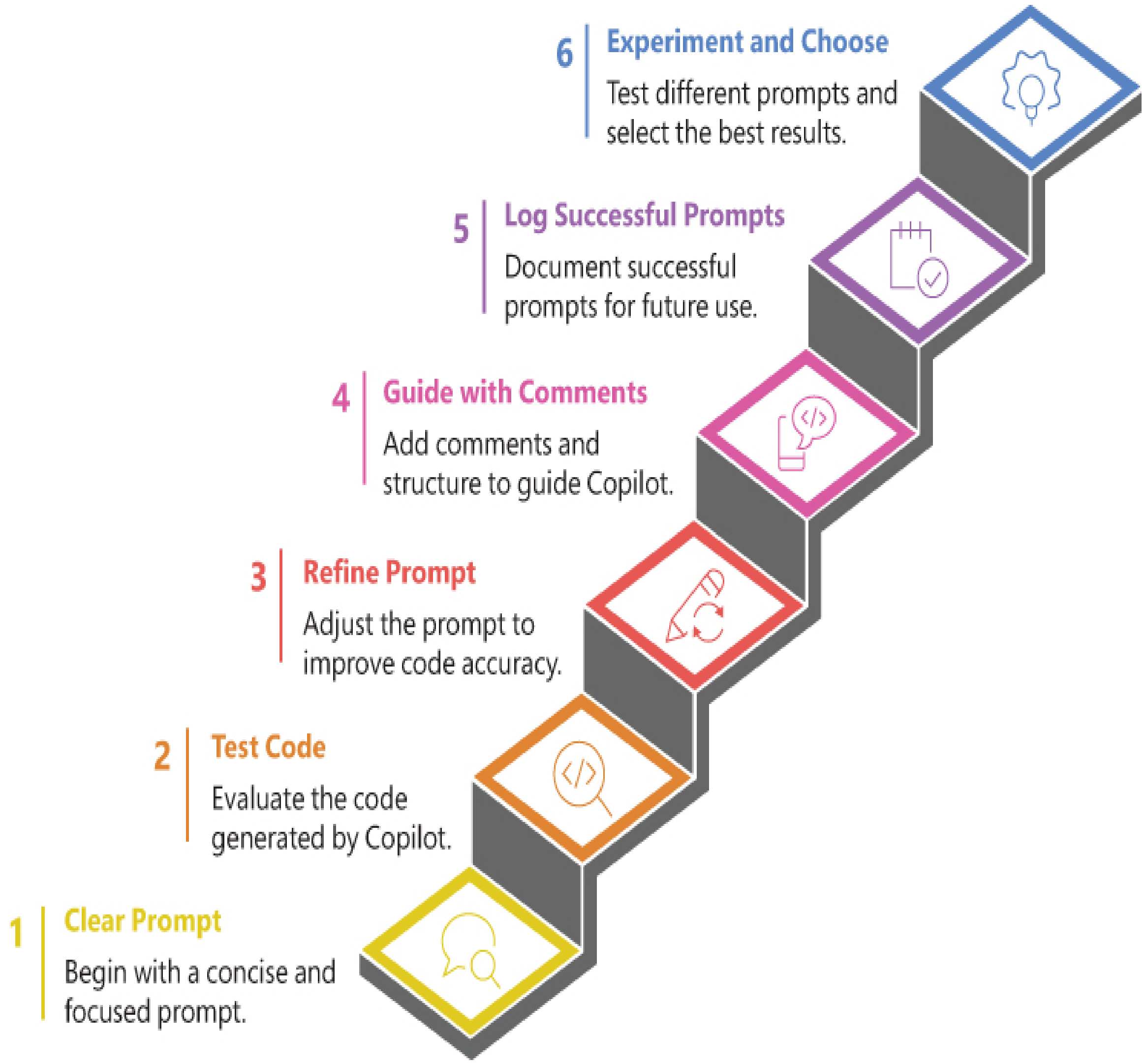
Chapter 10: Exploring the future of AI in software development provides insights into how prompt engineering will evolve with smarter AI models.

Keep improving through feedback

As your project evolves, so should your prompts. If the code changes, revisit earlier prompts and update them for clarity and accuracy. Prompting is not a "set and forget" process. It grows with your development process.

A solid iterative workflow with Copilot looks like the figure on the following page.

By practicing this cycle, you'll develop sharper prompting skills—and Copilot will become an even more powerful coding partner.



The GitHub Copilot prompt refinement process — from writing a clear prompt to testing, refining, documenting, and experimenting with improved code generation

Skills review

In this chapter, you learned how to:

- Write clear, specific prompts that improve GitHub Copilot's code suggestions.
- Use comment-based, function signature, and docstring-style prompts effectively.

- Refine vague or weak prompts to improve code quality and relevance.
- Apply reusable prompt templates for algorithms, testing, refactoring, and documentation.
- Customize prompts for specific programming languages, libraries, and frameworks such as NumPy, Pandas, and Flask.
- Identify common issues in ineffective prompts and revise them for better outcomes.
- Implement an iterative workflow for testing, refining, and improving prompt results continuously.

Practice tasks

No practice files are necessary to complete the practice tasks in this chapter. Follow the steps below to reinforce the concepts and tools introduced in each section.

Use an integrated development environment (IDE) such as Visual Studio Code with GitHub Copilot enabled and signed in. The following practice tasks use Python as the language. You can adapt them to other supported languages as needed.

To try the following tasks in Python, begin by creating a new Python file (e.g., tests_example.py) in the IDE.

Understand what makes a prompt for AI code generation effective

Practice writing and testing prompts with clear goals and context:

1. Write the following vague prompt and observe Copilot's output:

```
# calculate tax
```

2. Refine the prompt as follows to include context and observe Copilot's output:

```
# Calculate 10% sales tax for a given price and return tc
```

3. Compare the outputs of the two prompts and note how the specificity improves the result.

Use comment-based, function signature, and docstring prompts

Explore how different prompt styles guide Copilot:

1. Choose a task, such as checking if a number is even.
2. Prompt Copilot using a comment.
3. Prompt Copilot using a function signature.
4. Prompt Copilot using a docstring.
5. Review the generated code and determine which style yields the most complete result.

Refine or rewrite prompts to improve suggestion quality

Experiment with rewriting vague prompts to improve Copilot's performance:

1. Write the following prompt and observe Copilot's output:

```
# sort a list
```

2. Rewrite the prompt as follows and observe Copilot's output:

```
# Sort a list of strings alphabetically, ignoring case.
```

3. Compare and evaluate the output for each prompt for clarity and accuracy.

Apply prompt templates for algorithms, tests, refactoring, and more

Use structured prompt templates to generate reusable code:

1. Use the following prompt, based on the algorithm prompt template:

```
# Problem: Perform binary search
# Input: Sorted list and target value
# Output: Index of target or -1
```

2. Let Copilot complete the function and review its correctness.
3. Try a second template (e.g., # Refactor this code to improve readability) on an existing function and review the correctness of Copilot's output.

Customize prompts for specific languages, libraries, and frameworks

Test prompts that specify tools like pandas, Flask, or matplotlib:

1. Use the following prompt:

```
# Using pandas, drop rows with missing values from a Data
```

2. Observe whether Copilot uses the correct method (df.dropna()).
3. Change the prompt to reference a different library and observe the new output.

Identify and fix vague and inefficient prompts

Improve prompt quality to guide Copilot toward efficient solutions:

1. Use the following prompt:

```
# Count frequency of elements in a list.
```

2. Note whether Copilot uses a loop or collections.Counter.
3. Adjust the prompt to include collections.Counter and compare the results to the results of the initial prompt.

```
# Using collections.Counter, count the frequency of eleme
```

Build an iterative workflow to continuously improve prompt results

Develop and document a feedback-driven prompting process:

1. Use a basic prompt to start:

```
# Extract emails from a string.
```

2. Review Copilot’s output and note any missing features.
3. Refine the prompt:

```
# Extract valid email addresses from a string using regex
```

4. Log the prompts and outputs in a markdown file for future reference.

4

Enhancing code efficiency with AI assistance

In this chapter

- Apply advanced prompting to guide GitHub Copilot
- Automate repetitive tasks using AI
- Refactor inefficient or legacy code
- Implement clean coding practices
- Improve code readability and structure
- Evaluate AI suggestions for quality

Practice files

There are no practice files for this chapter.

In modern software development, efficiency isn't just about writing code faster; it's about writing better code with less effort. As projects grow in complexity, developers often spend significant time rewriting boilerplate logic, refactoring legacy code, or implementing common design patterns. GitHub Copilot, powered by OpenAI's Codex, offers a transformative solution: real-time AI assistance that streamlines development, automates routine tasks, and helps you apply coding best practices consistently.

This chapter introduces advanced prompting techniques and explores how Copilot can help you improve the quality, readability, and performance of your code. You'll learn how to automate repetitive patterns, generate more maintainable functions, and accelerate refactoring workflows using plain English instructions. By the end of this chapter, you'll be able to work more efficiently with Copilot—producing cleaner code with fewer errors and more confidence.

Apply advanced prompting to guide GitHub Copilot

The more clearly you communicate with GitHub Copilot, the more helpful it becomes. While Copilot can often understand vague or partial prompts, it is most useful when you give it enough structure to guide it toward what you really need. This section walks you through using advanced prompting techniques to get the most out of Copilot—whether you're building functions, designing loops, or generating full code structures.

You don't need to be an AI expert to create effective prompts. You just need to think about what you'd say to a junior developer who's helping you write the code. Copilot is like that teammate: It responds best when you give it enough detail, context, and intent.

Think about what happens when you give instructions to a junior developer: If you're too vague, they'll guess. If you're too detailed, you'll end up doing the work yourself. Copilot, like a junior developer, thrives in the middle ground.

Rule of thumb: Write prompts the way you'd explain the task to a smart intern

If you wouldn't expect a human to understand a cryptic comment like # handle this, then don't expect Copilot to. Instead, aim for short, descriptive prompts like this:

```
# check if a number is prime using a for loop
```

This single line gives Copilot context, logic, and intent.

Prompting strategies to try

Here are some simple, repeatable techniques to apply when guiding Copilot with more refined prompts.

Use "what" and "how" together

Instead of only telling Copilot *what* you want, also explain *how* you want it done.

Try this yourself:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., login_form.py).
3. Type the following comment on a blank line:

```
# generate a login form
```

4. Observe Copilot's suggestion and review the generated HTML-based login form (see as shown in below figure).

   Copilot-suggested code:

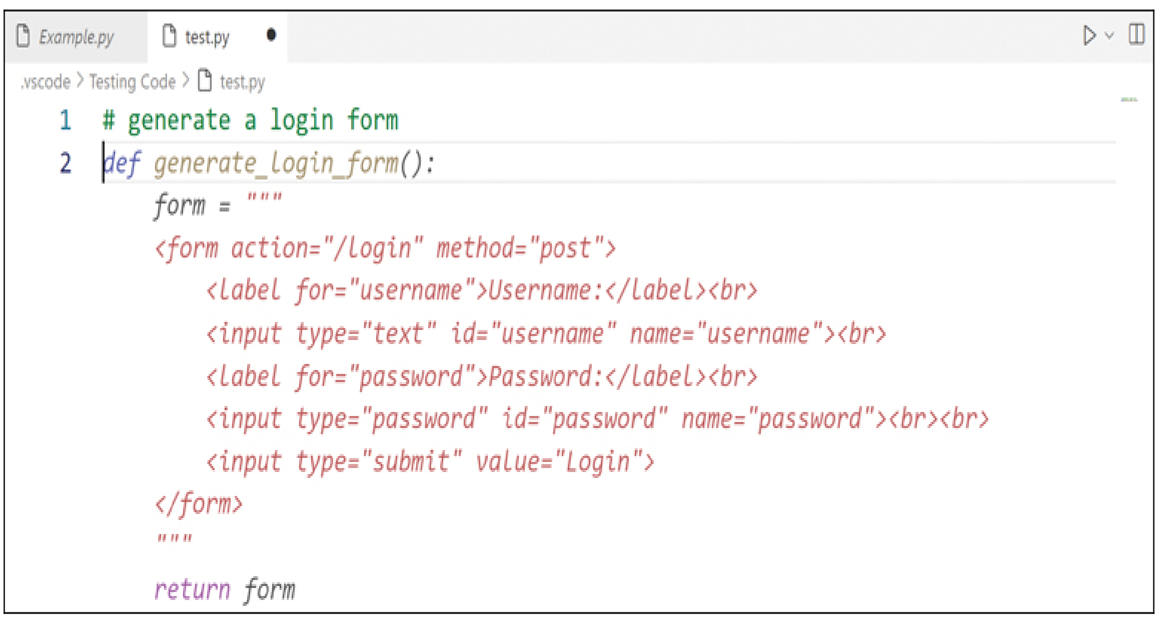
```
def generate_login_form():
    form = """
    <form action="/login" method="post">
        <label for="username">Username:</label><br>
```

```
        <input type="text" id="username" name="username">
        <label for="password">Password:</label><br>
        <input type="password" id="password"
        name="password"><br><br>
        <input type="submit" value="Login">
    </form>
    """
    return form
```

> **Important**
>
> The exact code generated by GitHub Copilot may vary depending on your file context and coding history, as well as the surrounding project code.

5. Tell Copilot how to generate the form:

```
#generate a login form using Tkinter library in Python
```



A GitHub Copilot-generated Python function that returns a login form as an HTML string, generated based on a comment describing the form's intent

Copilot-suggested code:

```
import tkinter as tk

def create_login_form():
    root = tk.Tk()
    root.title("Login Form")

    # Create labels and entry fields for username and password
    tk.Label(root, text="Username").grid(row=0)
    tk.Label(root, text="Password").grid(row=1)

    username_entry = tk.Entry(root)
    password_entry = tk.Entry(root, show="*")

    username_entry.grid(row=0, column=1)
    password_entry.grid(row=1, column=1)

    # Create a login button
    tk.Button(root, text="Login").grid(row=2, column=1)

    root.mainloop()
```

Small shifts like the one in this example can give Copilot more direction and often lead to more usable results.

Add examples inside comments

Giving Copilot an input or output example helps it meet your expectations.

A GitHub Copilot-generated Python script using Tkinter to build a login form, including labels, input fields, and layout structure

Try these steps:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., login_test.py).
3. Type the following comment on a blank line:

```
# Give me the complete code for sorting a list in ascending o
```

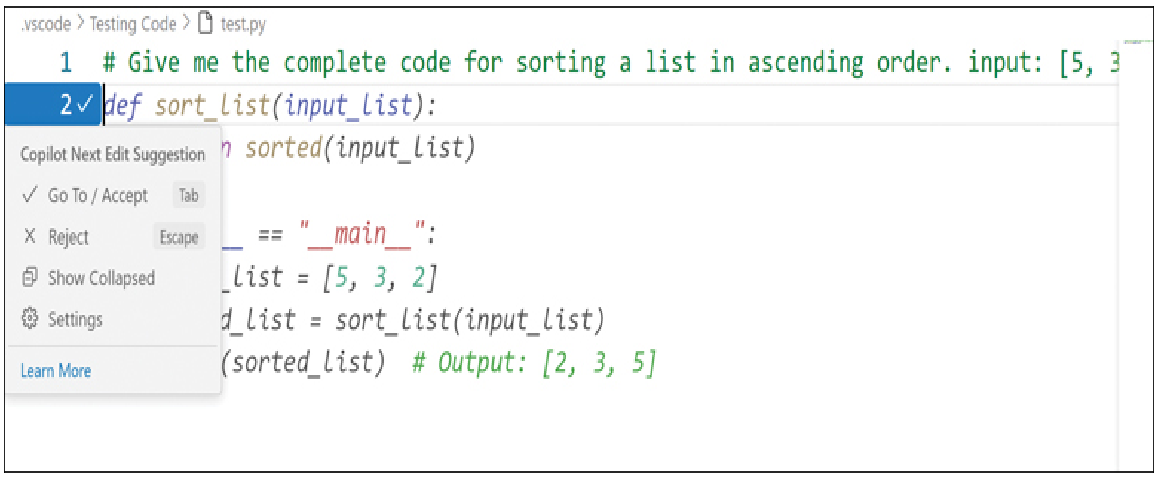
Copilot-suggested code:

```
def sort_list(input_list):
    return sorted(input_list)

if __name__ == "__main__":
    input_list = [5, 3, 2]
```

```
sorted_list = sort_list(input_list)
print(sorted_list)  # Output: [2, 3, 5]
```

Important

Efficiency does not only mean fewer lines of code—it also includes performance, scalability, and adherence to team coding standards



A GitHub Copilot suggestion for a Python function to sort a list based on a comment prompt

Break large tasks into smaller comments

Complex functions that are given in one big prompt can confuse Copilot. Breaking down a function into smaller bits helps Copilot give you better results.

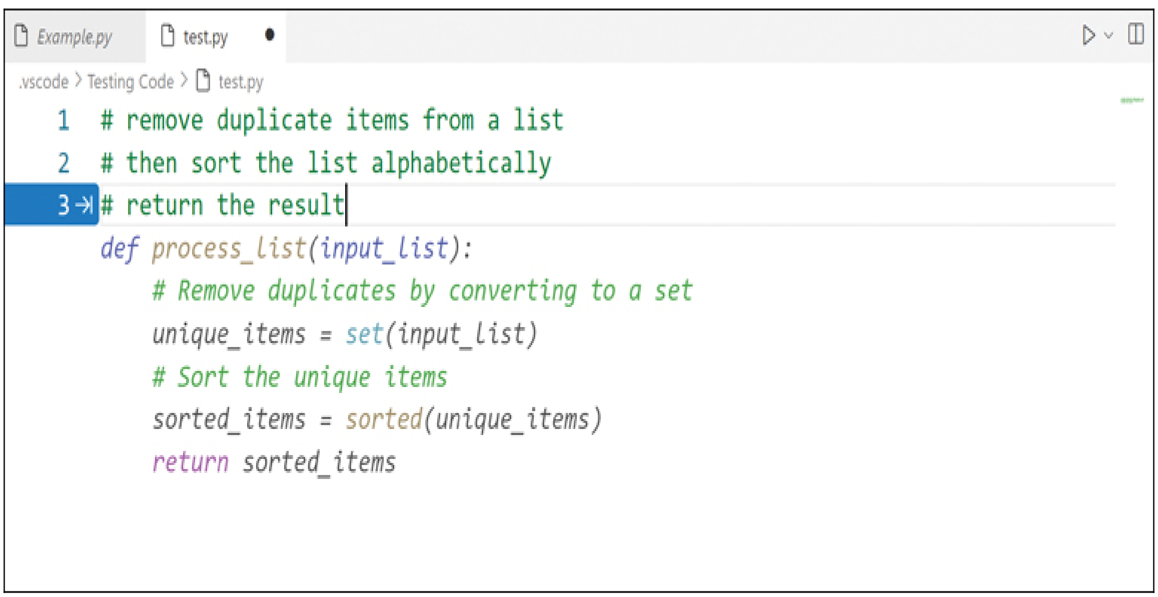
Try it for yourself:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., login_test.py).
3. Type the following comment on a blank line:

```
# remove duplicate items from a list
# then sort the list alphabetically
# return the result
```

Copilot-suggested code:

```
def process_list(input_list):
    # Remove duplicates by converting to a set
    unique_items = set(input_list)

    # Sort the unique items
    sorted_items = sorted(unique_items)

    return sorted_items
```



A Python function generated by GitHub Copilot to remove duplicates, sort a list alphabetically, and return the result

Specify the format you want

Do you need a dictionary? A list? JSON? Be specific in your prompt.

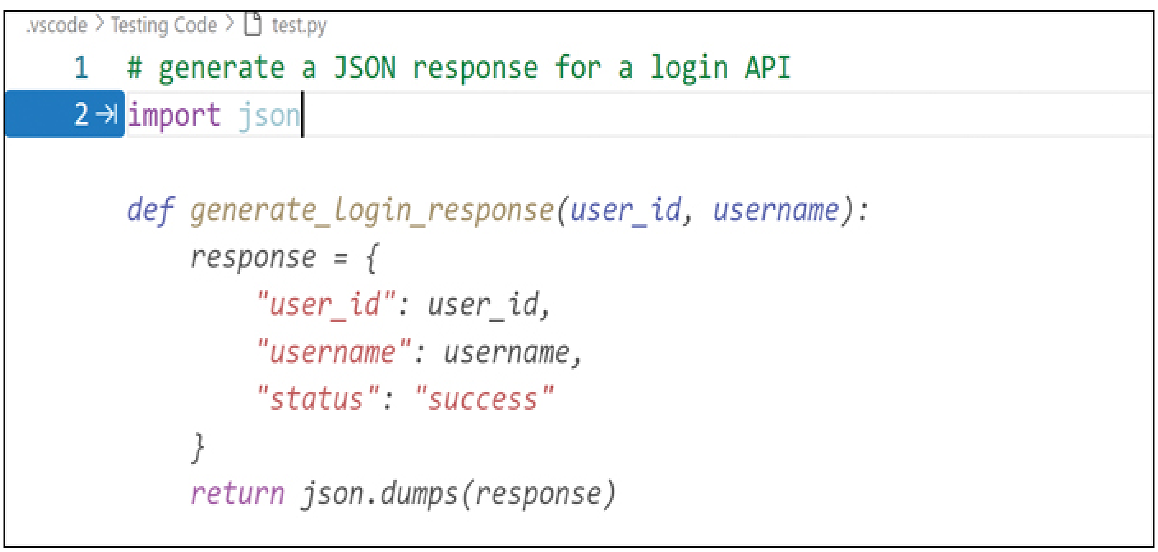
Try it out:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., login_test.py).
3. Type the following comment on a blank line:

```
# generate a JSON response for a login API
```

Copilot-suggested code:

```
import json

def generate_login_response(user_id, username):
    response = {
        "user_id": user_id,
        "username": username,
        "status": "success"
    }
    return json.dumps(response)
```



A Python function that generates a structured JSON string using the json module

Set the tone: Beginner or expert code

To set the tone

1. **Open** Visual Studio Code and make sure GitHub Copilot is enabled.
2. **Create** a new Python file (e.g., login_test.py).
3. **Type** the following comment on a blank line:

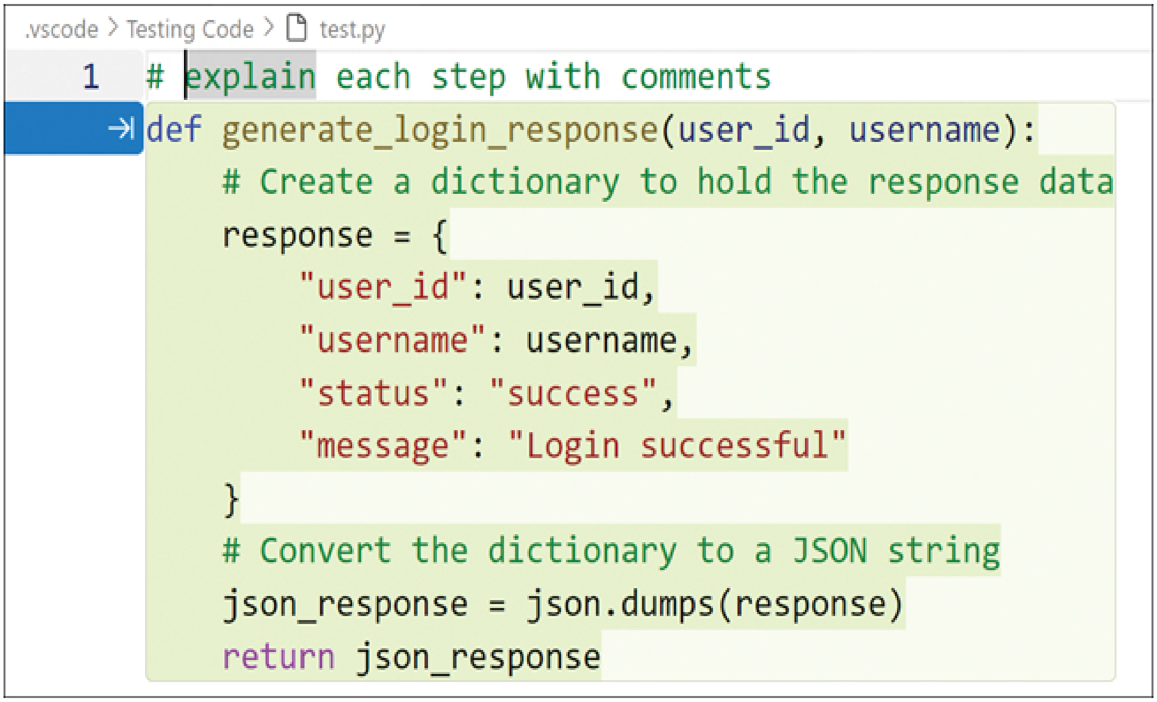
```
explain each step with comments
```

Copilot-suggested code:

```
import json
def generate_login_response(user_id, username):
    # Create a dictionary to hold the response data
    response = {
        "user_id": user_id,
        "username": username,
        "status": "success",
        "message": "Login successful"
    }
    # Convert the dictionary to a JSON string
    json_response = json.dumps(response)
    return json_response
```

Tip

Add context above the code, not just inline with it. Copilot reads the whole file. A comment at the top, like # This script handles customer orders and sends confirmation emails, sets the tone for everything that follows.



A GitHub Copilot-generated Python function that formats a user ID and username into a JSON response with a success status

Common prompting styles

To get the most out of GitHub Copilot, it helps to understand how different types of prompts influence its suggestions. The following table outlines common prompting styles and provides examples and the types of responses you can typically expect from GitHub Copilot.

Prompt type	Example	What Copilot typically generates
Descriptive task prompt	# create a function to check if a number is prime	A well-structured function using a loop or logic condition
Input/output pattern prompt	# input: [3, 2, 1], output: [1, 2, 3]	Sorting logic using Python's built-in functions

Prompt type	Example	What Copilot typically generates
Instruction and format prompt	# return a dictionary of word frequencies	A loop with a dictionary or using collections.Counter()
Multi-step prompt (chained instructions)	# Step 1: clean data\n# Step 2: calculate average	A structured multi-part function
Constraint/style prompt	# write compact version using list comprehension	Condensed logic using list/dict comprehensions

Important

Avoid overloading Copilot with long, rambling prompts. Break down your intent into simple, clear steps. If a prompt doesn't work, tweak it and try again. Prompting is an iterative process.

See Also

To learn about how Copilot adapts across different coding tasks, see Chapter 8: Using AI-powered development workflows in real-world scenarios.

Automate repetitive tasks using AI

Not every part of coding is creative. Some parts—like writing boilerplate code, setting up file operations, or validating inputs—are just plain repetitive. These tasks can take up a big chunk of your time, even though they rarely change from project to project.

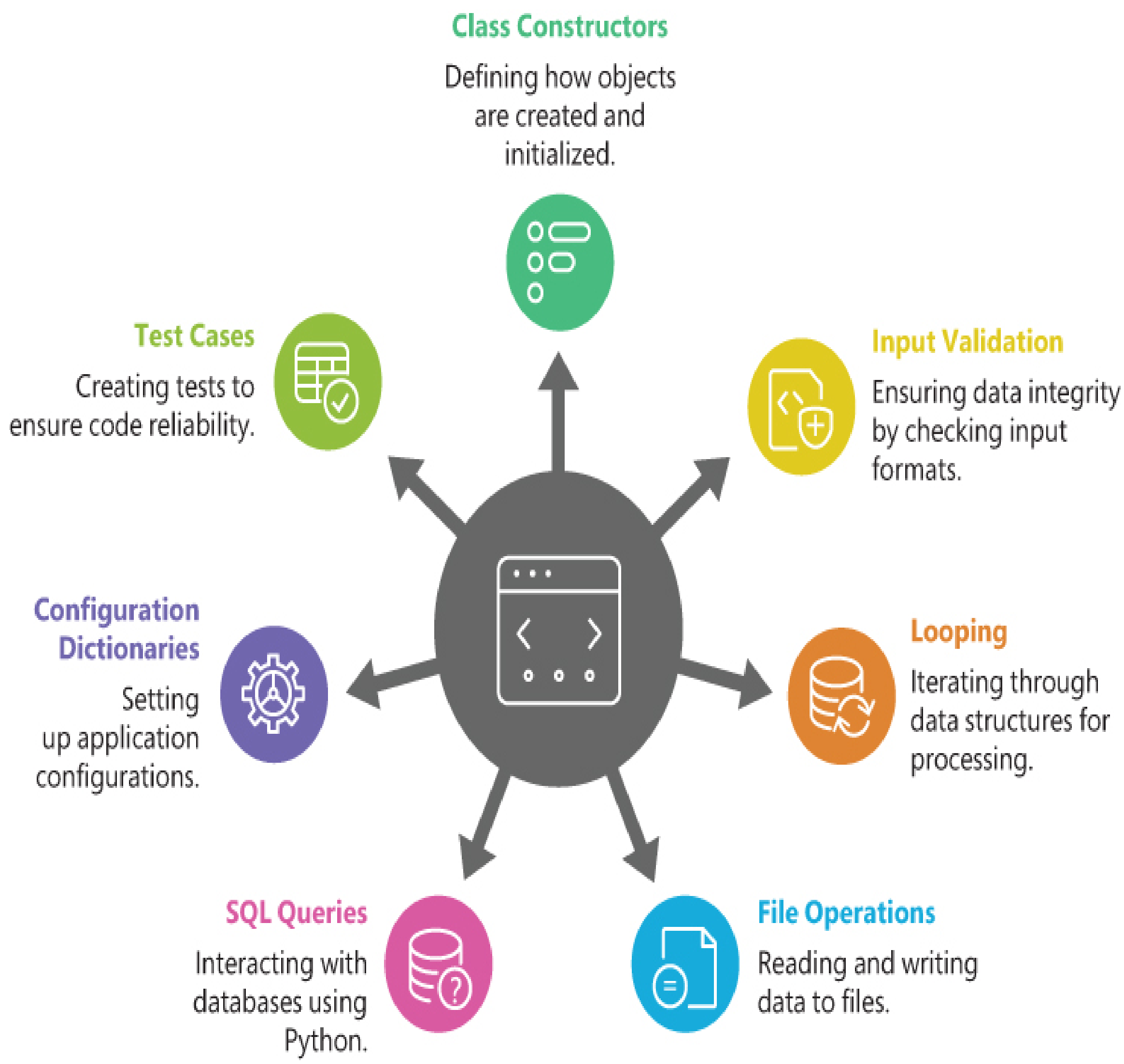
GitHub Copilot is especially useful for such tasks. It helps you fly through repetitive work by suggesting patterns, completing common code snippets, and even predicting what you'll need next. The result? You save time, reduce errors, and get to focus more on solving real problems.

What counts as a repetitive task?

Think about the parts of coding you could do in your sleep, such as:

- Defining class constructors.
- Writing input validation functions.
- Looping through lists or dictionaries.
- Reading from and writing to files.
- Writing basic SQL queries using Python libraries.
- Initializing configuration dictionaries.
- Setting up test cases with unittest or pytest.

Copilot has seen these patterns thousands of times and can fill them in for you quickly if you give it a little direction.



Common programming tasks suited for GitHub Copilot

Tip

If you find yourself writing the same logic more than twice, try prompting Copilot with a one-line comment instead. You may never need to write that logic manually again.

How to use Copilot to automate repetitive tasks

Here are a few ways to unlock Copilot's potential for performing repetitive tasks.

Use partial code as a trigger

Start typing the first few lines of a pattern you've used before, and Copilot will often finish it for you.

Example:

Type the following code and wait:

```
import sqlite3
def connect_to_database():
  connection = sqlite3.connect(
```

Copilot-suggested code:

```
    "students.db")
    cursor = connection.cursor()
    return connection, cursor
```

Comment out a task once, reuse often

Example:

Use a clear comment prompt like this:

```
# validate email address format using regex
```

Copilot-suggested code:

```
import re
def is_valid_email(email):
```

```
    pattern = r'^[\w\.-]+@[\w\.-]+\.\w+$'
    return re.match(pattern, email) is not None
```

Let Copilot handle repetitive structures

Example:

Instead of writing out a full try-except block every time, just type this:

```
# handle file not found error when opening a file
```

Copilot-suggested code:

```
try:
    with open("data.txt", "r") as file:
        contents = file.read()
except FileNotFoundError:
    print("The file was not found.")
```

Create class templates quickly

Example:

If you're creating multiple classes with similar structure, type:

```
# define a Student class with name and grade attributes
```

Copilot-suggested code:

```
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade
```

This is particularly useful in projects where you're frequently working with data models or objects.

Important

While Copilot can save you time, it's still up to you to check the logic, review edge cases, and add security or validation where needed. Automation doesn't mean skipping best practices.

See Also

For more on how Copilot handles real-world development scenarios, see Chapter 8: Using AI-powered development workflows in real-world scenarios. To learn how to evaluate and improve Copilot's generated logic, visit Chapter 6: Writing and automating tests with GitHub Copilot.

Real-world use cases for automating repetitive tasks with Copilot

GitHub Copilot shines when it comes to cutting out repetitive tasks. Whether you're a web developer writing form logic for the 100th time or a data analyst filtering out null values again, Copilot helps you do it faster and more consistently. The following are some domain-specific examples of how various roles can use Copilot to automate everyday coding tasks in Python.

Web developers

Web development involves a lot of repeated logic, including form validation, input sanitization, HTTP request handling, and route definitions. These aren't hard tasks, but they're time-consuming and easy to mess up.

Example: Generate a Flask route

Try it out:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., login_test.py).
3. Type the following comment on a blank line:

```
# Create a Flask route for user registration
from flask import Flask, request

app = Flask(__name__)

@app.route('/register', methods=['POST'])
def register_user():
  data = request.get_json()
  # validate input and store in database
```

Copilot will fill in the rest, including input checks and a success response, saving you from needing to do the boilerplate work.

Example: Form field validation

```
# validate email and password inputs in a registration form
def validate_user_input(email, password):
```

Copilot may automatically write logic to check email format and minimum password length.

Data analysts and data scientists

From loading data to cleaning it and performing basic transformations, Copilot can help automate steps you repeat daily in pandas, numpy, or scikit-learn.

Example: Clean missing values

```
# load CSV and drop rows with missing values
import pandas as pd
```

```
df = pd.read_csv("sales.csv")
df.dropna(inplace=True)
```

Copilot often fills in such patterns instantly.

Example: Create new columns from existing data

```
# create a new column 'total' as product of 'price' and 'quan
df['total'] = df['price'] * df['quantity']
```

Copilot often fills in such patterns instantly when it detects familiar data frame operations.

Backend developers

Working with databases, APIs, and authentication layers often involves repeated CRUD operations. Copilot can help scaffold these patterns quickly.

Example: SQL query automation

Try it out:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.

2. Create a new Python file (e.g., login_test.sql).
3. Type the following comment on a blank line:

```
# write a SQL query to fetch top 5 products by sales
```

Copilot may generate code like this:

```
query = """
SELECT product_name, total_sales
FROM products
ORDER BY total_sales DESC
LIMIT 5;
"""
```

Example: Prompt Copilot with natural language

```
# generate a SELECT query to get users who registered in the
```

Example: Exception handling

```
# handle connection error when connecting to PostgreSQL
```

```
try:
    connection = psycopg2.connect(...)
except psycopg2.OperationalError as e:
    print("Connection failed:", e)
```

Test engineers or developers writing tests

Test setup is one of the most repetitive parts of development. Copilot can scaffold tests using unittest, pytest, and other frameworks.

Example: Generate unit test for a calculator function

Try it out:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., test.py).
3. Type the following comment on a blank line:

```
# write unit tests for add(a, b)
```

Copilot may suggest the following code:

```
import unittest

class TestCalculator(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)
```

Example: Generate a full test suite

```
# test edge cases for divide(a, b) function
```

Students or beginner programmers

For students learning Python, Copilot helps automate repetitive learning patterns like loops, conditionals, and class definitions, speeding up practice without skipping understanding.

Example: Loop through a list and print squares

Try it out:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., test.py).

3. Type the following comment on a blank line:

```
# loop through numbers 1 to 10 and print their squares
```

Copilot may suggest the following code:

```
for i in range(1, 11):
    print(i ** 2)
```

Example: Define a class with attributes

Try it out:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., test.py).
3. Type the following comment on a blank line:

```
# define a Book class with title, author, and year
```

Copilot may suggest the following code:

```
class Book:
    def __init__(self, title, author, year):
        self.title = title
        self.author = author
        self.year = year
```

Tip

Use domain-specific terms in your prompts (e.g., "Flask route," "pandas dataframe," "unit test") to help Copilot generate better-targeted suggestions.

See Also

Explore how Copilot adapts to multiple coding styles and teams in Chapter 7: Using GitHub Copilot for code reviews and collaboration. For more advanced prompting tricks across projects, visit Chapter 3: Writing effective prompts for AI-powered coding.

Refactor inefficient or legacy code

Every developer eventually has to deal with legacy code—whether it's something they wrote last year or inherited from someone else. Legacy code might work, but it’s often messy, repetitive, or inefficient. Refactoring is the process of improving code without changing what it does.

GitHub Copilot can help speed up this process by analyzing your existing code structure and suggesting cleaner, more modern ways to write the same logic. It doesn’t rewrite the whole codebase for you, but it can certainly serve as another developer, pointing out a simpler or smarter way.

What does “inefficient” or “legacy” look like? Here are some signs that your code might need to be refactored:

- It repeats the same logic in multiple places.
- It uses outdated practices (e.g., manual loops instead of comprehensions).
- It's hard to read or lacks clear naming.
- It has deeply nested conditionals or unnecessarily long functions.
- It's written for older Python versions or unsupported libraries.

How Copilot helps with refactoring

While Copilot doesn’t understand your project’s architecture or business rules, it does understand clean coding patterns. You can use this to your advantage when modernizing or simplifying old code.

This section covers some practical ways to apply Copilot for refactoring.

Use descriptive prompts to trigger better structure

Start by copying the legacy code into your file. Then, just above it, write a comment like this:

```
# refactor the following function to be more efficient
```

Copilot will likely suggest a revised version using updated patterns or Pythonic structures. Let's look at an example.

Legacy code:

```
def square_numbers(numbers):
    result = []
    for n in numbers:
        result.append(n * n)
    return result
```

Prompt:

```
# refactor the following function to use list comprehension
```

Copilot-suggested code:

```
def square_numbers(numbers):
    return [n * n for n in numbers]
```

Simplify conditionals and loops

Older code might use verbose logic that Copilot can help streamline. Let's look at an example.

Legacy code:

```
def check_age(age):
    if age >= 18:
        return True
    else:
        return False
```

Prompt:

```
# simplify the conditional logic
```

Copilot-suggested code:

```
CopyEdit
def check_age(age):
    return age >= 18
```

Improve variable and function names

You can ask Copilot to rename variables or functions for clarity.

Example:

Try it:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., test.py).
3. Start writing your code, and after some lines, type the following comment on a blank line:

```
# improve the naming in this function for readability
```

You can use a prompt like this to have Copilot turn vague names like x or temp into more meaningful names, like price or user_input.

Break large functions into smaller ones

Functions that do too much are harder to test and maintain. You can guide Copilot to split them, using a prompt like this:

```
# break this function into smaller helper functions
```

Copilot often suggests logical divisions based on input parsing, processing, and output formatting.

Replace outdated syntax or patterns

If you're updating Python 2–style code or refactoring code that uses old libraries, Copilot can help bring it up to date.

Example:

Try it:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., test.py).
3. Type the following code line and comment:

```
Print("Hello,"+name+"!")
# update this code to use f-strings instead of concatenation
```

Copilot may suggest code like this:

```
print(f"Hello, {name}!")
```

Tip

> Try using prompts like “refactor using best practices,” “simplify this function,” or “make this more Pythonic.” Copilot understands these cues well and often provides smart suggestions.

When improving older or less efficient code, GitHub Copilot can be guided to refactor common patterns into more modern, readable alternatives.

Legacy code patterns vs. modern alternatives

The following table compares common legacy code patterns with the more modern, readable alternatives GitHub Copilot can generate when guided with specific prompts.

Legacy code pattern	Copilot prompt	Expected refactored output
Manual loop to build a list	# refactor using list comprehension	List comprehension version with cleaner syntax
Verbose if-else return block	# simplify the logic	One-line Boolean return or early return structure
Repeated logic blocks in the same function	# extract helper function for reuse	Smaller, reusable functions with descriptive names
Long function with mixed responsibilities	# break into smaller functions	Modular functions that separate logic and improve testing
Cryptic variable/function names	# improve naming for clarity	Readable, self-explanatory identifiers

Important

Copilot doesn't always understand the context of an application. While it can refactor for syntax and structure, it won't know your edge cases, security concerns, or domain-specific logic. Always test refactored code thoroughly before deploying it.

See Also

For more on writing tests to validate Copilot's refactored code, see Chapter 6: Writing and automating tests with GitHub Copilot. To understand how Copilot fits into team workflows for improving code quality, check out Chapter 7: Using GitHub Copilot for code reviews and collaboration.

Boosting performance with Copilot

Code that works is good; code that works efficiently is even better. As a project grows, small inefficiencies can become major slowdowns. Whether it's a slow loop, an unnecessary operation, or an expensive recursive call, performance problems often hide in plain sight.

GitHub Copilot can help you identify and replace inefficient patterns with cleaner, faster alternatives. While it doesn't "profile" code, it has learned from millions of examples and often suggests optimized solutions if you guide it with the right prompts.

This section explores how to use Copilot to make your code cleaner and also faster and smarter.

Prompting Copilot for performance

When working with large datasets, tight loops, or complex conditions, ask Copilot to rethink your logic with prompts like this:

```
# optimize this for performance
# replace nested loops with set lookup
# refactor using better time complexity
# avoid recomputing values inside loop
```

Copilot will often respond with more efficient logic, such as list comprehensions, set-based filtering, or memorization techniques.

Slow vs. fast: Real-world examples

Inefficient code: Repeated lookups in a list

```
def find_common_elements(list1, list2):
    return [x for x in list1 if x in list2]
```

This works, but x in list2 is O(n) and repeated inside a loop, making it $O(n^2)$.

Prompt:

```
# optimize for large datasets
```

Copilot-optimized code: Use sets for faster lookup

```
def find_common_elements(list1, list2):
    set2 = set(list2)
    return [x for x in list1 if x in set2]
```

GitHub Copilot can suggest a variety of performance-focused improvements when you guide it with precise prompts. While it won't automatically profile your code, it draws on common optimization patterns to help you replace inefficient logic with faster, more scalable alternatives. The table below outlines common inefficiencies you might encounter and the types of optimizations Copilot is likely to propose.

Inefficiency type	Copilot optimization pattern
Nested loops	Replaces with sets or dictionaries for faster lookup
Repeated calculations	Moves computations outside loops or uses caching
Redundant conditions	Simplifies logic with early returns
Manual sorting or filtering	Uses built-in functions like sorted(), filter(), or list comprehensions
Large file/data processing	Suggests chunking, buffering, or generator-based processing

Optimize space, not just time

Memory-heavy code: Keeping large intermediate results

```
results = [process(x) for x in big_list]
```

Prompt:

```
# reduce memory usage using generator
```

Efficient code: Use a generator expression

```
for result in (process(x) for x in big_list):
    handle(result)
```

Tip

Try using specific language in your prompts: instead of using # make this better, use # improve performance with set lookup or # use built-in functions to optimize. Copilot responds to precise prompts with better results.

Important

Copilot won't automatically know your performance bottlenecks. Use profiling tools like cProfile or timeit to measure real-world performance and then apply targeted refactoring with Copilot.

Implement clean coding practices

Clean code isn't about perfection; it's about communication. Code is read more often than it's written, and your future self (or your teammate) will thank you for code that's clear, intentional, and consistent.

GitHub Copilot can help accelerate your coding, but it's your responsibility to keep that code clean. While Copilot generates patterns that are mostly aligned with best practices, it's still up to you to review, refine, and restructure the output. In this section, you'll learn how to apply clean coding principles with Copilot by your side to turn raw code into readable, reusable, and professional-quality work.

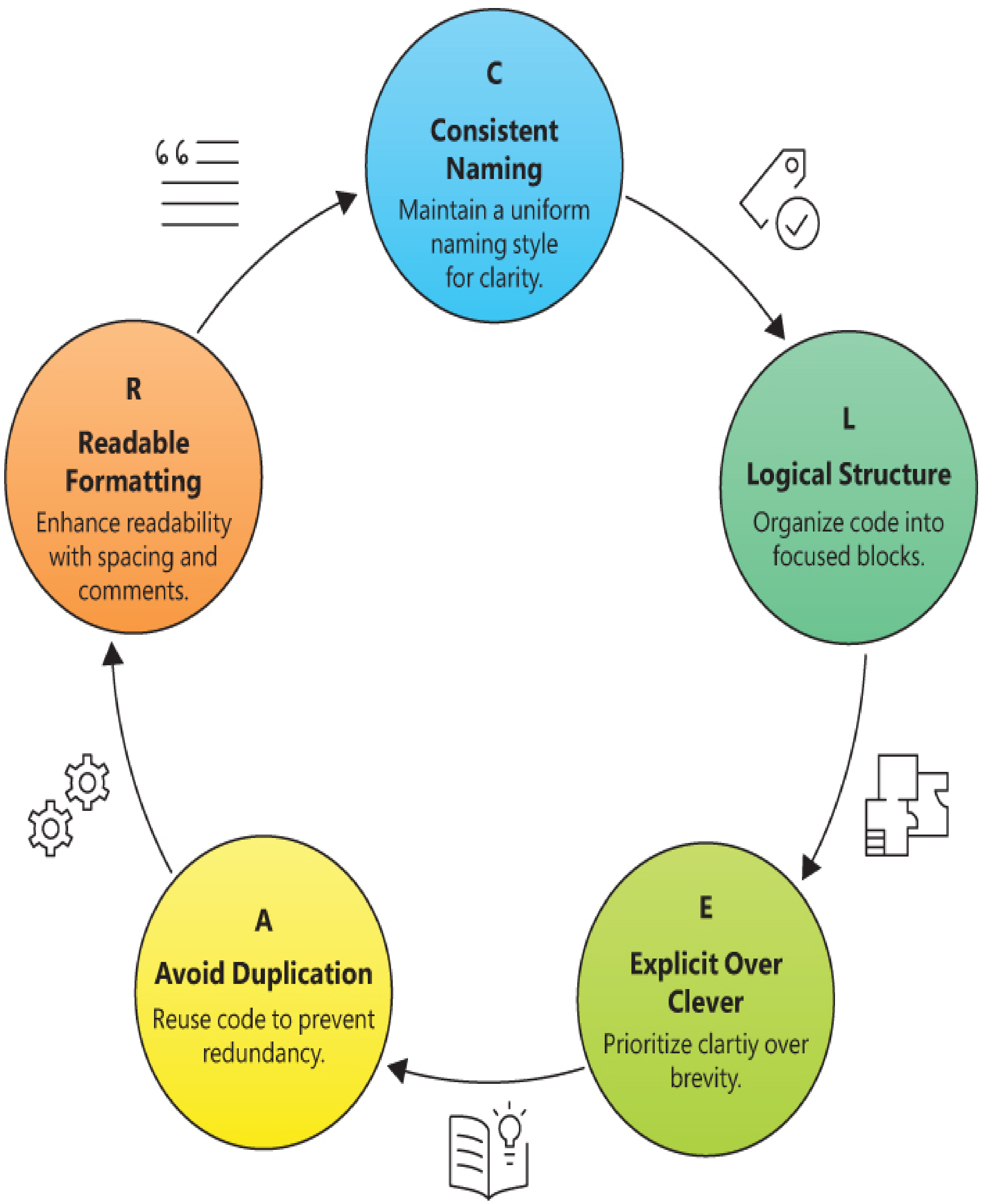
Clean code isn't just nice to have. It reduces misunderstandings, makes onboarding easier, and helps teams move faster. Think of Copilot as a starter for your ideas, not the final author of your code. The final responsibility for structure, clarity, and maintainability is still yours.

The CLEAR framework to guide clean coding

The CLEAR framework is based on a clever acronym that guides clean coding:

- **C**onsistent naming: Use meaningful, self-explanatory names for variables, functions, and classes.
- **L**ogical structure: Organize code so that it flows predictably and is easy to follow.
- **E**xplicit over clever: Avoid overly clever or compressed logic that sacrifices clarity.
- **A**void duplication: Eliminate repeated code by extracting reusable functions or components.
- **R**eadable formatting: Follow style guidelines (e.g., PEP 8) and use whitespace for clarity.

Think of the CLEAR framework as your personal code quality checklist—whether you’re reviewing Copilot’s suggestions or writing code yourself. For example, Copilot might generate a function that technically works but uses vague names, compressed logic, or poor formatting. Applying CLEAR involves making names consistent, structuring logic logically, avoiding clever one-liners, eliminating repetition, and formatting for readability. It can help you turn a rough draft into clean, professional code that's easy to understand and maintain.



The CLEAR framework for writing maintainable and professional code

Prompts for clean coding:

```
# rewrite this function using clear naming and structure
# clean up variable naming and function length
# refactor to remove repeated code
# reformat to follow Python best practices
```

Clean code checklist

Use this checklist after generating code with Copilot or refactoring your own:

- ✔ Are the variable and function names self-explanatory?
- ✔ Does each function do one thing only?
- ✔ Are comments helpful and not obvious or outdated?
- ✔ Are there any repeated blocks of code that should be extracted?
- ✔ Is the logic broken into readable sections?
- ✔ Is there unnecessary complexity that could be simplified?
- ✔ Does the code follow the Python style guide (PEP 8)?
- ✔ Are error-handling cases covered cleanly?

Clean code isn't just nice to have; it reduces misunderstandings, makes onboarding easier, and helps teams move faster. Think of Copilot as a starter for your ideas, not the final author of your code. The final responsibility for structure, clarity, and maintainability is still yours.

Examples of using Copilot to clean code

GitHub Copilot can do more than just generate new code; it can also help you improve the quality of existing code by making it cleaner, clearer, and easier to maintain. The following examples show how Copilot can transform unclear or overly complex code into more readable, structured, and professional output when guided with the right prompts.

Example 1: Improve function clarity

To understand how Copilot can improve function clarity, try these steps:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., test.py).
3. Type the following code and prompt:

```
def fn(x, y):

    return x * y + x - y
```

```
# clean up this function for better readability and naming
```

Copilot might suggest revising the code so it looks like this:

```
def calculate_weighted_score(base, modifier):
    return base * modifier + base - modifier
```

Example 2: Break long logic into helper functions

Prompt:

```
# break this function into smaller parts
```

Copilot will split long blocks into manageable subfunctions, such as functions for input parsing, core logic, and output formatting.

Tip

Use docstrings to summarize what a function does. Copilot often generates them if you start typing triple quotes ("""). This encourages better documentation and understanding.

Important

Don’t let speed compromise quality. Copilot might suggest shortcuts or overly clever solutions. Choose clarity over cleverness every time.

Common clean coding mistakes Copilot helps avoid

Even experienced developers fall into bad habits that can make code harder to read, maintain, or extend—especially when working under pressure. Clean coding is about writing code that is understandable, intentional, and easy to work with. It benefits you now as well as your future self and your team.

GitHub Copilot can help you avoid many common clean coding pitfalls. It draws from vast examples of community best practices and, when prompted effectively, generates cleaner alternatives than most developers might write under tight deadlines. However, it’s not flawless, and it will sometimes mimic poor practices. You need to know what to avoid so you can recognize and refine Copilot’s suggestions when needed.

The following table lists common clean coding mistakes and shows how GitHub Copilot can help you address them when prompted effectively.

Mistake	What it looks like	How Copilot can help

Mistake	What it looks like	How Copilot can help
Vague or one-letter variable names	x = 100 or def f(x, y):	Copilot often suggests better names. Prompt: # rename for clarity
Overly long functions	Functions doing multiple things, sometimes 30+ lines long	Prompt: # split this into smaller helper functions
Repeated blocks of code (copy/paste logic)	The same if or for logic copied across functions	Prompt: # extract common logic into a reusable function
Deeply nested conditionals	if...else if...else if...else chains	Prompt: # simplify logic with early returns
Magic numbers or unexplained constants	if score > 85: instead of if score > PASSING_GRADE:	Copilot can suggests constants. Prompt: # use constant
Inconsistent formatting	Irregular indentation, missing blank lines	Copilot generally follows the file's formatting style
Poor function names	def do_stuff(): or process()	Prompt: # improve function naming for clarity
Missing docstrings or comments	Functions without explanations or intent	Prompt: # add docstring to explain function

Mistake	What it looks like	How Copilot can help
Writing clever instead of readable code	One-liners or obscure tricks just to save lines	Prompt: # rewrite this for readability
Mixing too many concerns in one place	Logic and formatting and output in one function	Copilot helps when asked to. Prompt: # separate logic and output

Tip

Start your code cleanup by prompting Copilot with a comment describing what the code should do. If it generates the same logic in fewer lines or with clearer naming, consider replacing your original version.

Important

Copilot may occasionally replicate these mistakes if your current code includes them. Always review its suggestions and use comments as a steering wheel—not as an autopilot switch.

See Also

To learn how Copilot fits into collaboration and code review workflows, see Chapter 7: Using GitHub Copilot for code reviews and collaboration.

Copilot-powered developer toolkit

Clean coding isn't just about writing; it's about developing habits that produce consistent, high-quality code. Think of applying clean coding practices as sharpening your tools before you start building. Whereas GitHub Copilot helps you write faster, your coding toolkit helps you write better.

This section offers a practical, Copilot-assisted developer toolkit you can use throughout your workflow. These techniques and prompt templates are designed to help you reinforce clean coding practices, catch potential issues early, and streamline your development experience.

Use the following table toolkit as a checklist, a quick reference, or a guide when working on new projects, reviewing old code, or mentoring others.

Tool/habit	**How to use it with Copilot**
Descriptive function and variable names	Prompt: # rename for clarity or # improve naming in this function
Modular code (small, focused functions)	Prompt: # break this into smaller helper functions
Readable structure and formatting	Prompt: # rewrite for readability or follow well-formatted examples
Avoid duplicated logic	Prompt: # extract reusable function
Error handling for edge cases	Prompt: # add exception handling for division by zero
Docstrings and comments	Prompt: # add docstring to describe this function
Replace magic numbers	Prompt: # use named constant instead of hard-coded value

Tool/habit	How to use it with Copilot
Defensive programming patterns	Prompt: # validate input before processing
Consistent coding style (e.g., PEP 8)	Copilot follows your file’s indentation and style, so lead by example
Test scaffolding	Prompt: # write test cases for this function using unittest

How to use this toolkit:

- At the beginning of a project: Set a file format and coding style and use prompts to scaffold clean, reusable components.
- While coding: Use inline prompts to guide Copilot toward clarity and structure.
- Before committing: Review Copilot's suggestions against the toolkit checklist. If something feels “off,” prompt it again.
- When reviewing others’ code: Use this as a baseline to suggest improvements that Copilot can help implement quickly.

Tip

Make a habit of combining what and why in your prompts, like # check if a user is active before processing request. Copilot understands intent better when you give it purpose.

Improve code readability and structure

Code readability is the difference between getting stuck debugging for hours and understanding your logic in seconds. You may write code once, but you'll read it dozens of times. And if you work on a team, other people will, too.

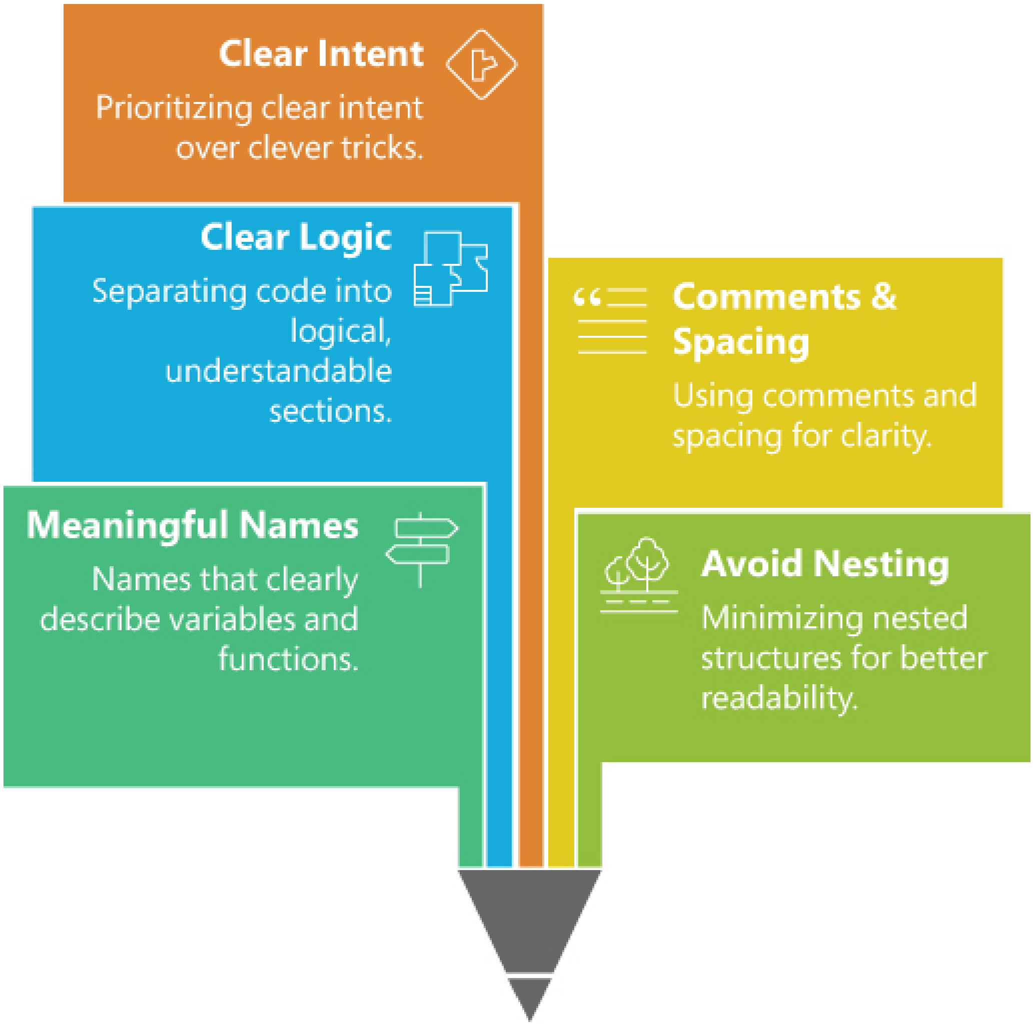
Structure goes hand in hand with readability. Good structure makes your code predictable, testable, and easier to maintain. GitHub Copilot can support you in creating cleaner code—but it's your job to guide it toward readable and well-organized output.

This section shows how to work with Copilot to improve readability and code structure, using practical patterns and simple habits in Python.

What makes code readable?

Readable code is like good writing:

- It has meaningful names.
- It avoids unnecessary nesting.
- It separates logic into clear sections.
- It uses comments and blank lines for breathing space.
- It avoids "clever" tricks in favor of clear intent.



Essential coding principles for clean code advocacy

You don't need to over-comment or use fancy syntax. As you're writing, imagine that your future self will read your code at the end of a long day.

Strategies to improve readability and structure with Copilot

To make the most of GitHub Copilot while keeping your code clean and professional, it's helpful to follow a few simple strategies. These practices will help you transform Copilot's raw suggestions into well-structured, readable code that's easy to understand and maintain.

Use expressive function and variable names

To understand how to use expressive function and variable names, try the following:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., test.py).
3. Type the following code and prompt:

```
def fx(x):
  return x + 10
# improve naming for clarity
```

Copilot will often rename parameters and functions using more descriptive words, based on your project context.

Split long functions into smaller units

If a function is more than 15 or 20 lines, chances are it's doing too much.

Initial code:

```
def process_student_data(data):
    cleaned = [d.strip() for d in data if d]
    scores = [int(x.split(',')[1]) for x in cleaned]
    avg = sum(scores) / len(scores)
    return avg
```

Prompt:

```
# break this function into smaller helper functions
```

Copilot-suggested code:

```
def process_student_data(data):
    cleaned = clean_data(data)
    scores = extract_scores(cleaned)
    return calculate_average(scores)
```

Flatten deeply nested logic

Nested loops and conditionals reduce readability. Use early returns and clearer control flow.

Initial code:

```
def check_access(user):
    if user:
        if user.is_active:
            if user.role == "admin":
                return True
```

Prompt:

```
# simplify nested if-else structure
```

Copilot-suggested code:

```
def check_access(user):
    if not user or not user.is_active:
        return False
    return user.role == "admin"
```

Use blank lines and consistent formatting

Good structure includes visual spacing, just like paragraphs in a book.

Well-structured code:

```
def send_email(to, subject, body):
    headers = {"To": to, "Subject": subject}
```

```
message = f"{headers}\n\n{body}"

# Send logic
print("Email sent:", message)
```

Tip

Copilot typically follows indentation and spacing cues in your code. Keep your formatting clean, and Copilot will do the same.

Write for the reader, not for the compiler

Yes, your code works, but can someone else understand it at a glance? You can use Copilot to help reword or rewrite confusing parts.

Prompt:

```
# rewrite this logic for clarity and maintainability
```

In response, Copilot might simplify your logic, flatten unnecessary complexity, or restructure your loops.

Tip

When in doubt, read your code out loud. If you have to pause to understand a function name or logic block, Copilot probably needs a clearer prompt, too.

Important

Copilot may suggest compact one-liners or complex list comprehensions that work—but that aren't necessarily readable.

> Choose clarity over cleverness, especially in collaborative environments.

Evaluate AI suggestions for quality

GitHub Copilot is fast, confident, and surprisingly helpful. But it's not perfect. Sometimes it writes clean, efficient code. Other times it gives you bloated, outdated, or even insecure logic. That's why it's important to treat every suggestion Copilot makes as a starting point—not as a final solution.

In this section, you'll learn how to critically evaluate Copilot's suggestions for correctness, performance, readability, and safety. Think of yourself as the code reviewer: Copilot writes the first draft, and you polish it.

Adopt a Reviewer's Mindset

Copilot saves time, but it doesn't replace your judgment. Always ask:

- Does this code work for my specific input/output needs?
- Is it efficient and scalable?
- Is it easy to read and modify later?
- Could this code introduce bugs, edge case failures, or security issues?

Let's walk through some ways to evaluate Copilot's suggestions effectively.

Check logic against sample inputs

Before trusting a function Copilot wrote, test it with a few edge cases.

Try this example:

1. Open Visual Studio Code and make sure GitHub Copilot is enabled.
2. Create a new Python file (e.g., test.py).

3. Type the following prompt:

```
# generate a function to check if a string is a palindrome
def is_palindrome(s):
  return s == s[::-1]
```

This works for simple strings like “racecar”. But does it work with mixed case, punctuation, or spaces? To find out, try this prompt:

```
print(is_palindrome(“RaceCar”))    # False
print(is_palindrome(“nurses run”)) # False
```

Now you know: This code needs to be refined.

New prompt:

```
# update to ignore case and spaces
```

Evaluate performance for large datasets

Copilot might suggest inefficient patterns for large datasets.

Prompt:

```
# remove duplicates from list
```

```
def remove_duplicates(lst):
    result = []
    for item in lst:
        if item not in result:
            result.append(item)
    return result
```

This works—but it's $O(n^2)$ performance, meaning the time it takes grows quickly as the list gets larger, since each lookup scans the result list again. For large lists, you can use a set for better efficiency:

New prompt:

```
def remove_duplicates(lst):
  return list(dict.fromkeys(lst))
```

This version is faster and cleaner.

Look for hidden assumptions

Copilot might suggest logic that works only under certain conditions—such as assuming that inputs are always valid.

Prompt:

```
# generate function to divide two numbers
```

Copilot-suggested code:

```
def divide(a, b):
    return a / b
```

To text this code, consider:

- What if b = 0?
- What if a or b is a string?

Tip

Wrap code that could raise errors (such as dividing by zero) in a try-except block, and write tests that cover these error conditions to ensure your program handles them safely.

Validate correctness with tests

Copilot can help you write tests to verify its own output. If you're unsure about a function, ask Copilot.

Prompt:

```
# write test cases for validate_email function
```

Then run the tests that Copilot writes. If they fail, refine your function and retest.

Watch for security gaps

Prompt:

```
# evaluate a user expression
```

Copilot-suggested code:

```
eval(user_input)
```

This works, but it's a massive security risk. Using eval(user_input) executes arbitrary user input as code, which could allow attackers to run malicious commands. Evaluate whether safer alternatives exist (e.g., ast.literal_eval in Python).

Ask Copilot to improve its own output

If you don't like a suggestion, tell Copilot why:

- "Too complex"

- “Not readable”
- “Needs error handling”
- “Simplify this logic”

Prompt:

```
# rewrite this for better readability and error handling
```

When you give it such a prompt, Copilot will often give you an improved version.

> **Tip**
>
> Treat Copilot like a junior developer who is helpful, fast, and smart but needs review, context, and correction. If you wouldn’t merge it without a second opinion, don’t copy it blindly.

> **Important**
>
> Don’t assume that working code is good code. Always test, read, and think critically—especially in production, team, or security-sensitive environments.

> **See Also**
>
> To learn how to test AI-generated code effectively, see Chapter 6: Writing and automating tests with GitHub Copilot.

Copilot code quality review checklist

Use the following checklist when reviewing Copilot-generated code. It helps you move beyond surface-level "does it run?" and into professional, high-quality practices.

Check	What to look for	Sample Prompts
Functional correctness	Does the code work with valid and edge inputs?	# test with edge cases like empty list or negative numbers
Input validation	Are all arguments type-checked and validated?	# validate input values and types
Readability	Are variables and functions clearly named? Is logic broken into readable parts?	# rewrite function with clearer variable names
Structure and layout	Are unrelated blocks separated? Is it easy to follow the flow?	# refactor to improve structure and readability
Performance	Does the code scale well with larger inputs? Are there any unnecessary operations?	# rewrite for performance on large data
Redundancy	Is any logic repeated? Can it be turned into a helper function?	# eliminate duplicate logic using helper function
Security and safety	Are any unsafe functions, like eval(), or inputs used without sanitization?	# check for unsafe operations
Maintainability	Is this function flexible and modular enough to be reused or extended later?	# make this function more modular

Check	What to look for	Sample Prompts
Testability	Can this logic be tested easily? Would you write unit tests for it?	# add unit tests for edge cases
Intent clarity	Can someone reading this understand the “why” without extra comments?	# rewrite code so intent is clearer

You can use Copilot’s strength to your advantage by turning this checklist into prompts.

You can guide GitHub Copilot toward higher-quality output by using targeted prompts for specific review goals. The table below lists common quality-control objectives and the corresponding advanced prompts you can use to achieve them.

Goal	Advanced prompt
Check boundary conditions	# write test cases including edge inputs
Find subtle bugs	# verify correctness with unexpected inputs
Enforce naming conventions	# rename variables using descriptive names
Strengthen defensive programming	# ensure code fails gracefully for invalid input
Promote clarity over cleverness	# rewrite for clarity, avoid compact one-liners

Prompt-as-you-review workflow

To elevate your workflow, talk to Copilot while reviewing its code, as a mentor would do:

- "Good start, but this needs better naming."
- "Let's rewrite this for performance."
- "Handle that possible failure case."
- "Refactor this into a helper function."

These types of comments make excellent inline prompts that Copilot can respond to intelligently.

Skills review

In this chapter, you learned how to:

- Apply advanced prompting strategies to guide GitHub Copilot more effectively.
- Automate repetitive coding tasks using AI-generated patterns and reusable logic.
- Refactor inefficient or legacy code into cleaner, more maintainable solutions.
- Implement clean coding practices by improving naming, structure, and consistency.
- Improve code readability and structure through helper functions, formatting, and simplified logic.
- Evaluate Copilot suggestions for quality, performance, correctness, and safety.

Practice tasks

No practice files are necessary to complete the practice tasks in this chapter. Follow the steps below to reinforce the concepts and tools introduced in each section.

Apply advanced prompting to guide GitHub Copilot

Practice using advanced prompting strategies with Copilot:

1. Create a new Python file.
2. In the file, type the following comment:

```
# create a function to check if a number is prime using a
```

3. Accept Copilot's suggestion and then edit the comment to include "return True if prime, else False" and observe how Copilot adapts.
4. Try a different prompt using an example of a pattern:

```
# input: [3, 2, 1], output: [1, 2, 3]
# write a function to sort a list
```

5. Reflect on which prompt or prompts generated the most accurate or readable results.

Automate repetitive tasks using AI

Use Copilot to automate repetitive tasks using Copilot:

1. Add a comment in a new Python file:

```
# generate a class for a Book with title, author, and yea
```

2. Use Copilot to complete the class definition.
3. Next, prompt Copilot with:

```
# remove duplicate items from a list
```

4. Observe the logic. Modify the prompt to focus on performance:

```
# remove duplicates from a list using set
```

5. Try using a comment like:

```
# write unit test cases for a login function
```

6. Review the automation for potential repetitive test code.

Refactor inefficient or legacy code

Use Copilot to refactor older or inefficient Python code:

1. Type the following legacy-style code into a file:

```
def calculate(numbers):
result = []
   for n in numbers:
       result.append(n * 2)
   return result
```

2. Above it, add this comment:

```
# refactor using list comprehension
```

3. Accept Copilot’s suggestion.
4. Now try this:

```
# simplify nested if-else
```

5. Test the results with a function you’ve already used.
6. Compare the output you get now with the output you got from the function before and evaluate whether it’s clearer or more concise.

Implement clean coding practices

Apply clean coding habits with Copilot:

1. Type the following function in your file:

```
def fx(x):
    return x + 10
```

2. Add this comment above it:

```
# improve naming and structure
```

3. Accept Copilot’s suggested changes.
4. Now try this:

```
# add docstring and use descriptive variable names
```

5. Review how Copilot helps make code clean and professional.

Improve code readability and structure

Use Copilot to improve code readability and logical structure:

1. Type the following:

```
def process(data):
  for x in data:
      if x != "":
    print(x.strip())
```

2. Add this comment above the function:

```
# break into smaller functions and improve readability
```

3. Accept Copilot's suggestion and consider whether it's easier to read.
4. Now add this comment:

```
# rewrite with comments and better formatting
```

5. Review how the code structure improves.

Evaluate AI suggestions for quality

Critically evaluate and test Copilot's code output:

1. Prompt Copilot with:

```
# divide two numbers
```

2. Accept the function and test it with 0 as the divisor.
3. Add this prompt:

```
# improve this with error handling
```

4. Now try this prompt:

```
# write tests for the division function
```

5. Run the tests to confirm their behavior.
6. Consider how Copilot handled edge cases and whether you would ship this code.

5

Debugging and troubleshooting code with Copilot

In this chapter

- Identify common code issues with GitHub Copilot
- Use Copilot to suggest and apply bug fixes
- Improve error handling by using AI-assisted recommendations
- Optimize code performance through Copilot-driven insights

Practice files

You will need to use the practice files provided with this chapter to complete the practice tasks.

Debugging and troubleshooting are essential skills in software development, often demanding a significant amount of time and effort. With GitHub Copilot, developers can accelerate this process using AI-powered suggestions for identifying bugs, proposing fixes, improving error messages, and optimizing performance. Debugging is a vital part of writing reliable and high-performing code. In this chapter, you will learn how to leverage GitHub Copilot to identify and resolve bugs, improve error handling, and optimize the performance of your code.

You'll begin by learning how to identify common code issues with GitHub Copilot, including syntax errors, logic flaws, and runtime exceptions. Then you'll explore how to streamline your debugging workflow by using Copilot to suggest and apply bug fixes.

Next, you'll look at how to improve error handling by using AI-assisted recommendations, which will help you make your applications more robust and maintainable. Finally, you'll discover how to optimize code performance through Copilot-driven insights, using suggestions that refactor inefficient logic and boost execution efficiency.

By the end of this chapter, you'll have practical knowledge of how GitHub Copilot can assist with key debugging tasks, making your development process faster and more effective.

Identify common code issues with GitHub Copilot

Debugging often begins with identifying the source of an error or unintended behavior. GitHub Copilot can play a crucial role in accelerating this process by proactively analyzing your code and suggesting possible problem areas. In this section, you'll explore how to use Copilot to detect syntax errors, logic flaws, runtime exceptions, and common code smells.

GitHub Copilot doesn't replace traditional debugging tools, but it enhances your ability to identify issues faster by analyzing code context and generating suggestions in real time.

> **Important**
>
> If Copilot repeatedly suggests incorrect or outdated code, it may be due to poorly written prompts or lack of project context. By providing descriptive comments, using relevant file names, and ensuring that the surrounding code context is clear, you can help Copilot more accurately detect issues.

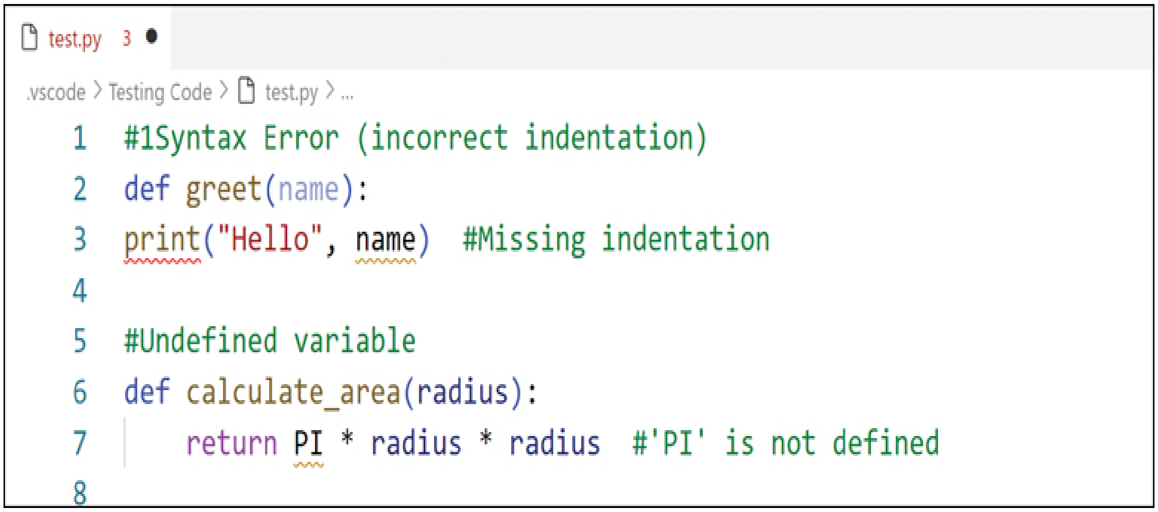
Common code issues Copilot can help identify

Before jumping into debugging or refactoring, it's important to recognize the types of issues that typically occur in code. GitHub Copilot has been trained on a vast amount of real-world code and patterns, and it can spot and flag many of the common code issues early in development. From syntax slips to inefficient logic, Copilot can highlight potential problems and offer real-time suggestions as you type—often before you run your code or hit a runtime error.

GitHub Copilot can assist in detecting a wide variety of code problems, including these:

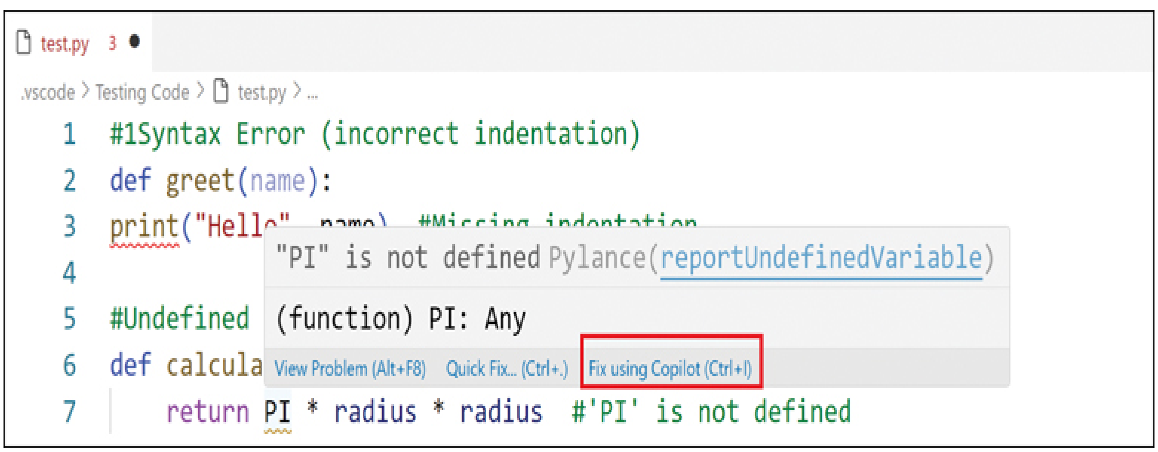
- Syntax errors (e.g., missing semicolons, incorrect indentation)
- Undefined variables or functions
- Incorrect data type usage
- Unreachable code
- Infinite loops
- Null reference or pointer exceptions
- Improper API usage
- Poor performance patterns (e.g., nested loops, inefficient queries)

The following image demonstrates common beginner mistakes: a syntax error caused by missing indentation and an undefined variable.



Code error example in Python, highlighting common beginner mistakes such as missing indentation in a function and using an undefined variable

The following image shows how GitHub Copilot can resolve the undefined variable by suggesting a fix directly inside Visual Studio Code's error tooltip.



GitHub Copilot fix suggestion prompt, demonstrating Copilot's ability to assist in resolving an undefined variable

Use Copilot with a debugging checklist

To systematically identify issues, you can use the following Copilot debugging checklist. By using this checklist, you can ensure that you're not overlooking common error types and make your debugging workflow more efficient.

Copilot Debugging Checklist

- Is the syntax correct (including brackets, colons, and semicolons)?
- Are all variables and functions declared before they are used?
- Are data types being used consistently?
- Are there any hardcoded values that should be dynamic?
- Are edge cases handled properly?
- Are third-party API responses validated?
- Are loops and conditionals optimized and terminating correctly?
- Are all exceptions handled?

Tip

Use GitHub Copilot alongside your IDE's built-in diagnostics and extensions (such as the VS Code Problems tab or JetBrains inspections). This complementary approach can help you quickly spot syntax errors, deprecated API usage, and logic flaws.

Strategy: Using the 5 Whys method for root cause analysis

Fixing a bug is one thing; understanding why it happened is another. When GitHub Copilot highlights an issue or suggests a fix, it's an opportunity to dig deeper and improve your code's overall quality and design. One effective technique for doing so is the "5 Whys" method, which is a simple but powerful approach to uncovering the root cause of a problem by asking "Why?" repeatedly—typically five times.

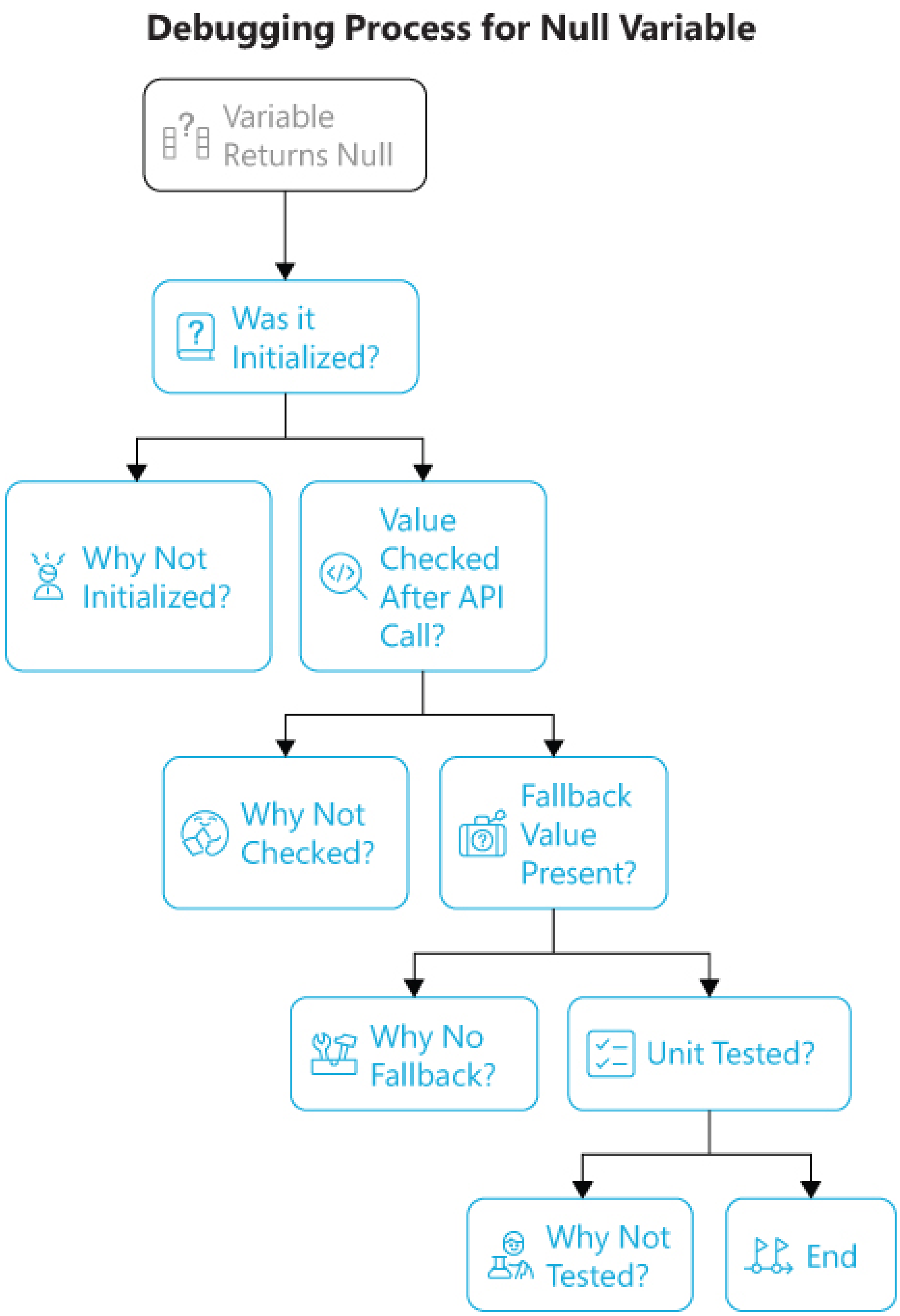
The 5 Whys method helps developers move beyond surface-level fixes to identify underlying issues such as poor validation, missing tests, and design flaws. You can use it alongside Copilot's suggestions to build not just working code but resilient and well-thought-out solutions.

When Copilot suggests an issue in code, one way to drill down to the root cause is to use the 5 Whys technique. When Copilot's suggestion highlights a problem in your code, you can use the **5 Whys technique** to drill down to the root cause. For example, if Copilot shows that a variable is returning null, you might ask the following "Why?" questions:

- Why did the variable return null?
- Why was it not initialized before the function call?
- Why wasn't the value checked after the API call?
- Why is there no fallback value?
- Why was that part of the logic not unit tested?

See Also

To see how Copilot adapts across different coding tasks, check Chapter 8: Using AI-powered development workflows in real-world scenarios.



Flowchart for debugging by applying the 5 Whys method to trace the root cause of a variable returning null from initialization through unit testing

This approach, supported by Copilot's contextual suggestions, helps you not only fix the problem but understand and correct underlying design flaws.

Tip

While you work through the 5 Whys on your own, you can also ask Copilot directly to suggest possible causes or fixes. Use its response as input but still apply the 5 Whys technique to confirm the true root cause.

```
Why does this function return undefined?
```

or:

```
What's wrong with this loop condition?
```

and analyze the response with your own debugging strategy.

How Copilot detects issues in real time

GitHub Copilot provides inline suggestions based on the current code context. It continuously reads and interprets your comments and partially written code to:

- Highlight expressions that are likely incorrect.
- Suggest fixes to runtime or logic issues.
- Recommend better design patterns or idioms.

Make sure to pause and review Copilot suggestions critically before applying them. AI is only as effective as the developer guiding it.

Best practices when identifying issues with Copilot

While GitHub Copilot offers powerful assistance in spotting potential code issues, its effectiveness greatly depends on how you interact with it. By following established best practices, you can improve the relevance of Copilot's suggestions, minimize false positives, and ensure that code quality remains high. These are some actionable techniques you can use to guide Copilot more effectively, reduce debugging time, and maintain control over AI-assisted development:

- Comment your intent: Write clear inline comments to help Copilot generate relevant suggestions.
- Use Copilot Labs (if enabled): It provides more advanced analysis, especially for code transformation or explanation.
- Split large functions: Copilot works better with small, focused functions.
- Cross-verify suggestions: Always test AI-generated code in your actual runtime environment.

Use Copilot to suggest and apply bug fixes

Even the most experienced developers encounter bugs, from runtime errors, to logic flaws, to misused APIs. Traditionally, identifying and fixing these issues has required manual inspection, trial and error, and a deep understanding of the codebase. GitHub Copilot introduces a powerful new way to handle bugs more quickly and intelligently.

By understanding the context of your code, Copilot can analyze logic, recognize common patterns, and suggest intelligent fixes to bugs that may not be immediately obvious to a developer. Copilot works within your IDE as you code, making real-time recommendations to prevent or correct errors on the spot.

In this section, you'll learn how to use Copilot to identify potential bugs in your code, prompt Copilot for accurate and targeted bug fixes, apply and test those suggestions effectively, and leverage Copilot as a collaborative debugging assistant.

Copilot doesn’t replace human oversight, but when used thoughtfully, it acts as an ever-present pair programmer with deep knowledge of common coding mistakes and best practices for resolving issues.

Understand Copilot’s role in fixing bugs

GitHub Copilot can go beyond just generating new code; it can help you detect bugs in your existing code, suggest fixes, and even apply corrections. When used effectively, it is a valuable partner in debugging, significantly reducing the time you need to spend tracing issues manually. Copilot analyzes context and prior code blocks to identify inconsistencies, missing logic, and misused patterns. It can even infer what a function is supposed to do and highlight potential logic flaws.

Whether you're dealing with a null reference, off-by-one errors, or API misuse, Copilot's real-time code understanding and suggestion engine can help streamline your debugging workflow.

Checklist: Using Copilot to fix bugs

When using GitHub Copilot to resolve bugs, it's important to approach the process with structure and clarity. While Copilot can offer quick fixes based on context, the quality and relevance of its suggestions depend heavily on how you guide it. This checklist will help you use Copilot more effectively when identifying and fixing bugs in your code.

From writing clear prompts to testing the fix and validating edge cases, the steps below ensure that you're not only correcting errors but reinforcing best practices in debugging and development.

Use the following checklist to ensure that you're leveraging Copilot effectively when trying to fix bugs:

- Enable Copilot in your IDE (VS Code, JetBrains, etc.).
- Write clear comments describing the bug or what the fix should do.
- Select problematic code blocks and prompt Copilot for suggestions.

- Review the suggestion for logical consistency and test coverage.
- Test the code using unit tests or integration tests after applying a fix.
- Manually validate edge cases that may not be obvious to Copilot.
- Check performance or side effects introduced by the fix.
- Commit with a meaningful message detailing the fix.

Best practices for applying Copilot bug fix suggestions

When using Copilot to help fix bugs, it's important to follow structured practices that ensure accuracy, maintainability, and team clarity. The following best practices will help you apply Copilot's suggestions effectively.

Use context-rich prompts

When prompting Copilot, include relevant error messages, indicate what the expected behavior is, and specify what is currently going wrong.

Prompt:

```
# Fix the IndexError caused when accessing the last item in a
```

Apply fixes incrementally

Apply fixes one bug at a time and test after each suggestion. Avoid applying multiple changes at once, which could lead to compounding errors.

Cross-reference with logs or tests

Use your error logs, stack traces, or test case failures as inputs for Copilot. Paste logs in a comment and let Copilot infer the likely bug.

Document fixes as you go

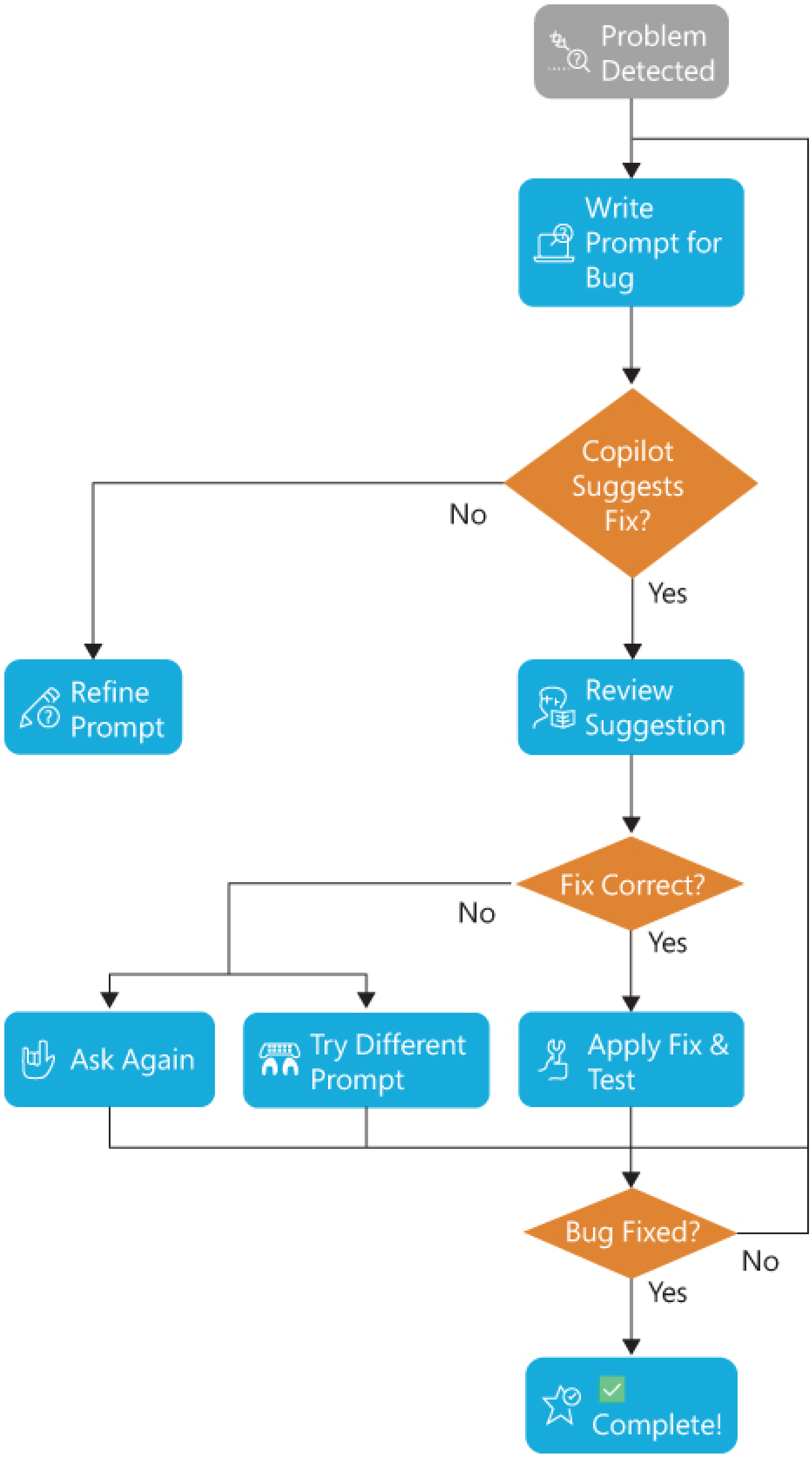
Use inline comments to describe Copilot's suggestions, especially in production or team environments.

Prompt to fix a bug:

```
# Fix the bug causing incorrect total calculation
```

Copilot-suggested code:

```
def calculate_total(prices):
    total = 0
    for i in range(1, len(prices)):  # Starts from index 1, skips
        total += prices[i]
    return total.
```


Problem Detected
Write Prompt for Bug
Copilot Suggests Fix?
No
Yes
Refine Prompt
Review Suggestion
Fix Correct?
No
Yes
Ask Again
Try Different Prompt
Apply Fix & Test
Bug Fixed?
No
Yes
Complete!

GitHub Copilot bug resolution workflow, outlining the iterative process from writing prompts for detected bugs to refining suggestions and applying tested fixes using Copilot

Use Copilot with test failures

If a test case fails, paste the test function and failure message as a comment above the code in question. Copilot can often:

- Diagnose the failure.
- Suggest a fix.
- Recommend test changes.

This works particularly well with unit testing frameworks like pytest, unittest, Jest, or Mocha.

Important

Copilot does not run or validate the code it suggests. Always review and test suggestions thoroughly. Blindly accepting fixes can lead to more subtle and harder-to-detect bugs in production.

Tip

Before asking Copilot for a bug fix, try refactoring the code for clarity (such as by breaking down long functions into smaller ones). This helps Copilot better understand the intent and generate more accurate fixes.

See Also

You will learn how to generate unit and integration tests to ensure that bugs are fixed and do not reoccur in Chapter 6: Writing and automating tests with GitHub Copilot.

Interactive troubleshooting flow: A step-by-step guide

Let’s look at a flow that is designed to simulate a developer's thought process while using GitHub Copilot to resolve bugs in code. This flow is easy for beginners or junior developers—to understand the logic behind using Copilot for debugging.

Use the following flow to guide your debugging interactions with Copilot.

Identify a problem

You notice that something isn’t working as expected—maybe the output is incorrect or an error is thrown.

Ask: Is this a bug or just incomplete logic?

Describe the issue with a prompt

Write a clear, concise comment above the problematic code block that tells Copilot what’s wrong.

Prompt:

```
# The loop skips the first item, fix this
```

Review Copilot’s suggestion

Watch for Copilot’s inline or multi-line suggestion.

Ask:

- Does it fix the bug?

- Is the logic clear?
- Are any side effects introduced?

Apply the fix if it makes sense

If the suggestion looks correct, apply it and test the code. If the suggestion does not look correct, rephrase the prompt or manually refine the logic.

Test for correctness

Validate with both expected and edge-case inputs. Run existing unit tests or write new ones to confirm that the fix works reliably.

Repeat as needed

If the issue persists, loop back to rephrase your prompt or analyze other parts of the code. Sometimes bugs are symptoms of deeper logic flaws.

Improve error handling by using AI-assisted recommendations

Robust error handling is a key element of building reliable, user-friendly applications. Unhandled exceptions can crash programs, corrupt data, or cause confusing behavior for end users. GitHub Copilot can assist developers by suggesting structured error handling patterns, such as try-except blocks, validation checks, logging mechanisms, and fallback logic.

This subchapter will show how to use Copilot to identify error-prone code, generate error-handling constructs, and improve resilience through AI-suggested best practices.

Use Copilot to insert try-except or try-catch blocks

One of the most common use cases for Copilot in error handling is generating the appropriate exception-catching structure based on the context of your code.

Example

```
def divide (a, b):
    return a / b
```

```
Type the below prompt # Fix this to handle divide by zero
```

Copilot Suggestion:

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return "Cannot divide by zero"
```

Checklist: Using Copilot for better error handling

When working with potentially unstable or unpredictable code, error handling is essential for building resilient applications. GitHub Copilot can suggest structured error-handling blocks, but to use these suggestions effectively, it’s important to follow a disciplined approach. The following checklist will help ensure that you're not only using Copilot's suggestions correctly but enhancing the reliability, readability, and user-friendliness of your code.

Use this checklist as a practical guide every time you prompt Copilot to handle exceptions, validate inputs, or apply fallback logic:

1. Add a prompt comment above the risky code block.
2. Watch for Copilot’s inline or multi-line try-except suggestion.

3. Confirm that the exception class is appropriate (e.g., FileNotFoundError, TypeError).
4. Add logging or fallback logic, if needed.
5. Test the block with both success and failure conditions.
6. Ensure that user-facing error messages are clear and useful.

Enhance validation logic and input checking

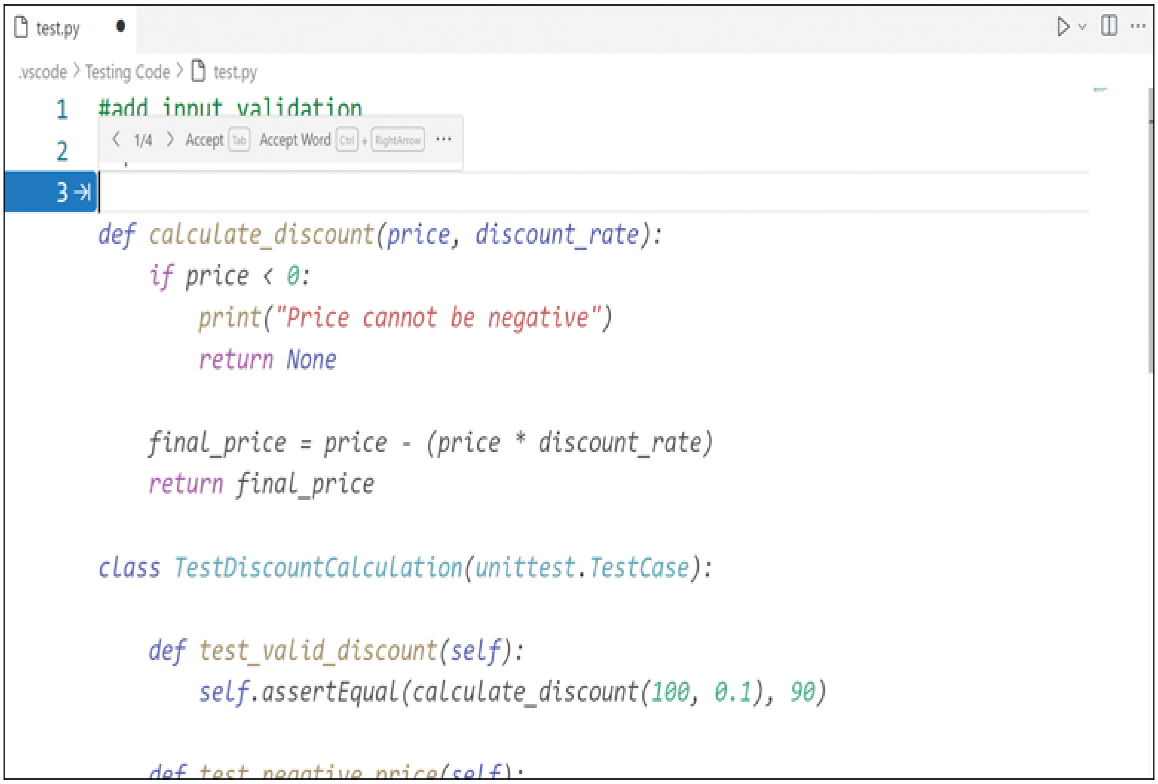
In many applications, improper user inputs or unexpected data can trigger exceptions. Copilot can suggest input validation code, such as type checks, null checks, or boundary constraints.

Prompt:

```
#Add input validation
```

Copilot-suggested code:

```
def process_data(data):
    if not isinstance(data, str):
        return "Invalid input"
    return data.upper()
```



GitHub Copilot suggestion in VS Code while writing a discount calculation function and related unit tests in Python

Use AI to handle edge cases and provide fallback logic

Copilot can propose logic to handle edge cases or alternative return paths. It can help prevent app crashes and provide a better user experience.

Prompt:

```
# Ensure the config file exists before loading
```

```
with open("config.json", "r") as f:
    config = json.load(f)
```

Copilot-suggested code:

```
import os
import json
if os.path.exists("config.json"):
    with open("config.json", "r") as f:
        config = json.load(f)
else:
    config = {}
```

Important

Copilot can suggest general exception handling patterns, but it may not always recommend the most specific or secure approach. Avoid using overly broad exception types (such as except: in Python) unless truly necessary.

Tip

Prompt Copilot with specific exception names or behaviors:

```
# Handle IndexError when accessing list
```

or:

```
# Add fallback value if key not in dictionary
```

Being specific increases the relevance of Copilot's suggestions.

See Also

For more on handling edge cases and improving code quality, see Chapter 9: Avoiding common pitfalls with GitHub Copilot.

Optimize code performance through Copilot-driven insights

Performance issues often creep into applications through inefficient loops, redundant computations, or poor use of data structures. Identifying and fixing these issues manually can be time-consuming. GitHub Copilot helps accelerate this process by suggesting optimized algorithms, refactoring costly operations, and recommending more efficient language constructs.

In this section, you'll learn how to use Copilot to identify slow or wasteful code and rewrite it for better speed, readability, and maintainability—without sacrificing accuracy.

Recognize inefficient code patterns

GitHub Copilot can identify common performance pitfalls like:

- Nested loops where a hash map would be faster.
- Redundant operations in loops.
- Slow sorting or searching techniques.
- Unnecessary file or API calls.

Prompt for redundant loop operation:

```
# Calculate total price (unoptimized)
```

Copilot-suggested code

```
def get_total(cart):
    total = 0
    for item in cart:
        total = total + item["price"]
    return total
```

Suggested Prompt

```
# Copilot refactors to use sum and a generator expression
```

Copilot-suggested code:

```
def get_total(cart):
    return sum(item["price"] for item in cart)
```

Checklist: Optimizing code with Copilot suggestions

When improving code performance, it’s easy to overlook small inefficiencies that accumulate into major bottlenecks over time. GitHub Copilot can be a powerful assistant in refactoring such areas by suggesting faster, cleaner, and more Pythonic solutions. However, to make the most of these suggestions, you need to use a systematic approach.

The following checklist will guide you through key steps to follow when prompting Copilot for performance improvements. Use it as a quick reference when reviewing loops, recursive functions, sorting mechanisms, and other frequently executed code blocks:

1. Identify slow or frequently called code blocks.
2. Add prompts like # optimize this or # rewrite for performance.
3. Look for Copilot suggestions that simplify loops or use built-in functions.
4. Test for correctness and speed improvements.
5. Check for memory-efficient alternatives when working with large data sets.
6. Profile performance before and after applying a fix.

Use built-in functions and language idioms

Copilot often suggests replacing verbose logic with built-in language functions, which are generally faster and more reliable.

Prompt for manual sorting

```
# Sort a list of names
```

```
for i in range(len(names)):
    for j in range(i+1, len(names)):
        if names[i] > names[j]:
            names[i], names[j] = names[j], names[i]
```

Copilot-suggested code

```
names.sort()
```

Optimize loops and recursive logic

Performance bottlenecks often lie in loops and recursion, which Copilot can help rewrite into more efficient alternatives by using memoization or comprehensions.

Prompt for inefficient recursion

```
# Naive Fibonacci (slow)
```

Copilot-suggested code:

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

Suggested Prompt

```
# Optimized with memorization
```

Copilot-suggested code:

```
from functools import lru_cache
@lru_cache(maxsize=None)
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

GitHub Copilot can often transform verbose or inefficient code into concise, optimized alternatives. For example, it can replace manual sum calculations with the built-in sum() function.

The following table compares several common coding patterns with optimized versions suggested by Copilot.

Scenario	Before	After (Optimized by Copilot)
Manual sum calculation	for item in items: total += item	sum(items)
Manual sorting loop	Nested for loops	list.sort() or sorted()
Repeated computation in loop	calc(x) called multiple times in a loop	Variable to store the result for reuse
Naive recursion	fib(n-1) + fib(n-2) without cache	@lru_cache for memoization

Scenario	Before	After (Optimized by Copilot)
Filter using if and append	for x in data: if condition: result.append(x)	result = [x for x in data if condition]

Important

Copilot does not always measure the performance of what it generates. You are still responsible for benchmarking, profiling, and choosing the best solution based on your data and frequency of execution.

Tip

When working with large datasets, try prompts like:

```
# Optimize this loop for speed
```

or:

```
# Use a memory-efficient approach here
```

These prompts guide Copilot to offer better-structured solutions.

See Also

Explore how Copilot fits into performance-aware workflows in Chapter 8: Using AI-powered development workflows in real-world scenarios.

Skills review

In this chapter, you learned how to:

- Use GitHub Copilot to identify common code issues, such as undefined variables, unreachable code, and poor performance patterns.
- Use clear and contextual prompts to get Copilot to suggest and apply bug fixes.
- Improve error handling by generating structured exception management, validation logic, and fallback mechanisms.
- Optimize code performance by refactoring inefficient loops, recursion, and redundant operations based on Copilot suggestions.
- Apply structured debugging workflows by using decision-making strategies like the 5 Whys method and interactive troubleshooting flows.
- Review and validate Copilot's suggestions for correctness, performance impact, and logical consistency before implementing them.

Practice tasks

Before you can complete these tasks, you must copy the book's practice files to your computer. The practice files for these tasks are in the \Ch05 folder .

The Introduction to this book includes a complete list of practice files and download instructions.

Identify common code issues with GitHub Copilot

The file identify_issues.py that is provided with the practice files contains code snippets with common issues like undefined variables, skipped loops, and logic errors. Use it as you perform the following tasks:

1. In your IDE (such as Visual Studio Code), open the file identify_issues.py.
2. Review the function calculate_total and use a comment prompt to highlight the loop issue:

```
# The loop skips the first item, fix this
```

3. Use Copilot's inline suggestions to correct the loop and verify that all values are included.
4. Locate the function get_username(config), which uses a hard-coded dictionary key, and add this prompt before it:

```
# Add a safe way to access dictionary keys
```

5. Accept Copilot's suggestion to prevent a KeyError.
6. Review the error-handling flow in the file and use Copilot prompts to identify any missing checks or undefined values.

Use Copilot to suggest and apply bug fixes

The file buggy_functions.py that is provided with the practice files contains small functions with bugs like off-by-one errors, missing return values, and incorrect calculations. Use it as you perform the following tasks:

1. In your IDE (such as Visual Studio Code), open the file buggy_functions.py.
2. Review the function divide(a, b), which may cause a runtime error when dividing by zero. Add this prompt above it:

```
# Handle division by zero error
```

3. Accept Copilot's try-except block suggestion to handle the error.
4. Locate the function calculate_average(scores), which can throw a ZeroDivisionError on an empty list. Add this prompt above it:

```
# Add check for empty list before dividing
```

5. Review the function is_positive(number), which is missing a return statement for certain cases. Add this prompt above it:

```
# Ensure this function returns a value for all cases
```

6. Test the Copilot-generated fixes and use sample inputs to verify that the code behaves as expected.

Improve error handling by using AI-assisted recommendations

The file error_handling_cases.py that is provided with the practice files includes examples of functions that are missing validation, that include risky operations, and that use overly broad exception handling. Use it as you perform the following tasks:

1. Open the error_handling_cases.py file in your IDE.
2. Locate the function load_user_profile(filepath), which assumes that the error_handling_cases.py file exists. Add the following prompt:

```
# Add file existence check before opening
```

3. Review Copilot's suggestion and accept a solution that uses os.path.exists() or a try-except block.

4. Move to the function parse_data(data) and add this prompt above it:

```
# Add input validation to ensure data is a string
```

5. Find the function risky_operation() and add this prompt above it to have Copilot revise the overly broad exception handling:

```
# Replace with more specific exception handling
```

6. Test the new logic with inputs that simulate invalid file paths and incorrect data types.

Optimize code performance through Copilot-driven insights

The file performance_review.py that is provided with the practice files contains inefficient implementations involving loops, recursion, and verbose logic. Use it as you perform the following tasks:

1. Open the performance_review.py file in your IDE.

2. In the function sum_prices(cart), locate the loop that is used to sum item prices and add this prompt above it:

```
# Optimize this loop using built-in functions
```

3. Accept Copilot's suggestion to use a generator expression with sum().

4. Navigate to the fib(n) function and add this prompt above it to enhance performance:

```
# Optimize this recursive function
```

5. Accept the @lru_cache optimization to improve recursive call efficiency.
6. Locate the function compute_squares(numbers) and add this prompt above it:

```
# Use list comprehension to simplify and optimize
```

7. Test the updated functions with inputs of various sizes and compare the performance before and after the fixes.

6

Writing and automating tests with GitHub Copilot

In this chapter

- Generate unit tests and integration tests using GitHub Copilot
- Automate test case creation to reduce repetitive coding
- Work with popular testing frameworks
- Write clear and structured test prompts for Copilot to follow
- Apply best practices to improve the reliability of AI-generated tests

Practice files

There are no practice files for this chapter.

Testing is an essential part of building reliable software, ensuring that your code behaves as expected and remains stable as it evolves. However, writing tests can be repetitive and time-consuming, especially when you are covering multiple functions, edge cases, and integration points. GitHub Copilot changes this dynamic by acting as an AI-powered assistant that can quickly generate unit and integration tests, suggest parameterized cases, and adapt to popular frameworks, all from natural language prompts. In this chapter, you'll learn how to combine Copilot's speed with best practices for writing meaningful, maintainable, and comprehensive tests that improve software quality without slowing down development.

Generate unit tests and integration tests using GitHub Copilot

Testing is not just a final step; it's an ongoing practice that should be integrated into your development process. Whether you're building a small utility or a large application, it's important to write tests to ensure that your code works as expected and continues to work even as it changes.

But writing tests manually isn't usually the most exciting part of development. It often involves repeating similar patterns to check inputs and outputs, handle edge cases, and verify that components work together. This is exactly where GitHub Copilot shines. With a simple comment or prompt, you can ask Copilot to help generate unit tests for individual functions and integration tests for combined parts of your system. Instead of spending time writing boilerplate code, you get a head start, and you stay in flow.

What are unit tests?

A unit test is a small, focused test that checks a specific part of your code—typically a function or method. For example, if you have a function that calculates the average of a list, a unit test could feed it sample input and check that the output is correct.

Unit tests help answer questions like these:

- Does this function return the expected result?
- Does it handle empty inputs or invalid values properly?
- Will it break if the input changes?

Using Copilot to write a unit test

To see how Copilot can quickly generate useful tests for a simple function, try these steps:

1. Open Visual Studio Code (or any other IDE that supports GitHub Copilot).
2. Create a new Python file (for example, average_calculator.py).
3. Ensure that GitHub Copilot is installed and enabled in your IDE.
4. In the new file, type or paste the following function:

```
def calculate_average(scores):
           if not scores:
                return 0
           return sum(scores) / len(scores)
```

To test this function manually, you could import unittest or pytest, write a few test cases, and run them. But with Copilot, you can skip the setup and focus on what matters.

Prompt:

Now add the following comment:

```
# Write unit tests for the calculate_average function
```

Copilot-suggested code:

Copilot might suggest the following unit test function:

```
def test_calculate_average():
    assert calculate_average([90, 80, 70]) == 80
    assert calculate_average([100, 90, 80, 70, 60]) == 80
    assert calculate_average([]) == 0
    assert calculate_average([50]) == 50
```

```python
def calculate_average(scores):
    if not scores:
        return 0
    return sum(scores) / len(scores)
# Write unit tests for the calculate_average function
def test_calculate_average():
    assert calculate_average([90, 80, 70]) == 80
    assert calculate_average([100, 90, 80, 70, 60]) == 80
    assert calculate_average([]) == 0
    assert calculate_average([50]) == 50
```

GitHub Copilot generation of a Python test function using assert statements for the calculate_average function after reading a plain English comment prompt

This is a basic test function with multiple test cases. It covers regular input, an edge case (an empty list), and a single-element list. You can expand from here with more specific cases, such as handling floats or handling invalid data.

Tip

Copilot works best when you guide it with comments. Try something like this:

```python
# Add a test for when the list contains negative numbers
```

and Copilot will respond accordingly.

See Also

> The next section shows you how to automate test creation for common coding scenarios, helping you write tests faster with less repetition.

What are integration tests?

An integration test checks how different parts of a system work together. Instead of testing a single function in isolation, an integration test might test an entire process. For example, an integration test might call an API, save the result to a database, and verify the saved data.

Integration tests help answer questions like these:

- Does data flow correctly between components?
- Does the system behave properly when external services are involved?
- Are side effects like file writing or logging handled?

Using Copilot to write an integration test

Let's say you have a Python function like this one:

```
def fetch_user_data(api_url):
    response = requests.get(api_url)
    if response.status_code == 200:
        return response.json()
    return None
```

Prompt:

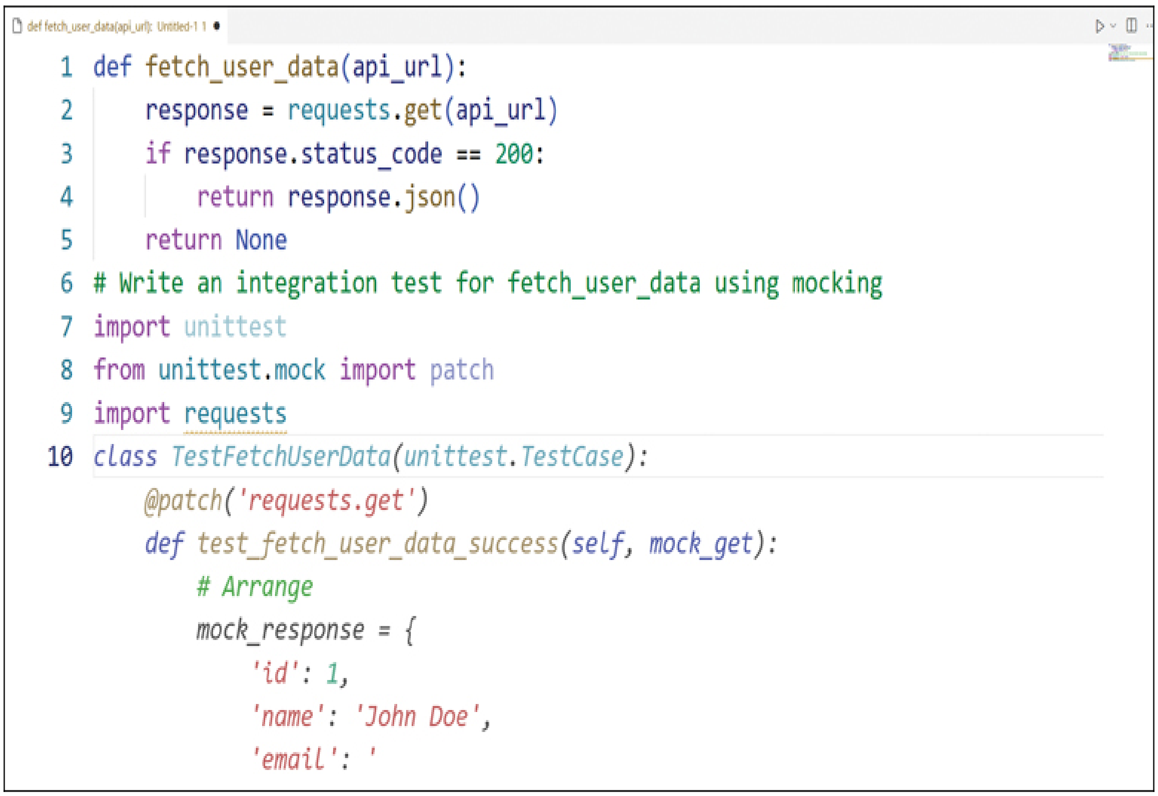
To have Copilot write an integration test, you can use a prompt like this:

```
# Write an integration test for fetch_user_data using mocking
```

Copilot-suggested code:

Copilot might suggest the following integration test:

```
import unittest
from unittest.mock import patch
import requests

class TestFetchUserData(unittest.TestCase):
    @patch('requests.get')
    def test_fetch_user_data_success(self, mock_get):
        # Arrange
        mock_response = {
            'id': 1,
            'name': 'John Doe',
            'email': ''
        }
```



```
def fetch_user_data(api_url):
    response = requests.get(api_url)
    if response.status_code == 200:
        return response.json()
    return None
# Write an integration test for fetch_user_data using mocking
import unittest
from unittest.mock import patch
import requests
class TestFetchUserData(unittest.TestCase):
    @patch('requests.get')
    def test_fetch_user_data_success(self, mock_get):
        # Arrange
        mock_response = {
            'id': 1,
            'name': 'John Doe',
            'email': '
```

Copilot-generated integration test with a unittest class and a test method using @patch

This test simulates an API response without actually making a network call, which is perfect for fast, reliable testing.

Important

Even though Copilot can generate integration tests, it doesn't know your full system architecture. Always verify that mock paths, dependencies, and setup code are accurate in your environment.

Languages and frameworks Copilot supports for testing

Copilot works across many languages and test frameworks. The following table shows popular languages and common testing frameworks that GitHub Copilot supports.

Language	Common test frameworks
Python	unittest, pytest
JavaScript	Jest, Mocha, Vitest
Java	JUnit
C#	xUnit, NUnit, MSTest
TypeScript	Jest, ts-jest, Vitest

Tip

When prompting Copilot, mention the framework you're using. For example:

```
// Write a Jest test for the addUser function.
```

Benefits of generating tests with Copilot

Using GitHub Copilot to write tests isn't just about saving time; it's about working smarter. Copilot acts as a helpful assistant that takes care of repetitive tasks, freeing you up to focus on the logic and quality of your code. It helps you spot edge cases, keeps your test coverage consistent, and encourages testing habits without slowing you down. Whether you're new to testing or already experienced, Copilot can make the process feel less like a chore and more like a natural part of your workflow.

Copilot offers a number of benefits related to code testing:

- Saves time: Copilot gives you a working draft, so you don't have to start from scratch.
- Covers more cases: Copilot may suggest edge cases that you hadn't considered.
- Boosts confidence: Copilot tests your code, helping you reduce bugs and improve reliability.
- Supports test-driven development (TDD): You can write a test first and then use Copilot to build the function to pass it.

With GitHub Copilot, testing is a faster, more enjoyable part of development, not a chore.

Automate test case creation to reduce repetitive coding

One of the most common reasons developers delay or skip writing tests is the repetitive nature of the task. Many test cases follow a similar pattern: Arrange the inputs, call a function, and check the output. Doing this manually for every function or feature quickly becomes tedious, especially in large projects.

GitHub Copilot can take on the heavy lifting by automating the creation of test cases. With well-written prompts, Copilot can generate multiple variations of a test method to cover expected outputs, edge cases, and even error conditions. The more descriptive the prompts, the more useful the generated test code.

Use comments to describe multiple test scenarios

Copilot relies heavily on context. If you want to generate several test cases at once, use a clear comment or docstring that outlines each scenario.

Before trying this example prompt,

1. Open Visual Studio Code (or any preferred IDE that supports GitHub Copilot).
2. Create a new Python file (e.g., test_divide_numbers.py).
3. Ensure that GitHub Copilot is enabled in your environment.

Prompt:

Type the following comments to prompt Copilot to generate multiple related test cases:

```
# Write tests for divide_numbers(a, b)
# 1. Divide two positive integers
# 2. Handle division by zero
# 3. Return float result for integer inputs
# 4. Handle negative numbers
```

Copilot-suggested code:

```
import unittest

class TestDivideNumbers(unittest.TestCase):
```

```
    def test_divide_positive_integers(self):
        self.assertEqual(divide_numbers(10, 2), 5)

    def test_divide_by_zero(self):
        self.assertEqual(divide_numbers(10, 0), "Division by zero

    def test_divide_integer_inputs(self):
        self.assertEqual(divide_numbers(10, 4), 2.5)

    def test_divide_negative_numbers(self):
        self.assertEqual(divide_numbers(-10, -2), 5)
```

GitHub Copilot creating well-structured Python unit tests using unittest, based on multiple natural language comments

Automate table-driven tests

Some testing frameworks support table-driven testing, also called parameterized testing, where the same test logic is applied to multiple sets of data. In Python, this is commonly done using the @pytest.mark.parameterize decorator in the pytest framework.

Prompt:

You can use GitHub Copilot to help generate table-driven tests by writing a prompt that describes the function and the types of input you want to test. For example, start with a simple comment like this:

```
# Write parameterized tests for is_even() using different inp
```

Copilot-suggested code:

```
import pytest

def is_even(n):
    """Check if a number is even."""
    return n % 2 == 0

@pytest.mark.parametrize("input, expected", [
    (2, True),
    (3, False),
    (0, True),
    (-2, True),
    (-3, False)
])
def test_is_even(input, expected):
    assert is_even(input) == expected
```

```python
# Write parameterized tests for is_even() using different inputs
import pytest
def is_even(n):
    """Check if a number is even."""
    return n % 2 == 0

@pytest.mark.parametrize("input, expected", [
    (2, True),
    (3, False),
    (0, True),
    (-2, True),
    (-3, False)
])
def test_is_even(input, expected):
    assert is_even(input) == expected
```

GitHub Copilot generating pytest-style parameterized unit tests for the is_even() function, based on a simple natural language prompt

This table-driven test can work on a variety of inputs, including positive numbers, zero, and negative numbers, all within a single, compact function. It's efficient and easy to read, and it encourages clean, scalable test code.

> **Tip**
>
> Use parameterized tests to reduce duplication in your test files. Copilot can generate the list of test cases for you if you provide a clear prompt and a few examples.

Extend existing test files

If you're working in a file that already contains some tests, you can have Copilot detect the test style and generate additional test cases that match the format. This is especially useful when you’re maintaining an older codebase, where adding new test cases manually would be time-consuming. Copilot helps you stay consistent and productive.

Prompt:

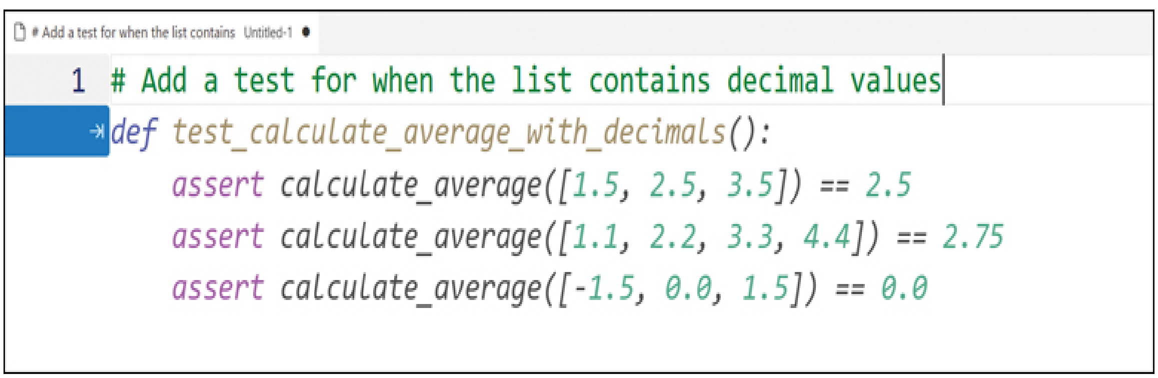
Simply type a new method header or comment:

```
# Add a test for when the list contains decimal values
```

Copilot-suggested code:

Copilot suggests a new method that fits right into the existing class structure:

```
def test_calculate_average_with_decimals():
    assert calculate_average([1.5, 2.5, 3.5]) == 2.5
    assert calculate_average([1.1, 2.2, 3.3, 4.4]) == 2.75
    assert calculate_average([-1.5, 0.0, 1.5]) == 0.0
```



GitHub Copilot auto-generating a new test function for the calculate_average() method, using decimal values

Generate negative and edge case tests

Copilot can help you write tests for the unexpected. For example, you might want to test how your code behaves with invalid inputs or extreme values. By handling edge cases up front, you make your code more robust and prevent potential runtime issues.

Prompt:

You can prompt Copilot directly:

```
# Write tests to handle invalid inputs for the calculate_disc
```

Copilot-suggested code:

The output might include:

```
def test_negative_price(self):
    with self.assertRaises(ValueError):
        calculate_discount(-100, 0.1)

def test_discount_greater_than_one(self):
    with self.assertRaises(ValueError):
        calculate_discount(100, 1.5)
```

Tip

When testing edge cases, include both expected failures and graceful fallbacks in your prompt (e.g., raise ValueError or return None).

Copy and paste refactoring in a prompt

If you have a set of repeated tests across different modules, you can refactor them into prompts and let Copilot generate the same pattern elsewhere. This approach is particularly useful for repetitive backend or database test scenarios.

Prompt:

```
# Write tests for each CRUD operation: create_user, read_user
```

Copilot-suggested code:

As shown in the following figure, Copilot will create a test for each function, using similar structure and naming for all of the tests.

GitHub Copilot auto-generating unit tests using unittest.mock for create_user and read_user operations in response to a structured CRUD testing prompt

Suggested structure for reusable test prompts

Here’s a helpful prompt template you can reuse:

```
# Write test cases for [function_name]
# - Test normal inputs
# - Test edge cases
```

```
# - Test invalid inputs
# - Use [test framework]
```

Tip

Feel free to copy and adapt this structure when you're working across different projects or teams.

When to automate test creation with Copilot

Use Copilot to automate tests when:

- You're working with standard logic and inputs.
- You already have a function and want full test coverage fast.
- You need to expand an existing test suite without writing from scratch.
- You want to reduce repetitive testing patterns across modules.

Copilot won't write perfect tests every time, but it gives you a strong starting point and saves time in every sprint.

See Also

In Chapter 7: Using GitHub Copilot for code reviews and collaboration, you'll see how Copilot can also assist in reviewing your code and test logic, helping you refactor and document your tests as part of a collaborative workflow.

Work with popular testing frameworks

GitHub Copilot isn't tied to just one way of writing tests. It supports many of the popular frameworks developers already use. In Python, the two most widely used testing frameworks are unittest, which comes built into the standard library, and pytest, which is more modern, flexible, and widely adopted in the open-source world.

Copilot works well with both unittest and pytest. You don't need to change your existing testing strategy; just write a prompt, and Copilot will adapt to your framework automatically.

Use Copilot with unittest

If you're using Python's standard library, unittest is likely familiar. Copilot understands this framework and can quickly scaffold test classes, methods, and common assertions with just a few words.

Here's how it typically works: You write a small prompt or comment that describes the function to test, and Copilot generates a unittest.TestCase subclass. Each test becomes a method that starts with test_.

Prompt:

```
# Write unit tests for the add_numbers function using unittes
```

Copilot-suggested code:

```
import unittest
class TestAddNumbers(unittest.TestCase):
    def test_positive_numbers(self):
        self.assertEqual(add_numbers(2, 3), 5)
    def test_zero(self):
        self.assertEqual(add_numbers(0, 0), 0)
    def test_negative_numbers(self):
        self.assertEqual(add_numbers(-2, -3), -5)
```

Tip

When you're working in legacy codebases or teaching environments, unittest is a great fit. Copilot automatically formats its output using correct unittest structure—with no additional configuration needed.

Use Copilot with pytest

Many developers prefer pytest to unittest for its simpler syntax, better readability, and rich ecosystem. It offers the following benefits:

- There is no need to subclass TestCase.
- Tests are written as plain functions.
- Assertions are made using standard Python assert statements.
- Fixtures and decorators like @pytest.mark.parametrize are supported.

Prompt:

```
# Write tests for multiply(a, b) using pytest
```

Copilot-suggested code:

```
def test_multiply_positive_numbers():
    assert multiply(2, 4) == 8

def test_multiply_by_zero():
    assert multiply(5, 0) == 0

def test_multiply_negative_numbers():
    assert multiply(-2, 3) == -6
```

Prompt:

You can also guide Copilot to generate parameterized tests by using pytest:

```
# Use pytest parametrize to test is_palindrome function
```

Copilot-suggested code:

```
import pytest
@pytest.mark.parametrize("text,expected", [
    ("radar", True),
    ("hello", False),
    ("", True),
    ("a", True),
])
def test_is_palindrome(text, expected):
    assert is_palindrome(text) == expected
```

Important

Copilot does not install external libraries. If you use pytest, make sure it's installed in your environment (pip install pytest). Otherwise, test runs will fail—even if Copilot generates correct code.

When to use unittest vs. pytest with Copilot

Both the unittest and pytest frameworks work well with GitHub Copilot, so your choice comes down to project needs and team preferences. The following table compares the use of unittest and pytest with GitHub Copilot, highlighting differences in setup, structure, syntax, and ease of use.

Feature	unittest	pytest
Included in Python	Yes	No (installation needed)
Requires class-based tests	Yes	No
Uses assert methods	Yes (self.assert)	Yes (plain assert)
Supports parameterized tests	Manual	Built-in

Feature	unittest	pytest
Readability and simplicity	Medium	High
Works with Copilot	Yes	Yes

Copilot doesn't prefer one over the other; it simply follows your style. As long as you include a descriptive prompt, Copilot will generate tests that fit the framework you're using.

Use framework-specific prompts to guide Copilot

When you want Copilot to use a particular testing style, be specific in your prompt. Here are some examples:

```
# Write a unittest test case for check_login()
# Use pytest to test the validate_email function
# Create a parameterized test in pytest for calculate_tax()
```

The more guidance you give, the more accurate and useful Copilot's suggestion will be.

Tip

You don't need to know all of the pytest or unittest syntax by heart. Let Copilot fill in the blanks. You can always tweak the code it creates to fit your exact needs.

Copilot supports your preferred testing workflow—whether you use unittest or pytest. It can generate:

- Class-based tests using unittest.TestCase.

- Plain functions using pytest.
- Assertions using either self.assertEqual() or assert.
- Parameterized tests using @pytest.mark.parametrize.

You are in control. Copilot simply helps you move faster and write more comprehensive test code with less effort.

> **See Also**
>
> In Chapter 10: Exploring the future of AI in software development, you'll explore upcoming AI trends in coding, the evolution of GitHub Copilot, and emerging developer roles in an AI-driven era.

Write clear and structured test prompts for Copilot to follow

GitHub Copilot is powerful, but it doesn't read minds. It works best when you give it clear, focused instructions about what you want. The good news? You don't need to learn any special syntax. Copilot responds to natural language prompts that are a lot like the comments you would leave for colleagues who are taking over your testing tasks. If you've ever written a "TODO" in your code, you're halfway there.

In this section, you'll learn how to write effective prompts specifically for generating test cases. Better prompts help Copilot return accurate, readable, and useful results.

Start with a clear intent

Tell Copilot exactly what you want.

Vague prompt:

```
# test
```

Specific prompt:

```
# Write unit tests for the login function using pytest
```

Even more specific prompt:

```
# Write test cases for register_user:
# - valid input
# - missing password
# - invalid email format
```

This kind of prompt, which breaks down the scenarios you expect, gives Copilot context about the function's purpose and the edge cases you care about.

> **Tip**
>
> Treat your comment like a checklist. Copilot often generates a separate test for each individual prompt automatically.

Use structure and test language

Copilot understands terms like these:

- Unit test
- Integration test
- Parameterized test
- Edge case
- Mock API response

- Handle exception

By using this familiar vocabulary in your prompts, you help Copilot generate the right kind of test with the correct syntax.

Prompt:

```
# Write a unit test for fetch_data() that handles a 404 error
```

Copilot can then produce a test that mocks an HTTP response and checks for error handling—without requiring you to describe every detail.

Mention the framework, if needed

If you want your tests in a particular style (such as pytest instead of unittest), say so in your prompt.

Prompt:

```
# Use pytest to test calculate_area with parameterized inputs
```

This prevents Copilot from defaulting to the wrong format and ensures that it uses assert instead of self.assertEqual().

> **Important**
>
> If you don't specify a framework, Copilot may guess based on nearby code. Always mention the framework in new or empty files to avoid inconsistent test structures.

Combine comments and code for better context

Sometimes Copilot performs even better when your function is already written and your prompt appears directly below it.

Prompt:

```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True
```

```
# Write pytest tests for is_prime including edge cases
```

This gives Copilot the context it needs to understand what the function does and what test cases make sense. In this case, it might generate tests for:

is_prime(2) returns True

is_prime(4) returns False

is_prime(0) → False

Add one example to steer the output

If you provide one good test case, Copilot can use it as a guide for style, naming, and logic.

Prompt:

```
def test_add_positive_numbers():
    assert add(3, 4) == 7
```

```
# Write more pytest test cases for add()
```

Copilot-suggested code:

```
def test_add_negative_numbers():
    assert add(-2, -3) == -5

def test_add_zero():
    assert add(0, 5) == 5
```

What to avoid in prompts

Some types of prompts can confuse Copilot or result in poor suggestions. The following table compares ineffective prompts, explains why they cause issues, and shows better alternatives you can use instead.

Bad prompt	Why it's a problem	Better alternative
# test	Too vague	# Write unit tests for convert_currency()
# handle this	No context	# Add test to handle empty input list
# finish the function	Not a test-related prompt	# Write a test for divide() with zero divisor
# function to test stuff	Informal, lacks detail	# Write a pytest test for validate_password()

Suggested structure for effective test prompts

Here's a reliable pattern you can follow for your test prompts:

```
# Write [unit/integration/parameterized] tests for [function
# - [Case 1 description]
# - [Case 2 description]
# - [Edge case, error, or invalid input]
# Use [pytest/unittest]
```

Even partial prompts like this are usually enough to get Copilot started with the right test scaffolding.

> **Tip**
>
> You can reuse this prompt format across projects. Save it in a snippet file or use it as a comment template for teams.

> **See Also**
>
> Explore how Copilot adapts to multiple coding styles and teams in Chapter 7: Using GitHub Copilot for code reviews and collaboration.

Apply best practices to improve the reliability of AI-generated tests

GitHub Copilot can generate test cases with remarkable speed, but speed doesn't always mean accuracy. As with any other assistant, the results you get with Copilot depend on how clearly you communicate and how carefully you review the work Copilot produces.

This section will help you build the habit of reviewing, editing, and improving Copilot-generated tests. Think of it as "code review mode." With a few smart practices, you can make sure your test cases are not just fast but also reliable, meaningful, and trustworthy in your production workflow.

Always review AI-generated test logic

Copilot doesn't always get it right. It might suggest a test that appears valid but makes incorrect assumptions about your function.

Prompt:

```
def calculate_discount(price, percentage):
  return price * (1 - percentage)
```

Copilot-generated code:

```
def test_zero_percentage():
    assert calculate_discount(100, 0) == 0  # This is incorrect
```

This test mistakenly checks whether the final price is zero, when it should actually be 100. The logic error comes from a misunderstanding of what the function is supposed to do.

> **Important**
>
> Always test a test. Don't trust that Copilot's suggestion is correct—even if it looks right. Double-check the input/output logic.

Validate against real use cases

When possible, run Copilot-generated tests against known values or existing documentation. If your function is supposed to return a certain result, compare it with real-world inputs.

Prompt:

```
# calculate_final_grade(scores) → returns weighted average
```

Check whether the tests align with your grading policy or rules. Copilot might suggest valid-looking tests, but without knowing your business rules, it can't guarantee accuracy.

Avoid over-trusting happy paths

Copilot tends to favor the most common use cases—that is, what developers typically write. This means it often skips edge cases such as these:

- Invalid inputs
- Empty inputs
- Unexpected types
- Boundary conditions

Be sure to add these edge cases yourself or explicitly ask for them.

Prompt:

```
# Add tests for invalid score values and empty lists
```

Tip

A good rule of thumb: For every test Copilot suggests, try adding one "failure case," or edge condition, manually—or prompt Copilot to do so.

Be consistent with naming and structure

Even if the test logic is correct, Copilot's style may vary between suggestions. You might see it create code like this in one suggestion:

```
def test_addition():
```

and create code like this in another suggestion:

```
def test_add_positive_numbers():
```

This type of inconsistency would likely confuse future maintainers or your teammates.

To avoid getting inconsistent results like this:

- Define a clear naming pattern and follow it (e.g., test_[function]_[condition]).
- Refactor Copilot's suggestions to follow the same format.
- Use docstrings, if needed, to clarify intent.

Prompt:

```
def test_login_with_valid_credentials():
  """Ensure login() returns True for correct username and pas
```

Use fixtures and setup blocks when needed

Copilot might repeat setup logic across multiple test cases. Here is an example:

```
def test_case1():
    db = connect_to_test_db()
    # ...

def test_case2():
    db = connect_to_test_db()
    # ...
```

In such cases, refactor the code to use a fixture (in pytest) or a setUp() method (in unittest).

Prompt:

```
# pytest example
@pytest.fixture
def db():
  return connect_to_test_db()
```

Tip

You can prompt Copilot to rewrite your tests using fixtures, like this:

```
# Refactor the tests to use a pytest fixture for the datal
```

Keep tests isolated

Copilot may generate tests that rely on shared global state, static files, or unmocked APIs. This can lead to flaky tests—that is, tests that pass or fail randomly, depending on the environment or order.

Before you accept a Copilot test, ask:

- Does this test work in isolation?
- Does it modify shared data (like a file or database)?
- Does it rely on an actual API call?

If the answer to any of these questions is "yes," consider mocking the call or using a temporary file.

Prompt:

```
# Write a test that mocks the requests.get call in fetch_data
```

Document assumptions in comments or docstrings

AI-generated tests often don't explain why a particular value was used. When in doubt, add a short comment or docstring to clarify the logic. Here is an example:

```
def test_large_input():
    # Ensures the function handles long input strings gracefully
    assert process_input("x" * 10000) == "OK"
```

This makes your tests easier to maintain—especially if a teammate inherits them later.

Don't forget security and edge conditions

Copilot is trained on public code, and while it can suggest good practices, it doesn't automatically follow your app's security, privacy, or compliance rules. Watch for the following:

- Insecure input handling (that is, no validation or escaping)
- Blind trust in API responses
- Use of hardcoded tokens or secrets

: Ineffective Prompt

```
# BAD: Hardcoding credentials in a test
```

```
assert login("admin", "password") == True
```

Copilot-suggested code:

```
assert login("admin", "password") == True
```

Better Prompt

Use environment variables or mock authentication

Important

Never hardcode credentials or sensitive data in your test prompts—and never accept tests that do. Use placeholders or environment variables instead.

Summary of best practices

It is important to keep in mind the essential habits that help you turn Copilot's suggestions into robust, trustworthy tests. The following table summarizes best practices for reviewing, extending, and maintaining AI-generated tests so you can ensure that your test suite remains accurate, readable, and reliable over time.

Best practice	Why it matters
Review test logic carefully.	Prevents silent failures and false positives.
Add missing edge cases and failure cases.	Improves coverage and confidence.
Refactor test naming and structure.	Improves readability and consistency.
Use fixtures and mocks when needed.	Ensures test reliability and isolation.

Best practice	Why it matters
Avoid insecure or unrealistic values.	Maintains test safety and code integrity.

See Also

In Chapter 9: Avoiding common pitfalls with GitHub Copilot, you'll explore how to troubleshoot AI-generated code and avoid common pitfalls.

Skills review

In this chapter, you learned how to:

- Generate unit tests and integration tests using GitHub Copilot by writing clear, targeted prompts.
- Automate repetitive test case creation by using structured comments and parameterized testing techniques.
- Work with the popular Python testing frameworks unittest and pytest to integrate AI-generated tests into your existing workflow.
- Write effective prompts that guide Copilot to generate reliable, well-structured test cases.
- Apply best practices to validate, refactor, and improve the quality of AI-generated tests.

Practice tasks

No practice files are necessary to complete the practice tasks in this chapter. Follow the steps below to reinforce the concepts and tools introduced in each section.

Use an integrated development environment (IDE) such as Visual Studio Code with GitHub Copilot enabled and signed in. The following practice tasks use Python as the language. You can adapt them to other supported languages as needed.

To try the following tasks in Python, begin by creating a new Python file (e.g., tests_example.py) in an open workspace/folder.

> **Tip**
>
> You can try these practice tasks in any programming language supported by GitHub Copilot, such as JavaScript, TypeScript, Java, or C#. Copilot will adapt its suggestions to the language and framework you're using.

Generate unit tests and integration tests using GitHub Copilot

Use Copilot to write both unit tests and integration tests for two Python functions, using unittest and mocking:

1. Create a function, def convert_to_celsius(fahrenheit):, that converts Fahrenheit to Celsius.
2. Below the function, add this comment:

```
# Write unit tests for convert_to_celsius using unittest
```

3. Accept Copilot's suggestions and inspect the generated test logic.
4. Write another new function, fetch_weather_data(url), that makes a placeholder requests.get() call.

5. Add this prompt:

```
# Write an integration test for fetch_weather_data using
```

6. Review how Copilot handles API mocking and adjust the test as needed.

Automate test case creation to reduce repetitive coding

Prompt Copilot and pytest to generate a parameterized test to check palindromes:

1. Define a function, is_palindrome(word), that checks whether a word is a palindrome.
2. Add this comment:

```
# Use pytest parametrize to test is_palindrome with diffe
```

3. Accept the Copilot-generated parameterized test.
4. Add at least one invalid input manually (e.g., None, a number) and verify how the function and test handle it.

Work with popular testing frameworks

Compare Copilot's test output using both unittest and pytest for the same Python function:

1. Write a function, square_root(n), that returns the square root of a number.

2. Add this prompt:

```
# Write unit tests for square_root using unittest
```

3. In the same file, add this prompt:

```
# Rewrite the tests using pytest
```

4. Compare both styles and note the key differences (e.g., assertEqual versus assert).

Write clear and structured test prompts for Copilot to follow

Use a checklist-style prompt to guide Copilot in generating complete and varied test cases:

1. Define a new function, format_currency(amount), that returns a string like $25.00.
2. Add a prompt that uses a checklist format:

```
# Write unit tests for format_currency:
# - valid integer input
# - valid float input
# - negative values
# - invalid types (e.g., string, None)
```

3. Observe how Copilot translates your checklist into tests.
4. Tweak the wording and see how the suggestions change.

5. Type a vague prompt like this and compare the result with the result of your checklist-format prompt:

```
# test
```

Apply best practices to improve the reliability of AI-generated tests

Review and refactor Copilot-generated tests to handle errors, ensure correctness, and follow consistent naming:

1. Define a function, def divide(a, b): return a / b.
2. Prompt:

```
# Write tests for divide using pytest
```

3. Review the generated tests and answer these questions:
   - Are the tests logically correct?
   - Are the tests named consistently?
   - Do the tests handle edge cases like division by zero?
4. Refactor the tests by doing the following:
   - Add a test docstring.
   - Replace repeated setup with a fixture (optional).
   - Remove or flag any incorrect assertions.

7

Using GitHub Copilot for code reviews and collaboration

In this chapter

- Explore Copilot's help with code reviews
- Generate refactoring suggestions
- Create clear inline documentation
- Use Copilot for alternative implementations
- Collaborate with teammates using AI-enhanced feedback

Practice files

There are no practice files for this chapter.

Code reviews are essential for improving code quality, catching bugs, and maintaining a consistent style across your team. But reviewing code can be time-consuming—especially when working on large projects or helping new team members ramp up. In this chapter, you'll learn how GitHub Copilot can become your silent partner during code reviews, offering suggestions to refactor code, improve readability, and identify potential issues early. In this chapter you'll discover how to use Copilot to write better documentation and get suggestions about alternative implementations in order to make your code easier for teammates to understand and maintain. You'll also learn practical ways to integrate Copilot into your collaborative workflow so you can share AI-generated insights with your team, speed up the review process, and keep everyone aligned on best practices.

By the end of this chapter, you'll know how to harness Copilot's AI capabilities not just for writing code but for reviewing, improving, and collaborating on it—making code reviews faster, more productive, and more enjoyable.

Explore Copilot's help with code reviews

Performing code reviews is one of the best ways to improve your code and grow as a developer. Code reviews help catch bugs before they reach production, ensure consistent coding standards, and encourage knowledge sharing across a team. But reviewing code—especially long or complex files—takes time and focus.

GitHub Copilot can step in as your silent assistant during a code review to make the review process faster and more effective. Copilot can quickly highlight areas of your code that could be improved, suggest optimizations, and identify outdated patterns.

Copilot code review vs. manual code review

While GitHub Copilot can make code reviews faster and easier by suggesting improvements and catching common mistakes, it's important to remember that Copilot is a tool—not a replacement for thoughtful human oversight. The following table highlights what Copilot excels at during code reviews and what still requires human judgment from you or your teammates to ensure code quality, correctness, and alignment with your project's standards.

Task	Copilot's strength	Human reviewer's role
Syntax improvements	Suggests corrections and optimizations	Confirm project conventions.
Logic correctness	Flags potential issues but cannot validate logic	Validate against requirements.
Code style consistency	Suggests consistent patterns if context is available	Enforce team standards.
Security considerations	May highlight risky patterns but lacks deep awareness	Assess vulnerabilities and use best practices.
Business logic validation	Cannot evaluate domain-specific rules	Ensure code aligns with business needs.
Code documentation	Drafts comments and docstrings	Edit for accuracy and clarity.

Use Copilot to spot inefficiencies

When you open a file you need to review, you can scroll through the code and add a prompt like this:

```
# Review this function for possible optimizations
```

Copilot will read the context and may suggest improvements, such as simplifying a loop, reducing repeated code, or replacing a slow algorithm with a more efficient one.

To see how Copilot can spot inefficiencies in code, follow these steps:

1. Open Visual Studio Code (or another IDE that supports GitHub Copilot)
2. Make sure Copilot is enabled.
3. Create a new Python file (for example, review_sum_even_numbers.py)
4. Paste the following function into the file:

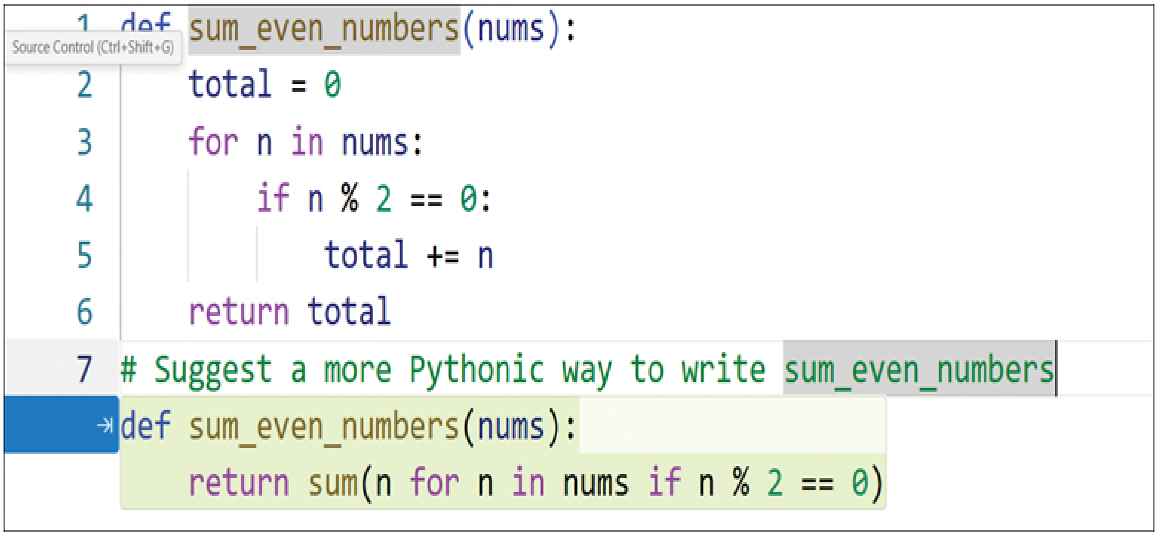
```
def sum_even_numbers(nums):
   total = 0
   for n in nums:
     if n % 2 == 0:
        total += n
   return total
```

5. Add the following prompt to ask Copilot for a more efficient rewrite, and see what it suggests:

```
# Suggest a more Pythonic way to write sum_even_numbers
```

Copilot-suggested code:

```
def sum_even_numbers(nums):
    return sum(n for n in nums if n % 2 == 0)
```



GitHub Copilot suggesting a more Pythonic implementation that improves readability and efficiency with a list comprehension

This change improves readability and performance with a single-line expression.

Tip

Use Copilot during code reviews to look for opportunities to refactor repetitive code. Copilot can often suggest clearer alternatives that you might miss when reading quickly.

Detect outdated or deprecated patterns

In many languages, functions or syntax can become outdated over time. Copilot can help identify such issues automatically. For example, say that your project uses old Python string formatting like this:

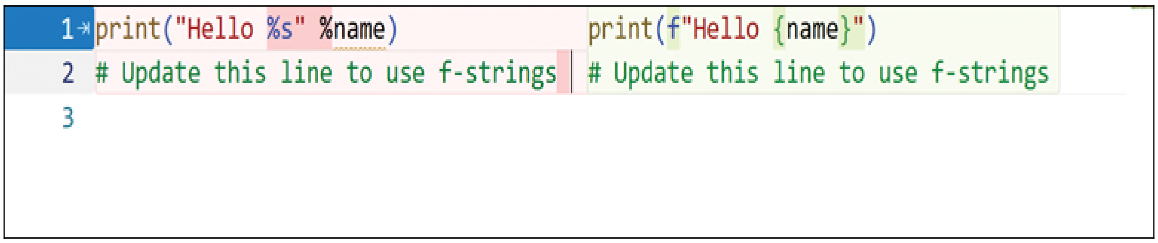
```
print("Hello %s" % name)
```

Prompt:

```
# Update this line to use f-strings
```

Copilot-suggested code:

As shown in the following figure, GitHub Copilot updates the legacy string formatting statement to a modern f-string to ensure cleaner and more Pythonic syntax that is in line with current best practices.



GitHub Copilot updating a legacy string formatting statement to a modern f-string to ensure cleaner and more Pythonic syntax

This type of change ensures that your code stays modern and consistent with current language best practices.

> **Important**
>
> Always check Copilot's suggestions for compatibility with your project's supported Python version. Some syntax (like f-strings) requires Python 3.6 or later.

Highlight potential bugs and code smells

Copilot can also help identify suspicious or risky code patterns during a review. For example, it can detect:

- Variables being used before they are assigned.
- Loops that may never terminate.

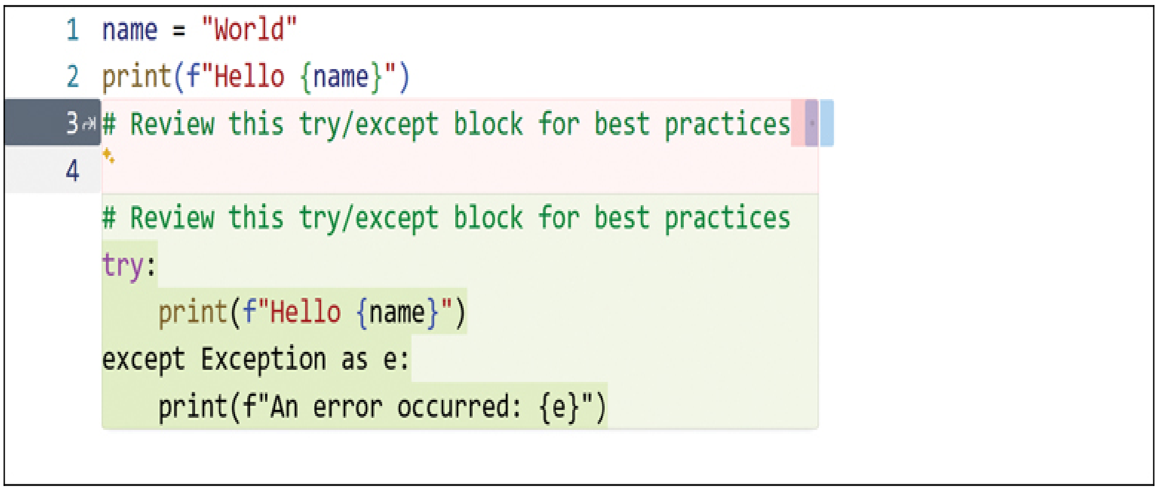
- try-except blocks that catch generic exceptions without handling them properly.

Prompt:

```
# Review this try/except block for best practices
```

Copilot-suggested code:

Based on this prompt, GitHub Copilot enhances the original print logic with the try/except block shown in the following figure. This promotes safer and more resilient Python code execution.



GitHub Copilot enhancing the original print logic with a try/except block to promote safer and more resilient Python code execution

Copilot's suggested change prevents the code from silently swallowing errors, which is a common source of hidden bugs.

Balance AI assistance with human judgment

Copilot won't replace a human reviewer, but it acts as a second pair of eyes —one that's trained on millions of code examples. By combining your expertise with Copilot's suggestions, you can:

- Reduce time spent on repetitive review tasks.
- Spot issues you might overlook on your own.
- Learn alternative, modern coding patterns.

> **Important**
>
> Copilot suggestions during code reviews should never replace your judgment. Always verify that the suggested changes match your project's logic, style guidelines, and security requirements before approving or applying them.

Common review prompts and their benefits

Before reviewing your code, you can use targeted prompts to guide Copilot in suggesting improvements. The following table shows examples of effective prompts and what they can help you achieve during code reviews.

Prompt	What it helps with
Suggest improvements for this code	Spot inefficiencies and modernize syntax.
Review for performance issues	Identify slow loops and redundant operations.
Check for outdated patterns	Update deprecated functions or syntax.

AI-assisted code review with GitHub Copilot

Use this list to ensure that you're leveraging Copilot effectively during code reviews. This list helps you identify syntax issues, outdated patterns, and inefficiencies and provides a reliable balance between AI assistance and human judgment.

1. Prompt Copilot for:
   a. Syntax improvements.
   b. Simplification opportunities.
   c. Deprecated or outdated code usage.
2. Compare Copilot suggestions with project conventions.
3. Confirm that suggestions don't break business logic.
4. Ensure that security-sensitive code is manually reviewed.
5. Use prompts like these:
   a. # Check for performance issues
   b. # Modernize syntax
   c. # Review error handling in this block
6. Validate that Copilot's suggestions are:
   a. Compatible with your tech stack.
   b. Aligned with team style guides.
7. Add inline comments summarizing Copilot recommendations.
8. Conduct a final review of whether Copilot helped you catch something you would have missed.

Generate refactoring suggestions

Refactoring means improving the structure of your existing code without changing its behavior. It makes your code easier to read, reduces duplication, and prepares your code for future updates. But figuring out the best way to refactor can be time-consuming—especially when dealing with long functions or deeply nested logic.

GitHub Copilot can help with refactoring by suggesting cleaner ways to write your code, often providing simpler, more Pythonic, or more efficient implementations. In this section, you'll learn how to prompt Copilot effectively to get refactoring help, review Copilot's suggestions, and apply changes confidently.

Prompt Copilot for refactoring ideas

You can prompt Copilot directly in a code file by adding a comment that describes what you want improved. For example, say that you have a function with duplicated logic:

```
def calculate_total(prices, discount):
    total = 0
    for price in prices:
        discounted = price - (price * discount)
        total += discounted
    return total
```

Prompt:

```
# Suggest a cleaner version of calculate_total
```

Copilot might then suggest extracting the discount calculation into a separate helper function or replacing the loop with a generator expression, as shown in the following figure.

```python
def calculate_total(prices, discount):
    total = 0
    for price in prices:
        discounted = price - (price * discount)
        total += discounted
    return total
# Suggest a cleaner version of calculate_total
def calculate_total(prices, discount):
    return sum(price - (price * discount) for price in prices)
```

GitHub Copilot refactoring the calculate_total function into a cleaner single-line sum to improve code simplicity and readability

Tip

When asking Copilot for refactoring ideas, include details like "make it more readable," "use a list comprehension," or "simplify nested if statements" to guide the style of suggestions.

Simplify conditionals and remove redundancy

Copilot can help you refactor complex if-else structures by suggesting shorter, more maintainable logic. Say that you have the following code:

```python
def categorize_score(score):
    if score >= 90:
        return "A"
    else:
        if score >= 80:
            return "B"
        else:
            if score >= 70:
                return "C"
            else:
                return "F"
```

Prompt:

```
# Refactor categorize_score to simplify nested conditionals
```

Copilot-suggested code:

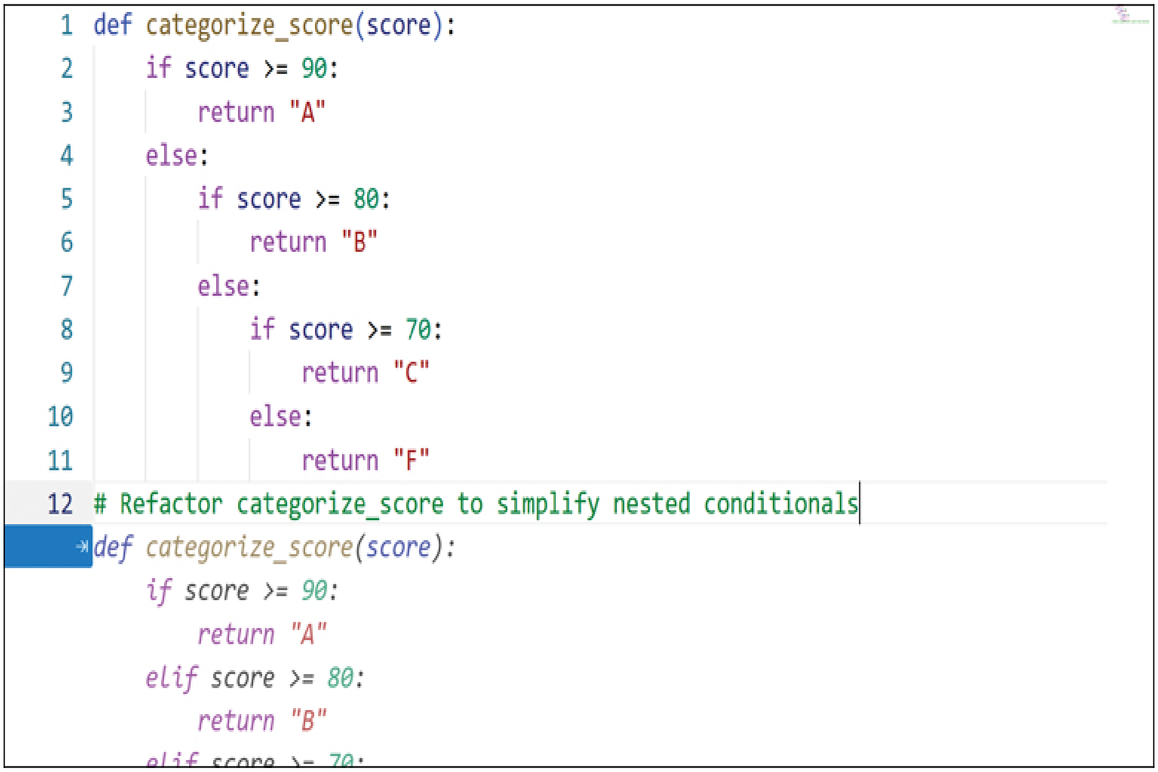
```
def categorize_score(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    else:
        return "F"
```

Rename variables and functions consistently

Copilot can suggest better, clearer names for your variables or functions. For example, say that you want Copilot to replace vague names like x, y, or temp with descriptive names to make your code easier for others to read.

Prompt:

```
# Suggest better names for variables in process_data()
```



GitHub Copilot refactoring the categorize_score function by replacing deeply nested conditionals with a cleaner if-elif-else structure based on a prompt to simplify the logic

> **Important**
>
> Renaming variables or functions affects the rest of your code. After applying Copilot's suggestions, use your IDE's rename tool or carefully find and replace to update all occurrences and avoid introducing bugs.

Identify opportunities to extract helper functions

Long functions with multiple responsibilities are a classic sign that you need refactoring. Copilot can help you split these functions into smaller, focused functions.

To see Copilot's refactoring suggestions in action, say that you have the following code:

```
def analyze_and_report(data):
    cleaned = [d.strip() for d in data]
    average = sum(len(d) for d in cleaned) / len(cleaned)
    print(f"Average length: {average}")
```

Start by asking it to extract the string-cleaning logic into its own helper function.

Prompt:

```
# Refactor analyze_and_report to extract cleaning logic into
```

Copilot-suggested code:

As you can see in the following figure, Copilot introduces a clean_data function and keeps the rest of the logic inside analyze_and_report.

```
def analyze_and_report(data):
    cleaned = [d.strip() for d in data]
    average = sum(len(d) for d in cleaned) / len(cleaned)
    print(f"Average length: {average}")
# Refactor analyze_and_report to extract cleaning logic into a helper function
def clean_data(data):
    return [d.strip() for d in data]
```

GitHub Copilot suggesting a first level of refactoring by creating a clean_data helper function to modularize the string-cleaning logic in analyze_and_report

More Copilot-suggested code:

After you accept the initial suggestion, as shown in the following figure, Copilot further improves the code by updating analyze_and_report to call the new clean_data function directly, making the code more modular and easier to maintain.

```python
def analyze_and_report(data):
    cleaned = [d.strip() for d in data]
    average = sum(len(d) for d in cleaned) / len(cleaned)
    print(f"Average length: {average}")
# Refactor analyze_and_report to extract cleaning logic into a helper function
def clean_data(data):
    return [d.strip() for d in data]

def analyze_and_report(data):
    cleaned = clean_data(data)
    average = sum(len(d) for d in cleaned) / len(cleaned)
    print(f"Average length: {average}")
```

GitHub Copilot providing a second-level enhancement, refactoring the analyze_and_report function to call clean_data, improving modularity and readability

By prompting Copilot with your refactoring goals, you can:

- Reduce code duplication.
- Improve readability and maintainability.
- Learn better ways to structure your code.
- Make updates easier and safer in the future.

See Also

In Chapter 9: Avoiding common pitfalls with GitHub Copilot, you'll learn how to avoid common pitfalls when applying AI-generated refactoring suggestions, ensuring that your code remains correct, maintainable, and secure.

Generate refactoring suggestions

Refactoring often starts with knowing what to ask Copilot. The following table provides common prompts you can use and the types of refactoring Copilot will suggest for each of them.

Prompt	Expected Copilot outcome
Refactor nested if-statements	Simplifies logic with elif or early returns.
Extract helper function	Splits long function into smaller parts.
Improve variable naming	Suggests clearer descriptive names.

Smart refactoring with Copilot

This list helps you use Copilot to refactor code for readability, modularity, and efficiency—without altering functionality. It's especially useful during code reviews or when cleaning up your own code.

1. Add comments like these to trigger refactoring:

   a. # Refactor this function for readability

   b. # Extract helper function

   c. # Simplify nested if-statements

2. Review Copilot's suggestions with these best practices in mind:

   a. List comprehensions - confirm they improve readability and performance.

   b. Pythonic one-liners - check if they enhance clarity without hurting maintainability.

c. DRY (don't repeat yourself) principles - ensure duplication is reduced without making the code harder to follow.

3. Break down long functions into smaller, testable chunks.
4. Rename variables for better clarity (for example use the variable name total_sales instead of x).
5. Use Copilot to identify and remove redundant or repeated logic.
6. Validate the behavior of the code post-refactoring by using tests or output checks.
7. Compare the original code to the refactored code, looking for whether it is:

   a. Shorter.

   b. More readable.

   c. Easier to maintain.

8. Rerun Copilot to optimize newly created functions (if needed).

Create clear inline documentation

Good code doesn't just work; it explains itself. But even clean, well-structured code can be hard for teammates (or your future self) to understand without inline documentation. Docstrings and comments provide context about what a function does, how it should be used, and what each parameter means.

Writing good documentation can feel repetitive and time-consuming, but GitHub Copilot can help you generate useful docstrings and comments quickly, saving time while improving code readability.

Generate docstrings automatically

A docstring describes a function's purpose, parameters, and return values. In Python, using docstrings is the standard way to document functions, classes, and modules.

To prompt Copilot for a docstring, place your cursor inside an empty triple-quoted string, right under your function definition, or add a comment.

Prompt:

```
def process_order(order_id, quantity):
  """ """
# Write a docstring for process_order
```

Copilot-suggested code:

As shown in the following figure, GitHub Copilot generates a docstring for the process_order function. This change improves the clarity and maintainability of the code.

```
def process_order(order_id, quantity):
    """ """
# Write a docstring for process_order
    """Process an order by ID and quantity."""
    pass
```

GitHub Copilot generating a docstring for the process_order function for better code clarity and maintainability

Tip

Include a short description of each parameter in your prompt (e.g., "order_id is a string") if you want more precise docstrings.

Add meaningful comments to complex logic

Sometimes a single docstring isn't enough—especially for complex functions or algorithms. Copilot can help generate line-by-line comments that explain what's happening at each step.

Prompt:

```
def find_duplicates(items):
  # Explain this function with comments
  seen = set()
  duplicates = set()
  for item in items:
    if item in seen:
       duplicates.add(item)
    else:
       seen.add(item)
  return list(duplicates)
```

Copilot-suggested code:

GitHub Copilot responds to the comment by explaining the find_duplicates function with inline comments. This improves the readability of the code and makes the logic easier to understand.

```python
def find_duplicates(items):
    # Explain this function with comments
    seen = set()
    # This function finds duplicates in a list of items.
    duplicates = set()
    for item in items:
        if item in seen:
            duplicates.add(item)
        else:
            seen.add(item)
    return list(duplicates)
# This function takes a list of items and returns a list of duplicates found
```

GitHub Copilot responding to a comment by explaining the find_duplicates function with inline comments

Keep comments and docstrings up to date

When you change your code, Copilot doesn't automatically know that your function has changed. As a result, old comments or docstrings may become misleading to both you and Copilot. It's important to review and update them whenever your code changes.

> **Important**
>
> After modifying your code, always review and update your documentation manually or prompt Copilot to refresh outdated comments or docstrings.

Avoid overly generic or incorrect documentation

Copilot sometimes generates generic or inaccurate docstrings, especially if your prompt is vague or the function name doesn't clearly reflect its purpose. Here is an example of a generic docstring that isn't helpful:

```
def calculate():
    """
    This function does something.
    """
```

It is important to use clear names and prompts so Copilot has context to generate meaningful documentation.

Writing and maintaining clear inline documentation helps your team:

- Understand your code more quickly.
- Avoid time-consuming explanations in meetings.
- Spot bugs because assumptions are explicit.

Copilot speeds up this process, but you need to pay attention to ensure that the documentation stays accurate.

See Also

In Chapter 8: Using AI-powered development workflows in real-world scenarios, you'll see how to apply these documentation practices in real-world AI-powered development workflows, where clear communication is essential for team success.

Create clear inline documentation

When you need help documenting your code quickly, Copilot can generate useful docstrings or comments. The following table shows examples of effective prompts to guide Copilot in generating clear and helpful documentation for your code.

Prompt	Copilot suggestion

Prompt	Copilot suggestion
Write a docstring for this function	Generates a complete function docstring.
Add comments to explain steps	Inserts inline explanations for logic.
Document class methods	Produces docstrings for all methods.

Generating inline documentation with Copilot

Documenting code clearly improves its maintainability. This list helps you use Copilot to generate useful docstrings and comments while keeping accuracy and style intact.

1. Use prompts like these:

   a. # Write a docstring for this function

   b. # Add comments to explain this logic

2. Ensure that each function has a meaningful docstring that covers the following:

   a. Purpose

   b. Parameters and their types

   c. Return value(s)

3. Ensure that inline comments:

   a. Explain why, not just what.

   b. Highlight assumptions or edge cases.

4. Avoid overly generic documentation (e.g., "This does something").
5. Refresh outdated comments/docstrings after code changes are made.
6. Align the format of docstrings with your project (e.g., Google style, NumPy style).
7. Run Copilot again post-refactoring to update docstrings automatically.
8. Use documentation as a code review aid to determine whether a new developer would understand the code.

Use Copilot for alternative implementations

Sometimes there's more than one way to solve a problem, and the first approach you write might not be the clearest, fastest, or easiest to maintain. Asking Copilot for alternative implementations can help you explore different algorithms, improve performance, or learn new techniques you might not have considered on your own.

In this section, you'll learn how to use Copilot prompts to generate alternative ways of solving the same task—and how to choose the right solution for your project.

Prompt Copilot for different solutions

If you have a working function but wonder if there's a better way, add a prompt like this:

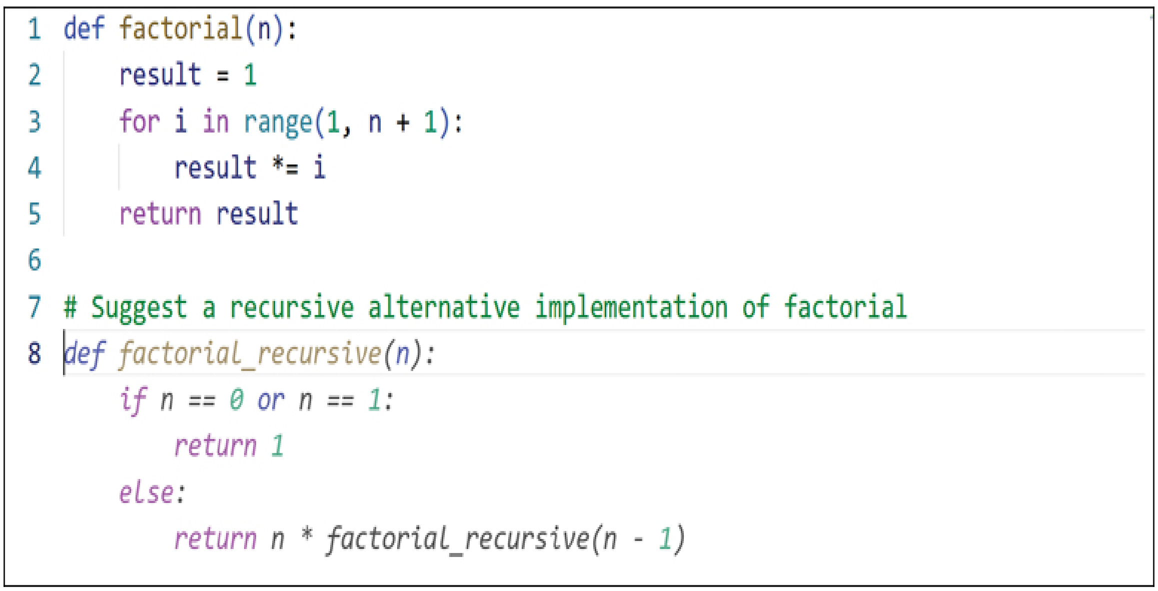
```
# Suggest an alternative implementation of this function
```

Prompt:

```
def factorial(n):
  result = 1
  for i in range(1, n + 1):
    result *= i
  return result
# Suggest a recursive alternative implementation of factorial
```

Copilot-suggested code:

As you can see in the following figure, GitHub Copilot provides a recursive version of the factorial function in response to this prompt.



GitHub Copilot providing a recursive version of the factorial function,.

This suggestion gives you a choice between iterative and recursive styles. This example showcases how GitHub Copilot can offer alternate implementations to suit different programming paradigms.

Tip

> Include details in your prompt if you have a preference—for example, “Suggest an implementation using list comprehensions” or “Write an alternative using built-in Python functions.”

Explore performance improvements

Copilot can suggest faster or more memory-efficient approaches.

Prompt:

```
def get_unique_items(items):
  unique = []
  for item in items:
    if item not in unique:
       unique.append(item)
  return unique
# Suggest a faster implementation of get_unique_items
```

Copilot-suggested code:

GitHub Copilot responds to this prompt by replacing a manual uniqueness check loop with a faster, more Pythonic implementation.

```python
def get_unique_items(items):
    unique = []
    for item in items:
        if item not in unique:
            unique.append(item)
    return unique

# Suggest a faster implementation of get_unique_items
def get_unique_items(items):
    return list(set(items))
```

GitHub Copilot responding to a user prompt by replacing a manual uniqueness check loop with a faster, more Pythonic implementation using list(set(items))

This approach leverages Python's set() function for O(1) lookups, which improves performance significantly on large lists.

Compare alternatives before choosing

While Copilot makes it easy to generate different implementations, it's up to you to:

- Compare readability.
- Consider performance impacts.
- Test correctness on real data.
- Think about maintainability for your team.

> **Important**
>
> Copilot doesn't know your business requirements, project constraints, or team conventions. Review each alternative it suggests carefully before replacing your existing code.

Discover new Python idioms and patterns

Exploring Copilot's alternatives can help you learn more idiomatic Python.

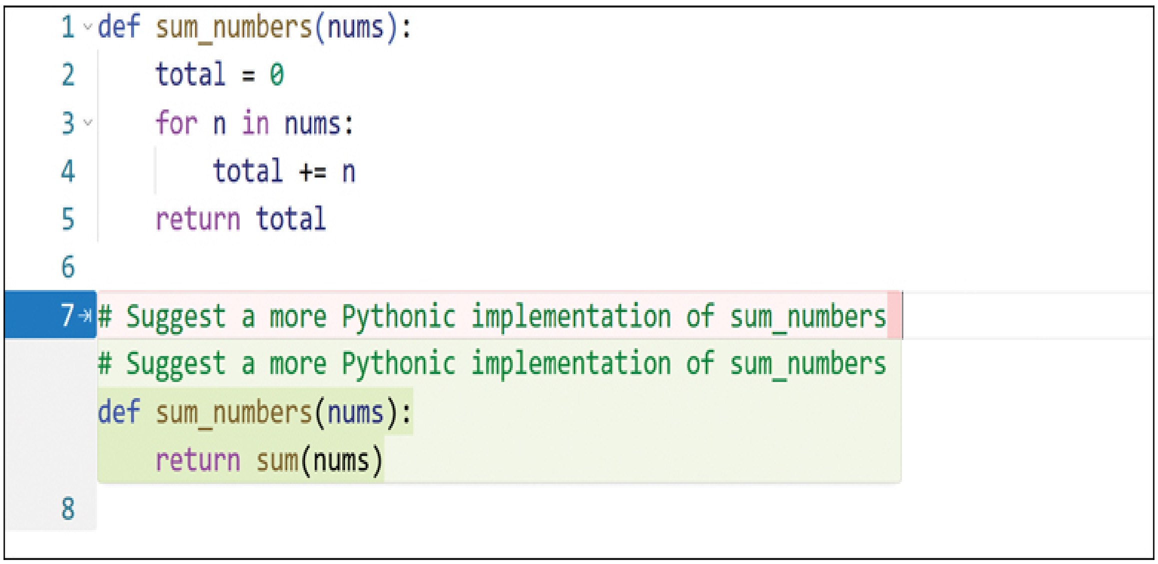
Prompt:

```
def sum_numbers(nums):
  total = 0
  for n in nums:
```

```
        total += n
    return total
# Suggest a more Pythonic implementation of sum_numbers
```

Copilot-suggested code:

GitHub Copilot replaces the manual summing loop with Python's built-in sum() function.



GitHub Copilot replacing a manual summing loop with Python's built-in sum() function, promoting concise and readable code

By prompting Copilot for alternative implementations, you:

- Expand your knowledge of Python features and idioms.
- Discover faster or simpler solutions.
- Make your code more maintainable by choosing clearer approaches.
- Build a mindset of considering multiple solutions before settling on one.

See Also

In Chapter 10: Exploring the future of AI in software development, you'll explore how AI tools like Copilot may evolve to offer even smarter and more context-aware suggestions in future coding workflows.

Exploring alternative implementations with Copilot

Sometimes Copilot's first suggestion isn't the best. Use this list to explore and evaluate different approaches to solving a coding problem—especially for performance or readability.

1. Use prompts like these with Copilot:
   a. # Suggest an alternative implementation
   b. # Write a recursive version of this
   c. # Use built-in Python methods instea
2. Generate at least two alternative solutions.
3. Evaluate suggestions based on the following:
   a. Time complexity
   b. Memory usage
   c. Clarity for future readers
4. Run tests to ensure functional equivalence.
5. Ask Copilot for idiomatic or language-specific alternatives (e.g., "Use JavaScript map/filter").
6. Avoid over-optimization; keep maintainability in mind.
7. Consider using alternatives as comments for future context.

8. Document the reason for choosing one approach over others.

Use alternative implementations

Sometimes exploring different solutions is the best way to improve your code. The following table outlines situations where asking Copilot for an alternative implementation can provide performance, readability, or maintainability benefits.

Situation	Why an alternative might help
Performance bottleneck	Discover faster algorithms.
Hard-to-read code	Improve clarity and maintainability.
Repeated complaints in code reviews	Explore different coding patterns.

Collaborate with teammates using AI-enhanced feedback

Code reviews and collaboration are at the heart of modern software development. They help teams share knowledge, maintain consistency, and catch bugs early. But they can also slow down projects when reviewers are overloaded or when feedback is vague. GitHub Copilot can help streamline collaboration by generating clear, actionable suggestions you can share with your team.

In this section, you'll learn how to integrate Copilot into your collaborative workflow, from reviewing pull requests to working together in pair-programming sessions.

Use Copilot to suggest improvements in pull requests

When reviewing a teammate's pull request, you can open their code in your IDE and prompt Copilot.

Prompts:

```
# Suggest refactoring for better readability
```

or:

```
# Recommend handling possible exceptions
```

Copilot will read the surrounding code and propose improvements you can copy into your review comments or add as inline edits in the pull request.

> **Tip**
>
> Use Copilot's suggestions as a starting point and customize your review comments to add context about why each change matters for your project.

Share AI-generated snippets in chat or documentation

If you're collaborating through Slack, Teams, or another chat tool, you can prompt Copilot to generate sample code or documentation snippets to share directly with your team.

Prompt:

```
# Create an example of how to use the new API endpoint
```

Copilot might generate sample code you can paste into your team discussion to help everyone get up to speed faster.

Improve pair programming efficiency

When developers are working together live, Copilot can act as a silent third teammate. While one developer writes code, Copilot can suggest improvements or fill in boilerplate, letting the pair focus on higher-level design decisions.

For example, as your teammate types:

```
def calculate_shipping(weight):
    # ...
```

Copilot might instantly suggest the entire function logic, speeding up the session.

```
def calculate_shipping(weight):
    # ..
    if weight <= 0:
        return "Invalid weight"
    elif weight <= 5:
        return 5.00
    elif weight <= 10:
        return 10.00
    elif weight <= 20:
        return 15.00
    else:
        return 20.00

```

A Python function that calculates shipping costs based on weight tiers, using multiple conditional branches and a placeholder comment that hints at future documentation or refinement

Keep your team aligned with consistent patterns

If you already have clean examples in your codebase, Copilot can help ensure that your team writes consistent code by generating suggestions that match your existing style.

Prompt:

```
# Write a function to log user activity following existing lo
```

This will lead Copilot to imitate patterns it finds in your project, which will help new contributors learn your conventions faster.

> **Important**
>
> Don't rely solely on Copilot to enforce team standards. Pair Copilot with code linters, formatters, and thorough reviews to keep your codebase consistent and maintainable.

Encourage thoughtful discussions

Copilot suggestions can be great conversation starters during reviews. Even if you don't accept Copilot suggestions exactly as is, discussing why a suggestion works—or doesn't—can improve your team's shared understanding.

By using Copilot during code reviews, team chats, and pair programming, you can:

- Speed up feedback cycles.
- Provide clearer, example-based suggestions.
- Help teammates learn patterns faster.
- Keep discussions focused and productive.

See Also

In Chapter 8: Using AI-powered development workflows in real-world scenarios, you'll explore real-world examples of using Copilot in team projects, including how different industries apply AI to their collaborative development workflows.

Collaborate effectively using Copilot in VS Code

You don't need a special environment like Codespaces to use Copilot with a team. By combining GitHub Copilot with VS Code's built-in Git features, you can share AI-assisted code changes, review them with teammates, and discuss improvements—all within your local editor.

Follow these steps to collaborate seamlessly:

1. Open your project in VS Code, making sure you're signed in to the GitHub account that is linked to your Copilot subscription.
2. Write or review code. Normally Copilot suggests code completions and improvements as you type.
3. Accept or modify Copilot's suggestions. Press Tab or Enter to insert suggestions or edit them as needed.
4. Commit your changes. Use the Source Control panel (Ctrl+Shift+G) to stage and commit your Copilot-assisted edits as you would any other code.

5. Push your commits to your team repository on GitHub. Sync your branch so your team can see your changes.
6. Create a pull request. Use VS Code's GitHub pull requests extension or open GitHub in your browser to create a pull request for your branch.
7. Discuss Copilot-generated code in the pull request. Add context about why you accepted a suggestion and invite your teammates to review the code.
8. Incorporate team feedback. Update your code based on comments. Prompt Copilot again to help revise your code faster.

Tip

Copilot suggestions become part of your regular commits and pull requests. You don't need to do anything special to share them. Just commit and push your code as usual.

Best practices for collaborating with Copilot in code reviews

When using Copilot in a team setting, clear communication and thoughtful collaboration are essential. While Copilot can accelerate your work, following best practices ensures that everyone on your team understands, trusts, and benefits from AI-generated suggestions. The following table summarizes simple but powerful habits to adopt when collaborating with Copilot-assisted code reviews.

Best practice	Why it matters

Best practice	Why it matters
Review Copilot suggestions together.	Builds shared understanding and trust.
Explain why you accept a suggestion.	Helps teammates learn decision making.
Avoid blind acceptance.	Maintains code quality and consistency.
Update team members on AI usage.	Ensures that everyone knows how Copilot is used.
Combine use of Copilot with manual review.	Balances the speed of AI and human insight.

Collaborating with your team using Copilot feedback

When working in teams, Copilot can help streamline discussions, code reviews, and shared development. This list helps you use AI suggestions to foster collaboration—not confusion.

1. While reviewing a teammate's code:

   a. Add Copilot prompts like # Suggest refactoring for clarity.

   b. Share Copilot-generated snippets in Pull Request(PR) comments or chat.

2. In pair programming:

   a. Have one developer write while the other validates Copilot suggestions.

   b. Use Copilot to speed up boilerplate and helper functions.

3. When writing shared utilities:

   a. Prompt Copilot to mimic existing code structure.

   b. Use prompts like # Follow team’s logging style.

4. Avoid pushing Copilot code without explanation. Leave notes or links.

5. Discuss Copilot suggestions in PRs (e.g., “AI suggested this loop unrolling, thoughts?”).

6. Use team linters and Continuous Integration (CI) tools to validate Copilot-generated code.

7. Encourage teammates to ask: “Would you write this differently than Copilot?”

8. Set team rules for how Copilot is used. Transparency is key.

Common Q&A

Before wrapping up this chapter, let’s look at some of the most common questions developers have about using GitHub Copilot for code reviews and team collaboration. These quick Q&As will help you feel confident integrating Copilot into your workflow and sharing AI-assisted code with your teammates.

Q: Do my teammates need Copilot in order to review Copilot-generated code?

A: No, your teammates don’t need to have Copilot subscriptions. Copilot suggestions become normal code once you accept them. Anyone can review your commits and pull requests as they would any other code.

Q: Will Copilot suggestions automatically follow my team's coding standards?

A: Not always. Copilot learns from public code and may generate suggestions that use different naming conventions or patterns. Always review and adjust suggestions to match your project's style guide.

Q: Can Copilot help me write comments for code I didn't write myself?

A: Yes. If you're reviewing a teammate's code, you can add a prompt like # Add a docstring for this function, and Copilot will suggest documentation, even if you didn't write the original code.

Q: How do I make sure Copilot-generated changes are safe?

A: Treat Copilot suggestions like code from a junior developer or an open-source snippet: Review carefully, write tests, and follow your team's review process before merging.

Q: Can I use Copilot to collaborate on private repositories?

A: Yes. Copilot works with private repos you have access to. Your prompts and code stay private unless you explicitly share them in a public PR or repository.

Q: Does Copilot store or share my code with others?

A: No. By default, Copilot does not save your code or suggestions for reuse. However, GitHub may collect telemetry data (if this setting is enabled) to improve the service. You can manage this and other privacy settings in your Copilot preferences.

Skills review

In this chapter, you learned how to:

- Prompt Copilot to identify potential issues, suggest improvements, and streamline the review process.

- Spot inefficiencies, simplify logic, and improve code readability without changing its functionality.
- Ask Copilot to suggest different coding approaches that may enhance performance, maintainability, or readability.
- Guide Copilot to produce concise, accurate comments and docstrings that make your code easier for others to understand.

Practice tasks

Complete these tasks in your preferred Python development environment with GitHub Copilot enabled to reinforce what you learned in this chapter.

Use an integrated development environment (IDE) such as Visual Studio Code with GitHub Copilot enabled and signed in. The following practice tasks use Python as the language. You can adapt them to other supported languages as needed.

To try the following tasks in Python, begin by creating a new Python file (e.g., tests_example.py) in an open workspace/folder.

> **Tip**
>
> You can try these practice tasks in any programming language supported by GitHub Copilot, such as JavaScript, TypeScript, Java, or C#. Copilot will adapt its suggestions to the language and framework you're using.

Explore Copilot's help with code reviews

Use Copilot to review and suggest improvements in existing code:

1. Create a new Python file named test_example.py and add a function to it.
2. Add the following prompt under the function:

```
# Suggest improvements for this function.
```

3. Accept or review Copilot's suggestions for refactoring or performance.

Generate refactoring suggestions

Prompt Copilot to simplify or restructure your code:

1. Write a function that includes nested if statements or duplicated code.
2. Add a prompt like this under the code:

```
# Refactor this function to improve readability.
```

3. Apply Copilot's suggested changes and test to ensure that your code still works.

Create clear inline documentation

Use Copilot to add docstrings or comments to your code:

1. Write a new function that does not include a docstring.
2. Place an empty triple-quoted string (""" """) below the function header or add the following prompt:

```
# Write a docstring for this function.
```

3. Accept Copilot’s docstring and adjust it as needed.

Use Copilot for alternative implementations

Explore different ways to solve the same problem:

1. Write a simple function, such as a function that calculates the factorial or reverses a string.
2. Add the following prompt under the function:

```
# Suggest an alternative implementation.
```

3. Compare Copilot’s suggestion with your original code and decide which version to keep.

Collaborate with teammates using AI-enhanced feedback

Practice sharing Copilot’s suggestions in a team context:

1. Pretend that you’re reviewing a teammate’s code or use a shared file.
2. Add the following prompt:

```
# Recommend improvements for this function.
```

3. Copy Copilot’s suggestion into a simulated pull request comment or share it in a team chat.

8

Using AI-powered development workflows in real-world scenarios

In this chapter

- Discover real-world Copilot use cases
- Explore AI workflows
- Understand the role of AI across industries
- Apply practical examples of AI in your projects
- Reflect on Copilot's impact on teams

Practice files

There are no practice files for this chapter.

GitHub Copilot is more than just a code-completion tool; it's a valuable assistant across many real-world development scenarios. In this chapter, you'll explore how developers in different roles and industries use Copilot to enhance productivity, streamline workflows, and reduce repetitive coding. From building web applications to processing data and maintaining backend systems, Copilot can adapt to the specific needs of each environment. Through practical examples and real-world use cases, in this chapter you'll see how AI-powered development is being applied in everyday projects. This chapter also highlights how teams are integrating Copilot into their processes and how it's reshaping collaboration, speed, and code quality across industries.

Discover real-world Copilot use cases

Developers across a wide range of fields are using GitHub Copilot not just to complete lines of code but to build real, working solutions faster. From frontend design to machine learning (ML) pipelines, Copilot adapts to your context and coding style, helping streamline common tasks and reduce friction in the development process.

In this section, we'll look at how GitHub Copilot is used in various real-world roles and coding environments.

Web development

Much of the day-to-day work of a frontend developer involves writing repetitive markup, styling elements, and handling common browser-side logic. Whether you're building from scratch or working in a framework like React or Vue, GitHub Copilot can help you speed up development and reduce the time you spend on boilerplate code.

In this section, you'll see how Copilot supports key frontend tasks such as HTML structure creation, responsive layout design, and client-side validation—all based on natural language prompts.

HTML/CSS

Writing HTML from scratch can get repetitive. Copilot can help. Consider this example.

Prompt:

```
<!-- Basic HTML5 page with Bootstrap -->
```

Copilot-generated code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-sc
    <link rel="icon" type="image/x-icon" href="favicon.ico">
    <title>Document</title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.c
bootstrap/4.5.2/css/bootstrap.min.css">
</head>
<body>
    <div class="container">
        <h1>Hello, world!</h1>
    </div>
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js
</body>
</html>
```

As you can see here and in the following figure, Copilot generates a complete structure. It gives you a complete starting point in seconds, including correct meta tags and linked stylesheets.

```html
<!-- Basic HTML5 page with Bootstrap -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
    <title>Document</title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/
</head>
<body>
    <div class="container">
        <h1>Hello, world!</h1>
    </div>
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
```

A basic HTML5 page with Bootstrap, generated using GitHub Copilot from a descriptive comment

Responsive components

You can also ask Copilot to generate responsive elements using modern CSS frameworks like Tailwind or Bootstrap.

Prompt:

```html
<!-- Create a responsive navigation bar using Tailwind CSS --
```

Copilot-generated code:

As shown here, Copilot creates code that is fully responsive out of the box and can be customized further with utility classes.

```html
<!-- Create a responsive navigation bar using Tailwind CSS -->
<!-- Create a responsive navigation bar using Tailwind CSS -->
<nav class="bg-gray-800">
  <div class="max-w-7xl mx-auto px-2 sm:px-6 lg:px-8">
    <div class="relative flex items-center justify-between h-16">
      <div class="absolute inset-y-0 left-0 flex items-center sm:hidden">
        <!-- Mobile menu button-->
      </div>
      <div class="flex-1 flex items-center justify-center sm:items-stretch sm:
        <div class="flex-shrink-0">
          <h1 class="text-white">My Website</h1>
        </div>
        <div class="hidden sm:block sm:ml-6">
          <div class="flex space-x-4">
            <a href="#" class="text-gray-300 hover:bg-gray-700 hover:text-whit
```

A responsive navigation bar built with Tailwind CSS, generated using GitHub Copilot from a descriptive HTML comment

This example showcases Copilot's ability to translate frontend UI prompts into Tailwind-powered navigation code.

Tip

GitHub Copilot is highly effective with frameworks like React, Vue, and Next.js. For example, in a .jsx or .tsx file, try using this prompt:

```
// React component for a mobile-friendly login form
```

Copilot will generate a functional, styled JSX block that you can drop into your app.

JavaScript validation logic

Frontend validation is another repetitive task that Copilot simplifies. Consider this example.

Prompt:

```
// Validate email and password fields on form submit
```

Copilot-generated code:

Copilot instantly scaffolds the basic validation logic, giving you a working template that can be expanded based on your needs.

```
// Validate email and password fields on form submit
// Validate email and password fields on form submit
document.getElementById("myForm").addEventListener("submit", function(event){
    var email = document.getElementById("email").value;
    var password = document.getElementById("password").value;
    if(!validateEmail(email)){
        alert("Invalid email format");
        event.preventDefault();
    }
    if(password.length < 6){
        alert("Password must be at least 6 characters long");
        event.preventDefault();
    }
});
```

A JavaScript form validation script generated by GitHub Copilot that validates email format and password length

Whether you're scaffolding a basic layout, writing component markup, or handling user input, Copilot speeds up frontend workflows by understanding common patterns and generating full HTML, CSS, and JavaScript blocks with minimal input.

See Also

In Chapter 9: Avoiding common pitfalls with GitHub Copilot, you'll learn to identify common mistakes developers make when using GitHub Copilot and learn how to refine AI-generated code for better accuracy and long-term maintainability.

Backend development

Backend developers build and maintain the invisible engine that powers web applications, APIs, and enterprise systems. This involves writing server-side logic, interacting with databases, managing authentication, and handling middleware. These tasks are often repetitive and logic heavy, so they're perfect for Copilot.

In this section, you'll see how Copilot supports common backend development activities, particularly in Python-based environments like Flask or Django. You'll also see solutions that are adaptable to other ecosystems, such as Express.js and FastAPI.

Creating RESTful endpoints

Writing API routes is a foundational backend task. Based on a simple prompt, Copilot can generate route handlers that return structured data, often prefilled with HTTP methods, JSON responses, and appropriate decorators.

Prompt:

```
# Flask route to return user profile as JSON
```

Copilot-generated code:

GitHub Copilot generates complete backend route logic from a natural language prompt. You can customize the logic based on real database access or request validation.

```
1  # Flask route to return user profile as JSON
   @app.route('/api/profile', methods=['GET'])
   def get_profile():
       user = {
           'id': 1,
           'name': 'John Doe',
           'email': 'john.doe@example.com'
       }
       return jsonify(user)
```

A Flask API route function that returns a user profile as JSON

Handling database queries

Backend services often need to interact with databases to fetch, insert, or update data. Copilot can help write object-relational mapping (ORM)–based queries for libraries like Structured Query Language (SQL) Alchemy or Django ORM, as well as raw SQL when needed.

Prompt:

```
# Write a SQLAlchemy query to get all orders placed in the la
```

Copilot-generated code:

GitHub Copilot generates a SQLAlchemy query that filters orders placed in the past 30 days. This time-based filter selects recent records using a common pattern that Copilot quickly replicates.

```python
# Write a SQLAlchemy query to get all orders placed in the last 30 days
# Write a SQLAlchemy query to get all orders placed in the last 30 days
from datetime import datetime, timedelta
from your_application import db
from your_application.models import Order

thirty_days_ago = datetime.now() - timedelta(days=30)
orders = db.session.query(Order).filter(Order.date >= thirty_days_ago).all()
```

A SQLAlchemy query generated by GitHub Copilot that retrieves all orders placed in the past 30 days

Tip

You can prompt Copilot for raw SQL, using a prompt like this:

```
# SQL query to count active users grouped by role
```

Implementing middleware

Middleware is used to intercept, transform, or log requests before they reach route handlers. Copilot can generate middleware functions tailored to your framework.

Prompt:

```
// Express.js middleware to log all incoming requests
```

Copilot-generated code:

```
function logRequests(req, res, next) {
  console.log('${req.method} ${req.url}');
  next();
}

app.use(logRequests);
```

Although this is a JavaScript example, the same concept applies to Python frameworks like Flask and FastAPI.

Prompt:

```
# FastAPI middleware to log request path and method
```

Copilot-generated code:

```
from fastapi import Request

@app.middleware("http")
async def log_requests(request: Request, call_next):
    print(f"Request: {request.method} {request.url}")
    response = await call_next(request)
    return response
```

Important

Always review database and security-related code. Copilot can't understand your business rules, data retention policies, or authentication models. It may generate code that appears correct but that introduces vulnerabilities like exposing raw database fields or skipping access controls.

As you have seen in this section, Copilot offers powerful assistance in backend workflows, from scaffolding APIs to streamlining database access and middleware logic. It helps reduce boilerplate, suggest best practices, and simplify complex logic. However, it is important to always review the results it offers, especially around security and data handling.

Data science and analytics

In data science workflows, developers and analysts often spend time cleaning data, running exploratory analyses, building machine learning models, and generating visualizations. These tasks involve a lot of boilerplate, repetitive steps, and function lookups across libraries like pandas, NumPy, scikit-learn, and matplotlib.

GitHub Copilot can assist by accelerating common tasks, remembering syntax, and generating ready-to-run code snippets that help you focus more on insights and less on formatting.

Data cleaning and transformation

Data cleaning is often the first and most time-consuming part of an analysis. Copilot can help automate a number of data cleaning and transformation tasks, including the following:

- Handling missing values
- Normalizing or scaling features
- Merging or reshaping DataFrames

Prompt:

```
# Drop rows with missing values and normalize numerical colum
```

Copilot-generated code:

GitHub Copilot generates a sample DataFrame and suggests dropping rows with missing values and normalizing numerical columns.

```python
# Drop rows with missing values and normalize numerical co
import pandas as pd

# Create a sample DataFrame
data = {
    'A': [1, 2, None, 4],
    'B': [4, None, None, 1],
    'C': [7, 8, 9, 10]
}
df = pd.DataFrame(data)

# Drop rows with missing values
df = df.dropna()

# Normalize numerical columns
df[['A', 'B', 'C']] = (df[['A', 'B', 'C']] - df[['A', 'B',
```

GitHub Copilot assistance in data preprocessing using Pandas

Tip

Copilot can generate preprocessing pipelines for scikit-learn. Try this prompt:

```
# Create a sklearn pipeline with imputation and scaling
```

Generating visualizations

Visualization is essential for understanding trends and patterns in data. Copilot helps by auto-generating charts with correct syntax and labeling.

Prompt:

```
# Plot a histogram of customer ages using matplotlib
```

Copilot-generated code:

GitHub Copilot automatically generates a complete Python script using matplotlib.pyplot to visualize customer age distribution. It intelligently includes sample data, formatting options, axis labels, and title configuration, showcasing Copilot's ability to assist in data visualization tasks with minimal user input.

```python
# Plot a histogram of customer ages using matplotlib
import matplotlib.pyplot as plt

# Sample data
ages = [22, 25, 27, 30, 22, 25, 30, 35, 40, 22, 25, 30, 35,

plt.hist(ages, bins=5, alpha=0.7, color='blue')
plt.title('Histogram of Customer Ages')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show()
```

Python code automatically generated by GitHub Copilot to plot a histogram of customer ages using Matplotlib

As you can see, Copilot can save you time typing boilerplate code and help you maintain clean, well-labeled charts.

Building machine learning models

Whether you're experimenting or prototyping, Copilot can write baseline ML models and evaluation logic, including the following:

- Classification or regression pipelines
- Split logic for training and testing
- Evaluation metrics such as accuracy or root mean square error (RMSE) metrics

Prompt:

```
# Create a decision tree classifier using scikit-learn
```

Copilot-generated code:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
X = df.drop("target", axis=1)
y = df["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_
size=0.2, random_state=42)
model = DecisionTreeClassifier()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, predictions))
```

Important

Copilot provides syntax and structure but not statistical guidance. Always validate assumptions, check for overfitting, and use domain knowledge when interpreting results.

Jupyter notebook support

Copilot is especially effective in Jupyter notebooks, where each cell is focused and context aware. You can:

- Prompt it with comments in markdown or code.

- Use autocomplete to generate chained .pandas or .plotly logic.
- Iterate faster while keeping notebooks readable.

Prompt:

```
# Show correlation heatmap of numeric columns
```

When you enter this prompt in a Jupyter notebook cell, Copilot will likely respond with a seaborn or matplotlib heatmap block.

Copilot enhances productivity in data workflows by helping with data preparation, modeling, and visualization—all while reducing syntax errors and repetitive typing. It doesn't replace domain expertise, but it gets you to analysis and experimentation more quickly.

See Also

In Chapter 10: Exploring the future of AI in software development, you'll explore how tools like Copilot are shaping the future of AI-assisted data analysis and the evolving role of developers and analysts in machine learning workflows.

DevOps and automation

In DevOps, small automation tasks can save hours of manual effort. GitHub Copilot is particularly helpful in this space, where shell scripting, infrastructure as code, and configuration management all benefit from reusable patterns.

From writing Dockerfiles to setting up CI/CD workflows and scripting with Bash or PowerShell, Copilot can reduce setup time and improve consistency across environments.

Writing Dockerfiles

Creating Dockerfiles requires you to remember specific syntax and best practices, which Copilot can quickly scaffold based on your intent.

Prompt:

```
# Dockerfile for a Python Flask app with requirements.txt
```

Copilot-generated code:

```
FROM python:3.9
WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

This is a complete, minimal setup that is ready for containerization.

Tip

You can refine your prompt to include specific needs, as in this example:

```
# Dockerfile for a FastAPI app with uvicorn
```

Creating GitHub Actions workflows

Copilot can generate YAML workflows for GitHub Actions based on your deployment or testing goals.

Prompt:

```
# GitHub Actions workflow to run Python tests on push
```

Copilot-generated code:

```yaml
name: Run Tests
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.9'
    - name: Install dependencies
      run: pip install -r requirements.txt
    - name: Run tests
      run: pytest
```

This configuration works out of the box, and you can easily modify it for your project needs.

Scripting with Bash or PowerShell

System administrators and DevOps engineers often write scripts to manage files, monitor logs, and automate cleanup tasks. Copilot supports both Bash and PowerShell, making cross-platform scripting faster and more reliable.

Prompt:

```
# Bash script to rename all .txt files to .md
```

Copilot-generated code:

```bash
for file in *.txt; do
  mv "$file" "${file%.txt}.md"
```

```
done
```

This kind of pattern-based scripting is where Copilot shines, especially when you need quick utilities without re-Googling syntax.

> **Important**
>
> Always test AI-generated scripts in a safe environment. Shell and PowerShell commands can cause irreversible file or system changes if not reviewed carefully.

Infrastructure as code (IaC)

You can use Copilot to scaffold Terraform, ARM, Bicep, or YAML files for provisioning cloud resources. Copilot significantly boosts productivity in DevOps workflows by eliminating repetitive scripting and configuration writing. From cloud provisioning to automation and deployment pipelines, Copilot acts as a fast, responsive assistant for infrastructure engineers.

Prompt:

```
# Terraform configuration for an AWS S3 bucket with versionin
```

Copilot-generated code:

```
resource "aws_s3_bucket" "my_bucket" {
  bucket = "my-versioned-bucket"
  versioning {
    enabled = true
  }
}
```

Even though IaC files are declarative, Copilot can assist with syntax, naming, and optional fields.

Testing and debugging

Testing and debugging are two of the most time-consuming yet essential parts of the development cycle. GitHub Copilot can help by automating test creation, spotting overlooked edge cases, and even suggesting fixes during debugging.

Whether you're working with unit tests or integration tests or chasing down a cryptic error message, Copilot offers helpful completions that can boost both speed and test coverage.

Generating unit tests

Developers often delay writing unit tests because of their repetitive nature. Copilot can generate well-structured test cases from just a function name or a short comment.

Prompt:

```
# Write pytest tests for calculate_discount()
```

Copilot-generated code:

Copilot generates a Python unit test that uses pytest to verify the correctness of the calculate_discount() function with multiple inputs and expected discounted outputs. This example demonstrates Copilot's ability to infer test logic based on a simple prompt, streamlining the test-writing process for developers. Copilot helps maintain test-driven development habits without slowing down progress.

```python
# Write pytest tests for calculate_discount()
def test_calculate_discount():
    assert calculate_discount(100, 10) == 90
    assert calculate_discount(200, 20) == 160
    assert calculate_discount(300, 30) == 240
```

A Python unit test automatically generated by GitHub Copilot

Tip

You can also ask Copilot to use a specific framework, as in this example:

```
# Write unittest-style test cases for convert_temperature
```

Creating integration tests with mocks

Integration tests often rely on mocked components, especially when testing APIs or external services. Copilot can assist by scaffolding common mocking patterns.

Prompt:

```
# Use unittest.mock to test fetch_user_data API call
```

Copilot-generated code:

```
import unittest
from unittest.mock import patch

class TestFetchUserData(unittest.TestCase):
    @patch('requests.get')
    def test_fetch_success(self, mock_get):
        mock_get.return_value.status_code = 200
        mock_get.return_value.json.return_value = {"name": "Alice

        result = fetch_user_data("http://api.example.com/user")
        self.assertEqual(result["name"], "Alice")
```

The above example provides a strong foundation for creating reliable and reproducible tests.

Suggesting fixes while debugging

Copilot is also useful during debugging. When you encounter an error, you can write a comment that describes the issue and asks Copilot to suggest a fix.

Prompt:

```
# Fix this IndexError in the loop
def print_list_items(items):
  for i in range(len(items) + 1):
    print(items[i])
```

Copilot-generated code:

```
def print_list_items(items):
    for i in range(len(items)):
        print(items[i])
```

> **Important**
>
> While Copilot can identify and suggest code fixes, it doesn't understand the full context or data. Always test AI-suggested fixes and verify that they solve the root problem without introducing new issues.

Copilot is especially valuable for reducing the friction involved in writing and maintaining tests and can serve as a second pair of eyes during debugging. It helps ensure better coverage, fewer missed edge cases, and faster time to resolution when things break.

See Also

In Chapter 9: Avoiding common pitfalls with GitHub Copilot, you'll dive into common pitfalls developers encounter when they rely too heavily on Copilot. You'll also learn how to review AI-generated tests and fixes for accuracy and maintainability.

Explore AI workflows

GitHub Copilot isn't just a tool for isolated tasks. It fits naturally into entire development workflows. Whether you're working solo or as part of a team, Copilot can support every phase of the software lifecycle: from planning and scaffolding to writing logic, generating tests, documenting, and even preparing for deployment.

In this section, we explore how AI-powered workflows look in practice and how Copilot can make development faster, more consistent, and more enjoyable without replacing the need for thoughtful human input.

The AI-assisted development lifecycle

GitHub Copilot supports more than just code completion. It can assist at nearly every phase of a typical software development lifecycle. By prompting Copilot with natural language comments, developers can automate repetitive work, reduce context switching, and maintain momentum throughout a project.

Let's explore how Copilot contributes to each phase of a real-world development workflow.

Plan

Before writing any code, a developer typically creates folders, files, and TODOs to map out the structure of their application. Copilot can help scaffold project setup and generate clear planning comments that make the next steps obvious.

Python prompt:

```
# TODO: Set up routes for user login, logout, and registratio
```

Copilot-generated code:

```
# Route: /login [POST] - validate credentials and start session
# Route: /logout [GET] - end session and redirect
# Route: /register [POST] - create new user account
```

Bash prompt:

```
# Shell script to create folders for backend, frontend, and t
```

Copilot-generated code:

```
mkdir backend frontend tests
touch backend/app.py frontend/index.html tests/test_app.py
```

Code

The code stage of the software development lifecycle is where Copilot really shines. As you begin writing functions, classes, and handlers, Copilot can autocomplete logic based on your naming conventions and surrounding code.

Python prompt:

```
# Function to calculate total price with tax and discount
```

Copilot-generated code:

```
def calculate_total(price, tax_rate, discount):
    taxed_price = price + (price * tax_rate)
    final_price = taxed_price - discount
    return final_price
```

Copilot also helps with edge cases, such as input validation, error handling, and conditional logic, saving you from writing boilerplate manually.

Test

Many developers delay or skip testing because of time pressure. Copilot can help by generating unit, integration, or even parameterized tests, based on just a few comments.

Python prompt:

```
# Write pytest tests for calculate_total()
```

Copilot-generated code:

```
def test_calculate_total_basic():
    assert calculate_total(100, 0.1, 10) == 100

def test_calculate_total_zero_discount():
    assert calculate_total(200, 0.2, 0) == 240
```

Additional Python prompt:

```
# Use unittest and mock an API call in fetch_user_data()
```

GitHub Copilot is especially useful in enterprise or backend workflows where APIs and database calls must be tested in isolation.

Document

Maintaining clear documentation is critical for team handoff, future updates, and open-source contributions. Copilot can generate docstrings, inline comments, and even full sections for README files.

Python prompt:

```
"""Calculate total price after tax and discount."""
```

Copilot-generated code:

```
"""
Args:
    price (float): Original item price
    tax_rate (float): Tax rate as a decimal
    discount (float): Discount amount

Returns:
    float: Final price after tax and discount
"""
```

Markdown prompt:

```
# Usage
```

Copilot-generated code:

```
Run 'python app.py' and navigate to http://localhost:5000
homepage.
```

Deploy

Whether you’re deploying a web app or pushing changes through CI/CD, Copilot can help generate Dockerfiles, GitHub Actions YAML workflows, and Bash scripts.

Dockerfile prompt:

:

```
# Dockerfile for a FastAPI app running with uvicorn
```

Copilot-generated code:

```
dockerfile
FROM python:3.10
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"
Or try:
```

GitHub Actions workflow prompt:

```
# GitHub Actions workflow to deploy Python app on push
```

In response to this prompt, Copilot will scaffold a ready-to-run deployment workflow.

From idea to production, GitHub Copilot enhances nearly every step of the developer experience. It doesn't replace your thinking; rather, it accelerates repetitive parts of the process, giving you more time to focus on high-value decisions.

> **Tip**
>
> Consider building a "workflow prompt library" of reusable comments or patterns that guide Copilot at each stage of the project lifecycle.

GitHub Copilot workflow tips

Use this list to guide your end-to-end development workflow with GitHub Copilot, from planning to deployment and reflection. It can help ensure that you're using Copilot effectively at every stage.

1. Plan
   - Define the project goal or feature with a comment or markdown prompt.
   - Use Copilot to scaffold the file/folder structure (e.g., backend/, frontend/, tests/).
   - Add TODOs to break down major tasks.
   - Prompt for route definitions, module layout, or config files.
2. Code
   - Start writing functions or classes that have descriptive comments or names.
   - Let Copilot autocomplete logic, loops, conditionals, or method stubs.
   - Prompt for edge-case handling or validation.
   - Modularize your code for easier testing and reuse.
3. Test
   - Use prompts like # Write pytest tests for... to generate unit tests.
   - Scaffold mocks and integration tests with framework-specific prompts.
   - Check for missing edge cases or input conditions.
   - Run tests and validate the AI-generated test coverage.
4. Document
   - Generate docstrings for all key functions and classes.

- Prompt for README or usage documentation sections.
- Add inline comments and API annotations for clarity.
- Include sample input/output, if applicable.

5. Deploy

- Use prompts to create Dockerfiles, requirements.txt files, or build scripts.
- Generate CI/CD workflows (e.g., GitHub Actions YAML) for testing/deployment.
- Confirm that environment variables, secrets, and platform configs are handled securely.
- Test deployment locally or on a staging environment.

6. Refactor and review

- Prompt Copilot for code improvements or simplifications (e.g., # Refactor this).
- Review AI-generated suggestions for logic, readability, and performance.
- Refactor to follow team or project conventions.
- Rerun tests after changes.

7. Collaborate

- Share consistent prompt patterns with your team.
- Create prompt templates for repeatable tasks (tests, docs, routes, etc.).
- Use Copilot to help onboard new team members by generating clear examples.

8. Reflect

- Track which tasks Copilot helped with most (e.g., boilerplate, test writing).

- Identify any inaccurate or misleading suggestions and refine prompt habits.
- Note useful prompts for reuse in future projects.

Manual vs. Copilot-assisted workflow

One of the clearest ways to understand GitHub Copilot's value is to compare traditional development steps with how they look when enhanced by AI. Copilot doesn't eliminate your role. Rather, it accelerates the repetitive parts, letting you focus more on structure, problem solving, and review.

The following table outlines how each stage of a development workflow changes when using Copilot compared to working manually.

Task	Manual workflow	Copilot-assisted workflow
Scaffold files	Manually create directories or clone repos.	Prompt Copilot to create folders and starter files.
Write logic	Think through syntax or search online.	Prompt and tab through suggestions inline.
Generate tests	Write from scratch based on known behavior.	Add comment like # Write unit tests for...
Create documentation	Write docstrings post-development.	Generate documentation inline using comments or empty quotes.

Task	Manual workflow	Copilot-assisted workflow
Create a deployment config	Look up Docker/YAML examples.	Use prompts like # GitHub Actions workflow for Python app.

This comparison highlights how Copilot is a time-saver at every step of the development cycle, taking repetitive tasks off your plate so you can concentrate on thinking.

> **Tip**
>
> Encourage your team to track where Copilot saves time. Patterns will emerge that help you build your own library of useful prompts and techniques.

Best practices for Copilot in your workflow

To make the most of Copilot in your daily work, it's important to approach it as a collaborator, not as a code factory. The more intentional and thoughtful your interaction, the better the results. Let's look at some key best practices.

Start small

- If you're just beginning with Copilot, don't try to generate full classes or services right away. Use it first for utility functions, boilerplate code, and repetitive setup logic (e.g., route handlers, test templates). Build confidence gradually and then scale up your usage.

Prompt clearly

Copilot responds best to simple, descriptive prompts. Vague prompts produce unreliable results. Clear prompts produce relevant, useful code more consistently.

Vague prompt:

```
# do the thing
```

Clear prompt:

```
# Write a function to sort a list of products by price descen
```

Review as you go

Don't treat Copilot's output as final. Treat it like a junior developer's suggestion. Check it for:

- Logic errors.
- Security flaws.
- Edge cases.
- Code style consistency.

Make editing Copilot's suggestions part of your natural flow.

Keep your own voice

As powerful as Copilot is, it doesn't know your project's goals, business logic, or long-term vision. Trust your instincts when something feels off. Be sure to:

- Structure your code your way.
- Name variables meaningfully.

- Refactor when Copilot suggestions feel overly generic.

Important

Don’t treat Copilot as an auto-complete engine for your whole project. Use it to support your thinking rather than as a substitute for it. You own the architecture, the trade-offs, and the outcome. Copilot writes code; you write software.

Best practices for using GitHub Copilot in your workflow

Use this list to get the most value from GitHub Copilot while maintaining code quality, security, and control.

1. Write clear prompts.
   - Use specific and descriptive comments (e.g., # Function to sort a list of products by price descending).
   - Break down complex tasks into step-by-step instructions.
   - Include contextual hints (e.g., expected input/output or technology used).
   - Avoid vague or generic prompts like # do something.
2. Use Copilot as a collaborator, not as a crutch.
   - Treat suggestions like input from a junior developer; review them before using them.
   - Don’t rely on Copilot for business logic or domain-specific decisions.
   - Maintain your own naming conventions and code structure.
   - Edit and refactor Copilot's output to suit your style and architecture.

3. Review and validate suggestions.
   - Check for logic errors, edge cases, and unexpected behaviors.
   - Verify security (e.g., no hardcoded secrets or missing validation).
   - Ensure compatibility with project requirements, libraries, or frameworks.
   - Run tests after accepting suggestions. Never skip verification.
4. Test and document as you go.
   - Use prompts like # write unit tests for this function to generate tests early.
   - Let Copilot assist in generating docstrings, README sections, and comments.
   - Keep your code well documented to improve future suggestions.
5. Customize Copilot behavior.
   - Configure file-specific suggestion settings (e.g., disable in .json or .md).
   - Use keyboard shortcuts to trigger, cycle, or dismiss suggestions (e.g., Alt+/, Tab, Esc).
   - Explore Copilot Labs or Chat for enhanced productivity.
6. Use secure and ethical practices.
   - Avoid accepting code that resembles license-restricted content.
   - Be cautious with auto-generated SQL, authentication, or system commands.
   - Always run Copilot code through manual and automated reviews.
7. Reflect and improve.
   - Keep a snippet library of effective prompts you use repeatedly.

- Track time-saving patterns or mistakes to avoid.
- Share prompt templates and learnings with your team.

Understand the role of AI across industries

GitHub Copilot is not limited to any single language, framework, or job title, and it is useful across industries. Whether it's helping healthcare teams analyze clinical data or enabling fintech startups to speed up product development, Copilot adapts to a wide range of technical and business environments.

In this section, you'll explore how Copilot is being used in specific sectors, from startups to large enterprises, and how its value shifts depending on goals like productivity, compliance, or innovation.

Healthcare and life sciences

Developers in the healthcare and life sciences sector face a unique combination of technical, ethical, and regulatory challenges. Applications often involve handling sensitive patient data, supporting clinical workflows, and generating insights from biomedical information. Accuracy, security, and auditability are not optional; they're mandatory.

GitHub Copilot can significantly speed up the development process in this domain by assisting with routine data tasks, scaffolding APIs, and generating compliant documentation structures. However, because healthcare software is highly regulated, Copilot-generated output must be carefully validated.

Let's look at a few ways developers are using Copilot in healthcare settings.

Clean and analyze patient data using pandas and NumPy

Clinical datasets are often messy and inconsistent. Copilot can help automate data cleaning steps, such as handling missing values, normalizing medical measurements, and flagging outliers.

Prompt:

```
# Normalize blood pressure readings in a clinical DataFrame
```

Copilot-generated code:

This prompt clearly specifies the task, enabling Copilot to return a complete and concise solution. Such a solution would be especially helpful when working with hundreds of variables in large electronic health record (EHR) datasets.

```
# Normalize blood pressure readings in a clinical DataFrame
# Normalize blood pressure readings in a clinical DataFrame
def normalize_blood_pressure(df: pd.DataFrame) -> pd.DataFrame:
    df['blood_pressure'] = (df['blood_pressure'] - df['blood_pressure'].mean()
    return df
```

A Python function auto-generated by GitHub Copilot to normalize blood pressure readings in a clinical DataFrame by subtracting the column mean

Build and test APIs for secure patient portals

Copilot can help write API routes for common patient-facing features, such as viewing appointments, updating records, or downloading medical reports.

Prompt:

```
# Flask route to return patient medical summary securely
```

Copilot-generated code:

Because the prompt clearly defines the goal, Copilot accurately generates the code, including proper decorators and secure identity handling.

> **Tip**
>
> To guide Copilot toward more domain-appropriate code, add specific details in your prompt, like "use JWT for authentication" or "return FHIR-compliant data."

```python
# Flask route to return patient medical summary securely
from flask import Flask, jsonify, request
from flask_jwt_extended import jwt_required, get_jwt_identity
from your_database_module import get_patient_summary

app = Flask(__name__)

@app.route('/patient/summary', methods=['GET'])
@jwt_required()
def patient_summary():
    current_user = get_jwt_identity()
    summary = get_patient_summary(current_user['id'])
    return jsonify(summary), 200
```

A Flask route created by GitHub Copilot to securely return a patient's medical summary, using JWT-based authentication

Generate documentation for regulatory compliance

Healthcare applications must be well documented to support audits and compliance. Copilot can assist by generating docstrings, logging templates, and README content aligned with documentation standards.

Prompt:

```
"""Return anonymized patient summary for internal reporting."
```

Copilot-generated code:

“”””

```
    dict: Anonymized patient summary including diagnosis and trea
plan.
Notes:
    This function strips identifiers in accordance with HIPAA Saf
Harbor rules.
"""
```

You can also prompt for markdown blocks in internal reports or notebooks.

Scaffold notebooks for diagnostics and prediction

Data scientists working in healthcare often use Jupyter notebooks to run diagnostics, generate insights, and visualize clinical trends. Copilot helps scaffold analysis cells with visualization and ML setup.

Python prompt:

```
# Visualize glucose levels over time and flag abnormal readin
```

Copilot-generated code:

```
import matplotlib.pyplot as plt
plt.plot(df["timestamp"], df["glucose"])
plt.axhline(y=140, color='r', linestyle='--', label="Threshold")
plt.title("Patient Glucose Levels")
plt.xlabel("Time")
```

```
plt.ylabel("Glucose (mg/dL)")
plt.legend()
plt.show()
```

As you can see, Copilot can help researchers prototype ideas faster and maintain visual consistency.

> **Important**
>
> Copilot can generate useful syntax and boilerplate for healthcare workflows, but it has no awareness of legal, ethical, or clinical best practices. Always review Copilot suggestions for compliance with HIPAA, GDPR, and your institution's security guidelines. Code should be audited for privacy risks before being used in production or shared with stakeholders.

Finance and fintech

In the world of finance and fintech, speed, precision, and compliance go hand in hand. Whether processing transactions, generating financial reports, or analyzing risk data, developers often work under strict regulations and tight deadlines.

GitHub Copilot is valuable in this domain because it can reduce human time spent on repetitive coding tasks; it can help scaffold secure APIs, write SQL queries for financial insights, and automate compliance-friendly logic. But as with healthcare, developers must apply domain knowledge to ensure that suggestions meet institutional, legal, and audit standards. Let's look at some examples of how Copilot supports developers in the finance and fintech worlds.

Build secure APIs for account and transaction management

APIs form the backbone of modern banking and fintech apps. Copilot can help generate secure route handlers, especially for common account-related operations like balance checks, transactions, or authentication.

Prompt:

```
# FastAPI route to return account balance securely
```

Copilot-generated code:

```
from fastapi import FastAPI, Depends
from fastapi.security import OAuth2PasswordBearer
app = FastAPI()
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
@app.get("/account/balance")
async def get_balance(token: str = Depends(oauth2_scheme)):
    # Placeholder for auth logic and balance fetch
    return {"balance": 1542.75}
```

This code helps bootstrap secure endpoints that you can later connect to internal systems or payment processors.

Write SQL queries for fraud detection or financial reporting

Finance teams rely heavily on SQL queries to monitor suspicious activity, generate dashboards, and meet reporting obligations.

Prompt:

```
-- SQL query to list all transactions over $10,000 in the pas
```

Copilot-generated code:

```
SELECT *
FROM transactions
WHERE amount > 10000
  AND transaction_date >= NOW() - INTERVAL '1 day';
```

This code speeds up routine report writing, especially when integrated into analytics platforms or Python ETL scripts.

Tip

Use Copilot to generate dynamic query strings within code as well. Here's an example:

```
# SQLAlchemy query to count users who exceeded transactio
```

Scaffold compliance-related logic and audit logs

Copilot can help generate logging and tracking systems to support internal audits and legal compliance (e.g., SOC 2, PCI DSS).

Prompt:

```
# Log user login attempts with timestamp and IP address
```

Copilot-generated code:

```
import logging
from datetime import datetime
def log_login(user_id, ip_address):
    logging.info(f"Login attempt by {user_id} from {ip_address} a
{datetime.now()}")
```

These kinds of logs are critical for audit trails and suspicious activity reports.

Build risk models and scoring systems

Developers and data scientists working in risk management and fraud prevention can use Copilot to accelerate model scaffolding, especially for prototyping.

Prompt:

```
# Train a logistic regression model to detect high-risk trans
```

Copilot-generated code:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X = df.drop("label", axis=1)
y = df["label"]

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

model = LogisticRegression()
model.fit(X_train, y_train)
```

This code enables a team to quickly build a baseline that it can then refine with domain-specific features or ensembles.

> **Important**
>
> In financial systems, accuracy and security are critical. Copilot cannot understand regulatory frameworks like PCI DSS, GDPR, or Basel III. Always review generated code with an eye on data sensitivity, authorization, logging policies, and institutional risk guidelines.

Education

The education sector, particularly computer science education and educational technology (edtech), is one that is a natural fit for GitHub Copilot. Educators, instructional designers, and edtech developers all benefit from faster code creation, clear examples, and consistent formatting.

Whether you're building a learning platform or teaching students how to write algorithms, Copilot serves as both a productivity tool and an educational resource.

Help educators generate sample code and assignments

Teachers and instructors often need clean, simple examples to demonstrate programming concepts. Copilot can generate variations of classic problems and beginner-friendly code that can be used in lectures or courseware.

Prompt:

```
# Example function to demonstrate recursion in Python
```

Copilot-generated code:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

You can ask for alternatives like Fibonacci, binary search, or tree traversal with similar prompts.

Tip

Prompt Copilot for variations:

```
# Recursive vs. iterative example of finding max in list
```

This helps students compare coding styles and develop deeper understanding.

Help students explore and practice programming logic

Students working on assignments often benefit more from small nudges than from full answers. Copilot can offer suggestions when students write clear function signatures or structured comments.

Prompt:

```
# Function to check if a number is prime
```

Copilot-generated code:

Copilot may suggest the entire function or offer enough of a structure for the student to complete it. This reduces frustration while encouraging learning by doing. In this example, the prompt is clearly written, resulting in a correct and efficient solution that includes type hints and proper logic for primality testing.

```python
# Function to check if a number is prime
# Function to check if a number is prime
def is_prime(n: int) -> bool:
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

A function generated by GitHub Copilot that checks whether a number is prime

Build and maintain educational platforms

Developers working on learning management systems (LMSs), assessment platforms, and coding sandboxes can use Copilot to:

- Scaffold REST APIs (e.g., for user, course, or submission management).
- Generate validation logic and frontend form handling.
- Write configuration scripts and test cases for platform deployment.

Prompt:

```
// Express.js route to submit quiz answers
```

Copilot-generated code:

```
app.post("/api/submit", (req, res) => {
  const { userId, quizId, answers } = req.body;
  // Save submission to database...
  res.status(200).json({ message: "Submission received." });
});
```

Generate documentation and guides

Copilot can assist with writing instructional content like README files for student projects or docstrings for lab templates.

Prompt:

```
# Instructions for students to run the Python grading script
```

Copilot-generated code:

GitHub Copilot automatically generates detailed markdown instructions for students to run a Python grading script, including package installation and execution commands.

Copilot can support educators/trainers who want to keep materials updated without rewriting boilerplate instructions repeatedly.

See Also

In Chapter 9: Avoiding common pitfalls with GitHub Copilot, you'll learn how to avoid common mistakes when using Copilot in real-world settings and apply techniques to validate AI-generated code in educational and production environments.

````
# Instructions for students to run the Python grading script
1. Ensure you have Python installed on your machine.
2. Install the required packages by running:
   ```
   pip install -r requirements.txt
   ```
3. Run the grading script using the command:
   ```
   python grade.py
   ```
4. Follow any additional instructions provided in the script output.
````

A GitHub Copilot-generated instructional markdown snippet on how to run a Python grading script

Manufacturing and logistics

In manufacturing and logistics, developers often work with systems that manage inventory, track shipments, and analyze supply chain data. These systems require accurate, high-volume processing, often involving sensor data, time-based metrics, or integrations with hardware and enterprise resource planning (ERP) platforms.

Copilot helps by accelerating scripting tasks, generating logic for calculations and condition handling, and scaffolding interfaces for monitoring or automation. While the workflows may be more operational, Copilot is a time-saving partner in both production environments and internal tool development.

Automate inventory tracking and stock validation

Developers frequently write scripts to track stock levels, compare reorder points, or calculate demand. Copilot simplifies these calculations.

Prompt:

```
# Function to check if any products are below their reorder l
```

Copilot-generated code:

```
def check_reorder(products):
    return [p for p in products if p["stock"] < p["reorder_level"
```

This code helps warehouse and inventory systems respond dynamically to supply conditions.

Estimate delivery windows and logistics delays

Shipping systems depend on real-time calculations to estimate delivery times and suggest contingencies. Copilot helps write business rules and logic for these systems.

Prompt:

```
# Calculate expected delivery date with a 3-day buffer
```

Copilot-generated code:

```
from datetime import datetime, timedelta

def expected_delivery(order_date):
    return order_date + timedelta(days=3)
```

You can add specificity to your prompt to have Copilot include business days or external API calls (e.g., traffic or weather APIs).

Process IoT sensor data from machinery or vehicles

Industrial systems often ingest timestamped sensor data from machines or vehicles. Copilot can assist in parsing, filtering, and transforming this data for monitoring and alerting.

Prompt:

```
# Calculate average temperature from IoT readings ignoring nu
```

Copilot-generated code:

```
def average_temp(readings):
    valid = [r for r in readings if r is not None]
    return sum(valid) / len(valid) if valid else None
```

This code pattern is a common pattern in predictive maintenance and telemetry systems.

Generate dashboards and monitoring logic

You can use Copilot to scaffold backend code for dashboards or frontend logic to track machine status, order flow, or transport metrics.

Prompt:

```
// React component to display current vehicle status
```

Copilot might generate a basic card layout or status box that integrates with your existing UI.

How to explore Copilot use cases in your industry

While this section highlights healthcare, finance, education, and manufacturing, GitHub Copilot can be equally powerful in other domains, such as legal tech, retail, gaming, and nonprofits. If you’re working in a niche industry, start by identifying common repetitive coding tasks or compliance requirements in your field. Then prompt Copilot with comments that simulate your real-world scenarios.

For example, if you’re in legal tech, try a prompt like this to get Copilot to generate boilerplate that you can customize for privacy rules:

Create a function to redact sensitive client names from documents.

Experiment with your domain-specific language in prompts. Copilot adapts well to terms like “SKU,” “case number,” or “shipment ID,” and it provides better suggestion accuracy when your prompt includes terms like these.

Apply practical examples of AI in your projects

Now that you’ve seen how GitHub Copilot works across industries and workflows, this section walks you through simple but practical examples you can try directly in your own environment. These examples are not just for testing Copilot; they’re for building your confidence and helping you discover patterns you can reuse.

You’ll explore examples from web development, data analysis, and backend scripting. Each example is structured with a prompt, a possible Copilot suggestion, and a small challenge to extend the idea further.

Web project example: Contact form handler

Goal: Build a simple backend route to receive form input from a contact page.

Prompt:

```
# Flask route to accept POST data from a contact form and ret
```

Copilot-generated code:

```
from flask import Flask, request, jsonify
app = Flask(__name__)
@app.route("/contact", methods=["POST"])
def contact():
    data = request.json
    name = data.get("name")
    email = data.get("email")
    message = data.get("message")
    return jsonify({"status": "success", "message": "Message received"})
```

Try this: Add email format validation and return a custom error for invalid input.

Data analysis example: Clean a DataFrame

Goal: Clean missing data and prepare it for visualization.

Prompt:

```
# Drop rows with null values and convert 'order_date' column
```

Copilot-generated code:

```
df = df.dropna()
df["order_date"] = pd.to_datetime(df["order_date"])
```

Try this: Add a line to extract the month from the order_date column.

Backend logic example: Calculate user discounts

Goal: Create a function to calculate discounted prices for users, based on tier.

Prompt:

```
# Function to apply discount based on user tier (basic: 5%, p
```

Copilot-generated code:

```
def apply_discount(price, tier):
    discounts = {"basic": 0.05, "premium": 0.15}
    discount = discounts.get(tier, 0)
    return price * (1 - discount)
```

Try this: Add logic to raise an error if the price is negative.

DevOps example: Bash script for file cleanup

Goal: Write a shell script to delete log files that are more than seven days old.

Prompt:

```
# Bash script to delete .log files older than 7 days in /var/
```

Copilot-generated code:

```
find /var/logs -name "*.log" -type f -mtime +7 -exec rm {}
```

Try this: Add a dry-run option that prints the files instead of deleting them.

Testing example: Write unit tests

Goal: Generate tests for a function that calculates totals.

Prompt:

```
# Write pytest tests for calculate_total(price, tax_rate, dis
```

Copilot-generated code:

```
def test_calculate_total():
    assert calculate_total(100, 0.1, 10) == 100
    assert calculate_total(200, 0.2, 0) == 240
```

Try this: Add a test for negative tax rate and verify proper handling.

> **Tip**
>
> Use these examples as templates. Save your favorite Copilot prompts in a personal snippet library for reuse across projects.

> **See Also**
>
> In Chapter 10: Exploring the future of AI in software development, you'll explore how GitHub Copilot and other AI tools are shaping the future of software development and how developers can prepare for evolving workflows, new skills, and AI-integrated team dynamics.

Build a prompt library for reusability

As you explore Copilot-generated code, start collecting the prompts that work well for you. By doing so, you can create a prompt library that can serve as a go-to resource for future projects and teammates.

Organize your prompts by category (e.g., forms, APIs, DataFrames, tests) and store them in a simple .md or .txt file. You can even version-control your library in GitHub.

Refine your favorite prompts over time. For example, evolve this prompt:

Function to calculate discount into

to the following:

Function to calculate dynamic discount based on customer tier and cart total

This type of refinement saves time and also helps train Copilot more effectively in your coding style and domain context.

Reflect on Copilot's impact on teams

GitHub Copilot is becoming more widely adopted, and its influence isn't limited to individual developers; it's changing how teams work together. From faster onboarding to more efficient code reviews and fewer bottlenecks in delivery, teams are discovering both benefits and challenges as they integrate AI assistance into their workflows.

This section reflects on how Copilot is reshaping collaboration, productivity, communication, and development culture at the team level.

Improved collaboration and shared understanding

Copilot makes it easier for developers to express intent early by writing clear comments, function names, and TODOs. When teammates review pull requests with Copilot-assisted code, they often encounter cleaner, better-documented logic and fewer inconsistencies, making feedback loops faster and more focused.

> **Tip**
>
> Teams can create and share prompt patterns or templates (e.g., for tests or API handlers) to ensure consistent code across contributors.

Faster onboarding and mentorship

New developers often take time to adapt to team coding standards and project structure. With Copilot, junior developers can get unstuck more quickly when writing unfamiliar logic. In addition, they can learn by example how to format code consistently, and they can explore solutions independently, without waiting on seniors.

Copilot doesn’t replace mentorship, but it reduces friction during the learning curve.

Reduced time to implementation

With Copilot assisting in boilerplate, validation, and even deployment scripts, teams spend less time on repetitive coding and more time on architecture, business logic, and testing. This improves delivery speed, especially in agile environments.

For example, a developer might spend 30 minutes writing a RESTful API handler from scratch. By using Copilot, a developer can get the same handler scaffolded in seconds and spend the saved time refining error handling and edge cases.

AI as a second reviewer

Some teams use Copilot to sanity-check logic even after the code is written. For example, they might use a prompt like this:

```
# Write a test for this function to validate logic
```

Or they might paste legacy code and use this prompt:

```
# Suggest improvements or refactor this function
```

Developers can use Copilot to get a quick "peer check" before formal review.

> **Important**
>
> Teams should not rely on Copilot alone for review. While it’s helpful for spotting patterns or writing tests, only humans can verify business requirements, edge case coverage, and architectural alignment.

Changing developer roles and team dynamics

As AI takes on more repetitive work, developers can:

- Focus more on design, testing, and problem solving.
- Spend less time fixing syntax errors.
- Write cleaner code more quickly—although they must stay accountable for quality.

The use of Copilot is shifting the focus of teamwork from who writes code to how the code meets goals, which is a healthy evolution for most agile teams.

See Also

In Chapter 10: Exploring the future of AI in software development, you'll explore how GitHub Copilot and similar tools are shaping the future of development, what skills developers will need, and how teams can stay adaptive as AI becomes a coding collaborator.

Skills review

In this chapter you learned how to:

- Identify real-world Copilot use cases across various domains.
- Integrate Copilot into your end-to-end development workflow.
- Understand how Copilot adapts to industry-specific challenges.
- Apply Copilot prompts to web, data, backend, and automation projects.
- Reflect on how Copilot affects collaboration and team productivity.

Practice tasks

Complete these tasks in your preferred Python development environment with GitHub Copilot enabled to reinforce what you learned in this chapter.

Use an integrated development environment (IDE) such as Visual Studio Code with GitHub Copilot enabled and signed in. The following practice tasks use Python as the language. You can adapt them to other supported languages as needed.

Discover real-world Copilot use cases

Use Copilot to generate useful code snippets across multiple domains and web backends to understand where it can help in real-world scenarios:

1. In Visual Studio Code or another IDE that supports GitHub Copilot, create a new folder with three empty files:
   - web_route.py
   - backend_logic.py
   - data_cleaning.py
2. Open web_route.py. On the first line, type this prompt:.

   ```
   # Flask route to return product details as JSON
   ```

   Let Copilot generate the rest. Review and modify the code as needed.
3. Open backend_logic.py. Add this comment:

   ```
   # Function to apply a loyalty discount to customer orders
   ```

   Let Copilot generate the logic. Add a few test calls or validations.
4. Open data_cleaning.py. Add this comment:

   ```
   # Remove null rows and convert date column to datetime in
   ```

   Let Copilot generate the code.

Explore AI workflows

To experience how GitHub Copilot can assist throughout a full development lifecycle, from planning to deployment, perform the following tasks:

1. Plan: In a new file, write a comment like this:

```
# TODO: Build a CLI tool to convert temperature from Cels
```

   Let Copilot suggest the function and file structure. Add one or two additional TODOs and observe how Copilot supports you in outlining a plan.

2. Code: Below your TODOs, start writing the core logic. Use a prompt like this:

```
# Function to convert Celsius to Fahrenheit
```

   Let Copilot generate the function and enhance it with input validation.

3. Test: In a new cell or file, add this prompt:

```
# Write pytest tests for convert_temperature()
```

   Copilot will likely generate two or three relevant unit tests. Review and refine them.

4. Document: At the top of your main function, write this prompt:

```
"""Convert temperature from Celsius to Fahrenheit."""
```

   Let Copilot expand the docstring. Try prompting it to create a README section, too.

5. Deploy (optional): Ask Copilot to help you create a deployment configuration by using this prompt:

```
# GitHub Actions workflow to run tests on push
```

Understand the role of AI across industries

To use GitHub Copilot to simulate practical coding tasks that are specific to the healthcare and finance domains, perform the following tasks:

1. To clean and prepare clinical data for analysis, create the file dataset.py in Visual Studio Code.

2. Enter the following prompt in the file:

```
# Create a pandas DataFrame with patient_id, blood_pressu
  including some missing values
```

> **Tip**
>
> You can modify the prompt to be more specific, such as by changing it to the following:

```
# Simulate clinical data with missing blood pressure and
```

3. Add the following prompt to the file:

```
# Drop rows with missing values and convert date column t
```

Let Copilot generate the cleaning logic.

4. Extend the logic by prompting Copilot to normalize the blood_pressure column using MinMaxScaler.

Apply practical examples of AI in your projects

To practice using Copilot to solve hands-on, real-world coding problems in your own environment, perform the following tasks:

1. To build a reusable data cleaner, create a file with the .py extension in Visual Studio Code.
2. Type the following prompt in the file:

```
# Function to drop null values and convert 'date' column
```

   Let Copilot generate the function.

3. Apply the new function to a sample DataFrame that has at least three columns and some missing values.
4. Extend the function to:

   a. Remove duplicates.

   b. Normalize a numeric column (e.g., "amount" or "score").

Reflect on Copilot’s impact on teams

Think critically about how GitHub Copilot changes collaboration, code review, onboarding, and productivity within a team setting. Try applying Copilot in a simulated team workflow:

1. Simulate a team coding session:

a. Open a file and write a comment like this in it:

```
# TODO: Add an endpoint to return the current user's prof.
```

Let Copilot generate the code.

b. Imagine that you're reviewing this code in a pull request. Determine:

- Is it secure?
- Does it follow project conventions?
- Would a teammate understand it clearly?

c. Write a short review comment (two or three lines) as if you were giving feedback to a teammate.

2. Compare Copilot's suggestion with your own:

a. Manually write a short function to accomplish a particular task (e.g., def is_palindrome(word):).

b. Prompt Copilot to generate a function that accomplishes the same task.

c. Compare the two functions and determine:

- Which one is cleaner?
- Are there edge cases missing in either?

d. Note one improvement you'd make to Copilot's version.

3. Identify a workflow bottleneck:

a. Think about your last project or team experience. Identify one area where repetitive code slowed the team down.

b. Describe (in three to five lines) how Copilot could help reduce this bottleneck. Be specific (e.g., test writing, deployment configuration, documentation).

9

Avoiding common pitfalls with GitHub Copilot

In this chapter

- Spot and fix common Copilot mistakes
- Understand why Copilot mistakes happen
- Fix and refine AI-generated code
- Guide Copilot effectively
- Practice real-world examples

Practice files

There are no practice files for this chapter.

Using GitHub Copilot can feel like magic: Code appears as you type, suggestions come in fast, and your workflow speeds up. But as with any other powerful tool, it's easy to misuse Copilot if you're not careful.

In this chapter, we'll walk through the common mistakes developers make when relying too heavily on Copilot or not using it thoughtfully. You'll learn why these issues happen, how to recognize them early, and, most importantly, how to avoid them.

We won't just talk about what not to do. You'll also pick up simple techniques to help Copilot give you better, safer, and more readable code. Whether you're a beginner or an experienced developer, these practical tips will help you get the most out of Copilot without letting it take the wheel completely.

Spot and fix common Copilot mistakes

GitHub Copilot is incredibly helpful, but it's not perfect. When you're coding quickly, it's easy to miss when something Copilot suggests isn't quite right. In some cases, it might guess the wrong intent, use outdated patterns, or simply generate code that doesn't work.

Let's look at some of the mistakes developers most commonly make when using Copilot and how to recognize them before they sneak into your codebase.

Blindly accepting suggestions

It's tempting to get a Copilot suggestion, hit Tab, and move on. After all, Copilot just filled in a whole function—why not use it?

This is the number-one mistake developers make: accepting code without reviewing it. Copilot might generate code that looks correct but that contains subtle logic errors, performance issues, or security flaws.

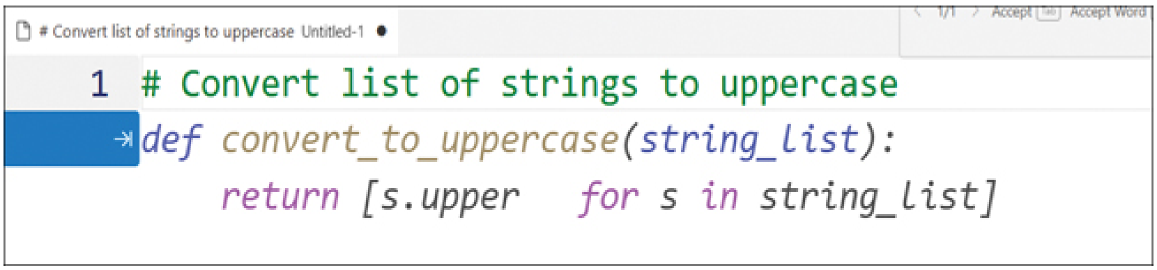
Python Prompt:

```
# Convert list of strings to uppercase
```

Copilot-generated code:

```
def convert_to_uppercase(string_list):
    return [s.upper for s in string_list]
```

This code looks fine, but it’s actually wrong. The function convert_to_uppercase(string_list) uses a list comprehension to return [s.upper for s in string_list], but it incorrectly references the upper method instead of calling it. The correct usage should be s.upper() to convert each string to uppercase.



A Python function that attempts to convert a list of strings to uppercase using a list comprehension

Tip

Always test the code Copilot creates and trust your instincts. If a suggestion looks unusually simple or suspiciously effortless for a complex task, it probably needs a closer review.

Overusing Copilot

Copilot is great at creating repetitive code, helper functions, and boilerplate. But it’s not meant to design your app architecture or write critical business logic. If you rely on it too much, you might end up with code that works but that you don’t really understand.

This matters because you’re the developer, and Copilot doesn't know your app's goals, user needs, or edge cases. If you don't fully understand the code Copilot writes, debugging it later will be a nightmare.

Tip

Use Copilot for support, not shortcuts. Think of it as a junior developer: helpful but still learning.

Letting Copilot guess without enough context

Copilot works best when it understands what you're trying to do. If your comment or code is vague, Copilot may take a wild guess, and sometimes it's completely off.

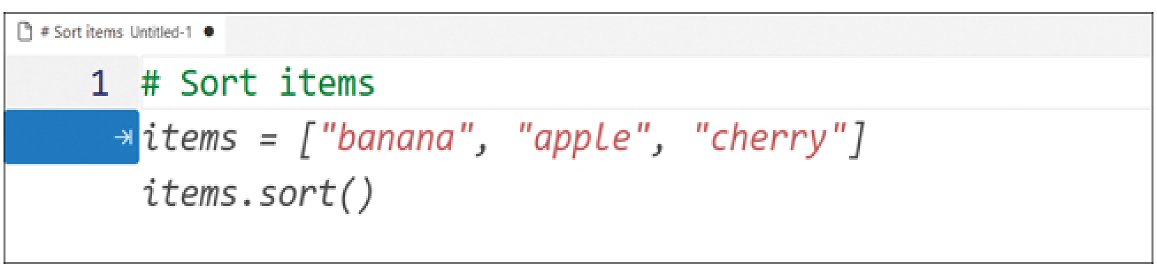
Vague prompt:

```
#Sort items
```

Copilot-suggested code:

```
items = ["banana", "apple", "cherry"]
items.sort()
```

Because the prompt is vague, Copilot doesn't know if you want to sort alphabetically, by case, by length, or in ascending or descending order.



A Python code snippet that creates a list of strings and sorts them using the sort() method.

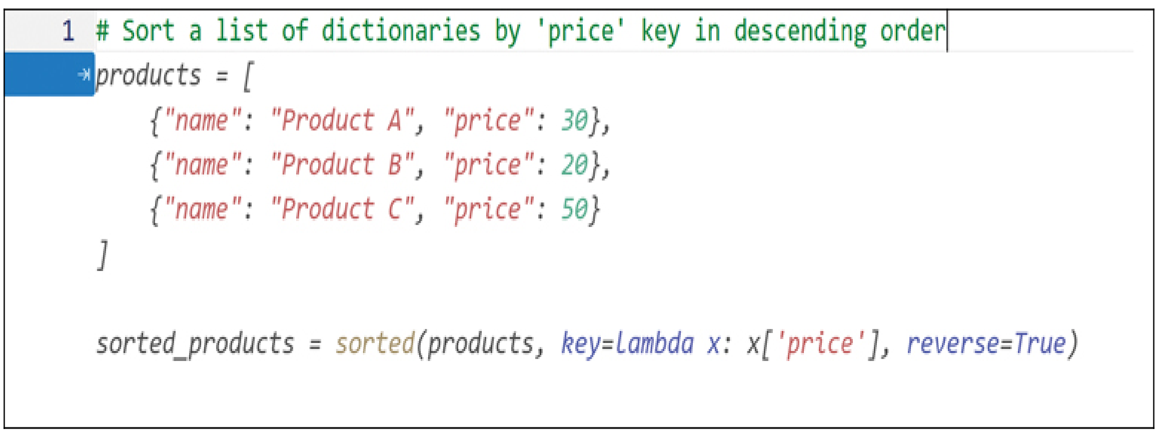
Copilot doesn't know whether your prompt means sorting alphabetically, by date, by size, or in ascending or descending order. Copilot can't read your mind.

Improved prompt:

```
# Sort a list of dictionaries by 'price' key in descending or
```

Copilot-suggested code:

```
products = [
    {"name": "Product A", "price": 30},
    {"name": "Product B", "price": 20},
    {"name": "Product C", "price": 50}
]

sorted_products = sorted(products, key=lambda x: x['price'],
reverse=True)
```



A Python code snippet that sorts a list of dictionaries by the 'price' key, in descending order, using the sorted() function and a lambda expression.

Tip

Be clear in your prompts. The more specific your comments, the better Copilot’s suggestions.

Failing to check for edge cases or validation

Copilot might give you the "happy path" solution—the scenario where everything goes right. But real-world apps need to handle things going wrong, too.

For example, Copilot might generate a function that assumes a list always has values or that a database call never fails. It's important to consider what happens if a list is empty or if an error occurs—and adjust the code as needed.

Forgetting to match your team's style

Copilot might write code in a style that's different from what your team prefers. That could mean different variable naming, indentation, or function structure.

Consistency matters—especially when you're working as part of a team. Even if Copilot's suggestions are technically correct, mismatched code styles can lead to messy pull requests and review friction.

Tip

Customize Copilot settings or take a minute to reformat the code. Your team (and future you) will thank you.

Important

You are still the developer. Copilot assists, but it doesn't know your goals or architecture.

See Also

To learn how to prompt Copilot effectively, refer to Chapter 3: Writing effective prompts for AI-powered coding.

Spotting common Copilot mistakes

Even helpful code can hide subtle errors. Use this list whenever you accept suggestions to avoid common Copilot pitfalls. Before accepting a Copilot suggestion, consider:

- Did I read and understand the entire piece of code that Copilot suggested?
- Are function calls properly written (e.g., upper() rather than upper)?
- Does the logic actually match my goal, or does it just "look right"?
- What edge cases might Copilot have skipped?
- Does this code follow my team's naming and formatting conventions?

Understand why Copilot mistakes happen

It's important to spot mistakes from GitHub Copilot. But it's even more valuable to understand why they happen. When you know what causes these slipups—whether it's Copilot's limitations or your own coding habits—you'll be able to avoid them more easily and work with Copilot more effectively.

Let's break down the reasons behind the most common issues and explore what's happening.

Copilot doesn't understand your goal; it predicts patterns

At its core, Copilot is powered by a machine learning model that is trained on public code. It's not trying to “solve” your problem as a human would. Rather, it’s simply guessing what should come next, based on patterns it has seen.

For example, say that millions of people have written the following code on GitHub:

```
for i in range(10):
    print(i)
```

The loop prints each integer from 0 to 9 on a new line using the built-in range() function and print(). Copilot learns that this is a common pattern and may suggest it when you write something similar. But it doesn’t understand why you need a loop or what you're looping over.

```
for i in range(10):
    print(i)
```

A basic Python for loop that prints numbers from 0 to 9 using range(10) and the print() function

So, if you enter the following prompt:

```
# loop through data and print it
```

Copilot might assume that you mean numbers, strings, or even something entirely unrelated. It’s not being lazy; it’s just making a statistically likely guess.

Important

Copilot predicts based on training data. It doesn't "think" or understand your business logic. To get accurate results, you need to give Copilot clear directions.

Copilot makes it easy to move fast

Let’s be honest: Using Copilot feels great. It saves keystrokes, handles repetition for you, and helps you finish tasks quickly. But that speed can create overconfidence. It’s easy to fall into a habit of accepting suggestions without stopping to think: “Is this actually what I need?”

When you're on a deadline or juggling multiple tasks, reviewing every line Copilot suggests might feel like an unnecessary slowdown. But skipping those checks can lead to errors that take longer to fix later.

JavaScript prompt:

```
// Check if user is admin
```

Copilot-suggested code:

```
if (user.role === "admin") {
    // do something
}
```

This looks fine until you realize that your actual data structure is user.permissions.isAdmin. If you accept the suggestion without noticing, you end up with a silent bug.

You're not giving enough context

Copilot’s suggestions are only as good as the information it’s working with. If your file is mostly empty, your comments are vague, or the surrounding code is unrelated, Copilot might go in the wrong direction.

Imagine trying to help someone complete a sentence when you’ve heard only one word. That’s what Copilot experiences when you don't give it enough clues.

Python prompt:

```
# fetch data
```

Copilot might assume you want an API call, a file read, or a database query. Who knows?

Copilot-suggested code:

```
import requests
```

This code snippet imports the requests library, suggesting an upcoming HTTP request operation. However, no actual request or usage of the library is shown; the code lacks enough context or implementation to show how data will be fetched.

A Python code snippet that begins with a comment indicating the intention to fetch data and imports the requests library

Improved Python prompt:

```
# Fetch data from external API and return JSON response
```

Copilot-suggested code:

```
def fetch_data(url, headers=None, params=None):
    try:
        response = requests.get(url, headers=headers, params=para
        response.raise_for_status()  # Raise an error for HTTP er
        return response.json()
    except requests.RequestException as e:
        print(f"Error fetching data: {e}")
        return {"error": str(e)}
```

The improved prompt gives Copilot something to work with, and it creates more useful code.

> **Tip**
>
> Write comments as though you're giving instructions to a helpful intern. Be specific. Tell it what you want and possibly how you want it.

```
# Fetch data from external API and return JSON response
def fetch_data(url, headers=None, params=None):
    try:
        response = requests.get(url, headers=headers, params=params)
        response.raise_for_status()  # Raise an error for HTTP errors
        return response.json()
    except requests.RequestException as e:
        print(f"Error fetching data: {e}")
        return {"error": str(e)}
```

A Python function that fetches data from an external API and returns a JSON response

Prompting is still a new skill for many developers

We’re used to writing code directly, not describing what we want the code to do. But Copilot responds better to intention than to syntax. It needs to understand your thought process before it can generate the right output.

It’s normal to write comments that feel obvious to you but that make no sense to Copilot. You’re learning how to work with a new kind of tool—one that thrives on clear, high-level guidance rather than detailed instructions. And that takes practice.

Try to think of prompting as part of the coding process rather than as something separate. If you can clearly describe in a comment what you're trying to do, chances are Copilot will give you better code, and you'll also have a better understanding of what you’re trying to accomplish.

Copilot doesn’t know your project’s architecture, data, or rules

Copilot doesn’t have access to your business logic, internal libraries, or company security practices. It can only use what’s in the current file and, sometimes, nearby files. This means you need to be careful when using it in critical parts of your application.

For example, you may be working in a health app where certain data must be encrypted, but Copilot has no way of knowing that. It might suggest storing plain text values or using insecure libraries.

> **Important**
>
> Copilot doesn’t know your data rules, security policies, or user needs. Always verify that suggestions follow your project’s standards and constraints.

Understanding why Copilot makes mistakes helps you avoid frustration and saves time. You’ll stop expecting it to be perfect and start using it like a smart assistant that needs a bit of direction. The more intentional you are about how you use Copilot, the more valuable it becomes.

Copilot makes mistakes with vague prompts. The clarity of a prompt plays a significant role in determining the accuracy and usefulness of Copilot's output. Vague instructions often lead to generic or incorrect results, while specific and well-structured prompts guide Copilot toward producing the desired code. The following table compares different prompts, the output Copilot will likely generate for each one, and notes on why some prompts perform better than others.

Prompt	Copilot's likely output	Notes
sort list	Generic ascending sort	Too vague; could be misinterpreted
sort list of names	Alphabetical sort	Slightly better but still ambiguous
sort list of students by score descending	Sort of dictionaries by score with reverse=True	Clear and effective prompt
filter items	May use various filters	Unclear; needs more context
filter orders placed in last 30 days	Correct use of datetime and filters	Highly contextual and precise

Understanding why mistakes happen

To fix mistakes, you have to know where they come from. This list helps you stay mindful of how Copilot works and where it might go wrong.

While reviewing Copilot code, consider:

- Am I expecting Copilot to "understand" when it only predicts?
- Did I write a vague comment or give too little context?

- Am I trusting the code because it saves time, not because it's correct?
- Did I double-check that the code fits with my project's goals and data rules?
- Am I thinking like a developer or letting Copilot think for me

See Also

For more on how Copilot uses context from your IDE setup and file structure to generate suggestions, refer to Chapter 2: Setting up GitHub Copilot for your development workflow.

Fix and refine AI-generated code

One of the biggest advantages of using GitHub Copilot is speed. It helps you move fast by generating chunks of code, completing functions, and suggesting solutions. But speed without refinement can lead to messy, inconsistent, or even insecure code.

Learning how to refine the suggestions Copilot gives you is a must-have skill—not just for writing better code but for growing as a developer. This section is all about turning rough AI-generated drafts into clean, readable, and maintainable code.

Copilot can help you build faster, but it won't always generate the best version of a solution. It might use older syntax, write long functions, or skip important edge cases. Copilot's first suggestion, like a rough draft in writing, is rarely the final version. Think of Copilot's suggestions as a helpful starting point. It's your job to take that first draft and polish it into something great.

Break long suggestions into smaller pieces

Sometimes Copilot gives you a big block of code all at once. While that might seem helpful, it can also be overwhelming or hard to debug later. Instead of accepting a whole multi-line function, try breaking it into steps.

Python prompt:

```
# Calculate average score and return grade
```

Copilot-generated code:

```
def evaluate_student(scores):
    total = sum(scores)
    avg = total / len(scores)
    if avg >= 90:
        return "A"
    elif avg >= 80:
        return "B"
    elif avg >= 70:
        return "C"
    elif avg >= 60:
        return "D"
    else:
        return "F"
```

This code works, but it could be clearer if it were broken down.

Refined Python prompt

```
#Take Copilot’s initial suggestion and refactor it into small
maintainability.
```

Copilot-generated code:

```
def calculate_average(scores):
    return sum(scores) / len(scores)
```

```
def get_grade(avg):
    if avg >= 90:
        return "A"
    elif avg >= 80:
        return "B"
    elif avg >= 70:
        return "C"
    elif avg >= 60:
        return "D"
    else:
        return "F"

def evaluate_student(scores):
    avg = calculate_average(scores)
    return get_grade(avg)
```

By refining the prompt to specify the need for helper functions, the developer guided Copilot to produce clearer and more modular code. The improvement comes from better prompting, not from manually rewriting the code.

> **Tip**
>
> If Copilot gives you a long code block, pause and consider whether you can split it into smaller, reusable functions.

Rename variables for clarity

Copilot-generated code often uses generic variable names like x, temp, and data. These variables might work in small snippets, but they make your code harder to read and maintain.

Prompt

```
//Calculate average from a list
```

Copilot-generated JavaScript code:

```
function calculate(t) {
    let s = 0;
    for (let i = 0; i < t.length; i++) {
        s += t[i];
    }
    return s / t.length;
}
```

Revised Prompt

```
//Create a function calculateAverage(scores) that returns the
```

Revised version of the Copilot-generated JavaScript code:

```
function calculateAverage(scores) {
    let total = 0;
    for (let i = 0; i < scores.length; i++) {
        total += scores[i];
    }
    return total / scores.length;
}
```

With the refined prompt, Copilot generated clearer variable names, making the function understandable without additional comments.

The following table shows examples of Copilot's likely initial suggestions and how you might refine them to improve clarity, maintainability, or performance.

Scenario	Copilot's initial suggestion	Developer's refined version

Scenario	Copilot's initial suggestion	Developer's refined version
Fetch data from API	Basic requests.get()	Add a timeout, error handling, and a docstring
Calculate average of scores	Single-line average	Add a check for an empty list
Check whether a user is an administrator	Simple role check	Use configuration-driven role validation
Generate a password	Random letters only	Add complexity, length check, and a secure method
Format numbers	No localization	Use locale or formatted string functions

Add comments and documentation

Copilot sometimes generates working code but skips documentation or explanations. Adding a short docstring or inline comment helps teammates (and future you) understand the why behind the logic.

Python Prompt

```
# Write a function fetch_user_data(api_url) that fetches data
```

Add a docstring and an inline comment to explain the code

Copilot-generated code with a docstring and a comment added:

```
def fetch_user_data(api_url):
    """
```

```
    Sends a GET request to the given API URL and returns the resp
    """
    response = requests.get(api_url)
    return response.json()
```

This function includes a docstring and an inline comment, guiding readers on its purpose and behavior.

```python
#Write the proper comments for the code below
def fetch_user_data(api_url):
    """
    Sends a GET request to the given API URL and returns the response data.

    """
    response = requests.get(api_url)
    # Check if the request was successful (status code 200)
    return response.json()

```

A Python function used to demonstrate the importance of adding proper comments and documentation

Tip

Don't wait until the end of the coding process to add comments. Add them as you refine Copilot-generated code, especially if Copilot didn't provide any.

Use linters and formatters

Even if Copilot gives you good logic, the style of its code might not match your team's standards. Linters (like ESLint or Flake8) and formatters (like Prettier or Black) can help you catch issues that Copilot might miss, such as extra spaces, inconsistent indentation, or unused variables.

Important

AI-generated code is not exempt from your team's coding guidelines. Always run it through the normal checks you would use with any other code.

Review suggestions line by line

Even if Copilot completes an entire block of code, take a moment to read each line. Does it make sense? Does it match your logic? Could it be written in a clearer way?

It's easy to assume that code is fine if it compiles or passes a test, but to ensure that the code is truly high quality, you need to take ownership of every line. Copilot gives you speed, and refinement gives you quality. By combining the two, you'll really start to shine as a developer. When you take the time to review, edit, and polish the code Copilot suggests, you'll build better habits and much better software.

See Also

To explore how to craft better prompts that guide Copilot toward more useful results, refer to Chapter 3: Writing effective prompts for AI-powered coding.

Refining AI-generated code

Copilot often gives you a draft, and it's your job to make it better. Use this list to clean up, clarify, and improve Copilot's suggestions. When refining Copilot's output, consider:

- Can I break this code into smaller, reusable functions?

- Are variable and function names meaningful and descriptive?
- Is this code documented with comments or docstrings?
- Have I removed unnecessary logic and cleaned up the formatting?
- Have I run a linter or formatter to ensure consistency?

Guide Copilot effectively

GitHub Copilot is powerful, but it's not psychic. If you want the best results, you have to guide it thoughtfully. That doesn't mean micromanaging every suggestion; it means learning how to communicate clearly with Copilot, as you would with a junior developer who's eager to help but still learning the ropes.

Copilot is as helpful as the guidance it receives. When you take a moment to write clear prompts, use meaningful names, and break down your logic, Copilot becomes far more accurate and useful. It's not about controlling every detail; it's about communicating clearly. The more effort you put into guiding Copilot, the less time you'll spend fixing what it misunderstood

This section walks you through practical ways to guide Copilot so that it more quickly produces better code and with fewer mistakes.

Start with clear, specific comments

The easiest way to steer Copilot in the right direction is by writing a short, focused comment that explains exactly what you want the code to do.

Vague Python prompt:

```
# process data
```

This is too vague. Copilot might generate something, but it probably won't be what you had in mind.

Improved Python prompt:

```
# Process list of student dictionaries and return names of th
```

Now Copilot knows your intent, your data structure, and your goal.

> **Tip**
>
> Think of comments as mini-prompts. The more detail you give, the better Copilot can predict what you want.

Use examples or constraints in comments

Copilot learns from patterns. If you give it a starting point, an example, or a constraint, it's more likely to return exactly what you need.

Vague Python prompt:

```
# Generate a list of even numbers from 1 to 50
```

Copilot-generated code:

```
even_numbers = [num for num in range(1, 51) if num % 2 == 0]
```

Because the comment lacks specific usage or output expectations, the code that Copilot suggests isn't as helpful as it could be.

```
# Generate a list of even numbers from 1 to 50
even_numbers = [num for num in range(1, 51) if num % 2 == 0]
```

A Python list comprehension that generates even numbers in response to a vague comment

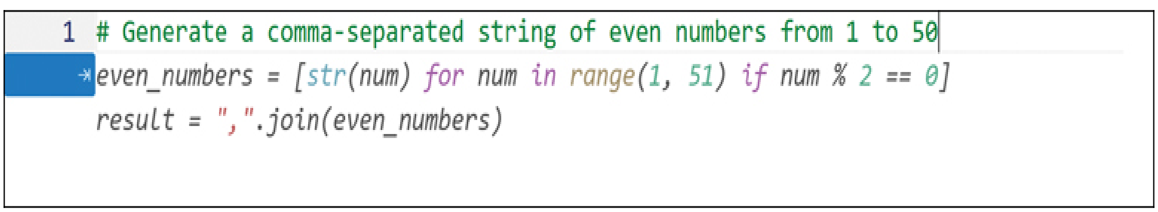
Improved Python prompt:

```
# Generate a comma-separated string of even numbers from 1 to
```

Copilot-generated code:

```
even_numbers = [str(num) for num in range(1, 51) if num % 2 == 0]
result = ",".join(even_numbers)
```

Because you requested both the input constraint and the expected output format, Copilot was able to suggest code that works better for your scenario. By adding that extra detail, you guided Copilot to deliver exactly what you had in mind. This is a good example of a context-rich comment.



A Python list comprehension that converts even numbers into a comma-separated string, based on a specific and helpful comment

Write descriptive function names and docstrings

Copilot gathers clues from your code structure. A well-named function gives Copilot context even before you write a prompt.

Python prompt:

```
def filter_active_users(users):
```

Copilot-generated Python code:

```
    return [user for user in users if user['status'] == 'active']
```

This code works, but you can add a docstring for an even better code.

Improved Copilot-generated Python code:

```
def filter_active_users(users):
    """
    Returns a list of users with 'active' status from the given u
list.
    """
```

By adding a clear docstring, Copilot generates a more self-documenting function. This not only improves readability for other developers but also ensures the function’s purpose is immediately understood without needing to read through the code line by line.

```python
def filter_active_users(users):
    """
    Returns a list of users with 'active' status from the given user list.
    """
    return [user for user in users if user.get('status') == 'active']
```

A Python function with a clear name and descriptive docstring

Break logic into steps

Break down your logic into steps before asking Copilot to write any code. This process helps you plan and gives Copilot a roadmap.

Python prompt:

```
# Step 1: Connect to the database
# Step 2: Fetch all records from the 'orders' table
# Step 3: Filter orders from the last 30 days
```

Copilot will likely follow this sequence and generate code accordingly.

Don’t settle; try again

If Copilot’s first suggestion isn’t right, don’t just delete it and move on. You can trigger alternative suggestions by using keyboard shortcuts or by rephrasing your comment. In Visual Studio Code, try:

- Alt+] to see the next suggestion
- Alt+[to see the previous suggestion
- Alt+/ to manually trigger a suggestion

Important

Copilot doesn’t always get its suggestions right on the first try. Use Copilot interactively: Adjust your prompt, cycle suggestions, or edit what it gives you.

See Also

To explore how Copilot can assist with real-world developer workflows, refer to Chapter 8: Using AI-powered development workflows in real-world scenarios.

Guiding Copilot effectively

Better prompts = better results. Use this list to improve how you communicate with Copilot so it helps you more rather than creating extra work for you.

Before you write a prompt or comment, consider:

- Is my comment clear about what I want the code to do?
- Have I included any needed constraints or data structures?
- Could a short example or step-by-step comment make this clearer?
- Am I naming functions in a way that hints at their purpose?
- Did I try cycling through alternate suggestions if the first one wasn't right?

Practice real-world examples

Now that you've learned how to recognize mistakes, understand their causes, refine Copilot's suggestions, and guide Copilot effectively, you're ready to put your knowledge into action.

This section gives you hands-on practice through real-world coding situations. First, you'll see an example that demonstrates how to apply the concepts from earlier sections. Then, you'll get a chance to try a similar task on your own. These scenarios are meant to help you build better habits and grow confident using Copilot in everyday development work.

You can try these examples in your preferred IDE, though we recommend Visual Studio Code for full Copilot integration.

Scenario 1: Clean up messy logic

Every developer eventually inherits code that's technically functional but difficult to read, maintain, or extend. This kind of legacy or rushed logic can slow down development and introduce hidden bugs. While Copilot isn't a replacement for human judgment, it can be a useful partner when refactoring old or unclear code.

Let's look at an example of code that can be cleaned up or refactored with the help of GitHub Copilot.

Original code:

```
def c(d):
    if d != None:
        for i in range(len(d)):
            if d[i] > 0:
                print(d[i])
```

This Python function contains a docstring and a list comprehension that filters users with the status active. The logic is functional but could be simplified.

```
def c(d):
    if d != None:
        for i in range(len(d)):
            if d[i] > 0:
                print(d[i])
```

A Python function with a working but verbose implementation for filtering active users

Python prompt:

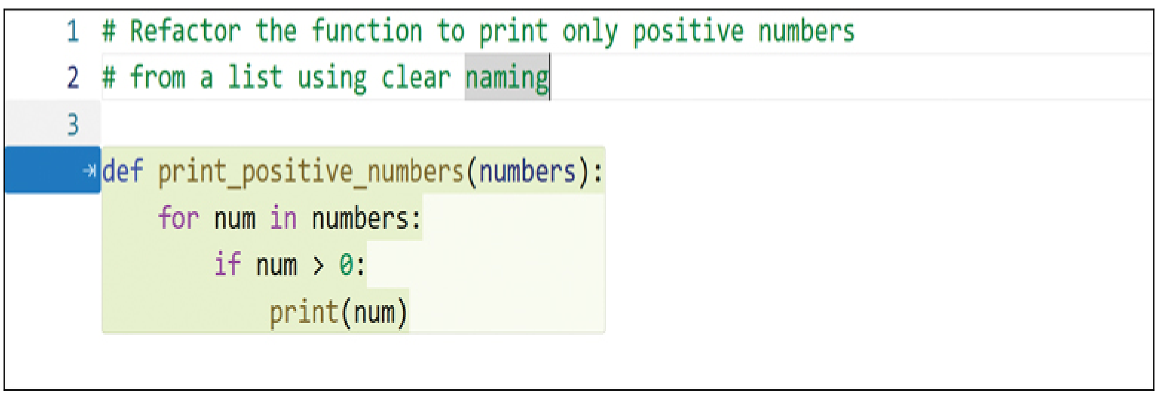
```
def c(d):
  if d != None:
     for i in range(len(d)):
       if d[i] > 0:



            print(d[i])
```

```
        print(a[i])
# Refactor the function to print only positive numbers from a
```

Copilot-suggested code:

```
def print_positive_numbers(numbers):
    for num in numbers:
        if num > 0:
            print(num)
```

The function now prints only the positive values from the list. GitHub Copilot cleaned up the verbose logic from the previous version by using clearer conditions and better structure, improving both readability and maintainability.



A refactored Python function, where GitHub Copilot has cleaned up messy logic

Now that you've seen an example, try it for yourself. In this scenario, say that.you have a function with verbose logic that prints positive numbers from a list.

Your tasks:

1. Ask Copilot to suggest a cleaner version of your code.
2. Rename the variables to be more descriptive.
3. Add a docstring.

Scenario 2: Write helper functions from prompts

In this scenario, say that you're building a dashboard and need utility functions that can format data.

```
#
```

Your tasks:

1. Give Copilot this Python prompt:

```
Create a function to format a number with commas (e.g., 1000
```

2. Accept or refine Copilot's response.
3. Add input validation for non-integer values.
4. Write one unit test using Copilot.

> **Tip**
>
> Try changing your prompt slightly to see how the output changes. For example: "format a number for display in currency."

Scenario 3: Add security and error handling

Copilot often delivers quick, functional code, but it isn't always secure or production ready. Many of Copilot's suggestions assume ideal conditions: The network connection always works, inputs are always valid, and no exceptions are ever raised. In real-world development, however, things go wrong all the time.

If you're using Copilot to scaffold code that interacts with APIs, user inputs, databases, or external services, it's your responsibility to add the layers of security and error handling that AI might overlook. This includes validating inputs, catching exceptions, setting timeouts, and avoiding risky patterns like hardcoded values or unchecked access.

For this scenario, assume that Copilot gives you a working function that lacks security or robustness.

Your tasks:

1. Give Copilot this Python prompt:

```
# Connect to an API and return JSON data
```

2. Examine the function that Copilot generates.
3. Add error handling (e.g., try-except, timeout).
4. Check for unexpected responses or missing fields.

> **Important**
>
> Copilot-generated code may not include error handling by default. You're responsible for making sure it's safe and reliable, especially for production code.

Scenario 4: Use Copilot for SQL, YAML, and HTML

Copilot isn't just for traditional programming languages. It also supports markup, configuration, and query languages like SQL, YAML, and HTML. This makes it especially helpful for full-stack developers, DevOps professionals, and data analysts who frequently switch between file types.

In this scenario, you'll create files for each of these formats and test how Copilot responds to natural language prompts. To start, open your preferred GitHub Copilot–enabled IDE (such as Visual Studio Code) and follow these steps:

1. Create a folder named copilot_multifile_practice.
2. Inside this folder, create the following three files:
   - top_customers.sql
   - ci_workflow.yml
   - navbar.html

SQL example:

Your goal is to get the top 10 customers by order value.

Your tasks:

1. Open top_customers.sql in Visual Studio Code.
2. Type the following comment on the first line:

```sql
-- Get top 10 customers by total order value
```

3. Let Copilot suggest the output.

Copilot-generated code:

```sql
SELECT customer_id, SUM(order_value) as total_order_value
FROM orders
GROUP BY customer_id
ORDER BY total_order_value DESC
LIMIT 10;
```

```sql
--SQL:
-- Get top 10 customers by total order value
SELECT customer_id, SUM(order_value) as total_order_value
FROM orders
GROUP BY customer_id
ORDER BY total_order_value DESC
LIMIT 10;
```

An SQL query generated with the help of GitHub Copilot, based on a natural language comment to retrieve the top 10 customers by total order value

YAML example

Your goal is to use GitHub actions to create a CI workflow.

Your tasks:

1. In Visual Studio Code, create and open a new file named ci_workflow.yml (for example, inside a .github/workflows/ directory).
2. Type the following comment:

```
# GitHub Actions workflow to run tests on push to main br
```

3. Let Copilot generate a YAML block.

Copilot-generated code:

```yaml
name: Run Tests
on:
                push:
          branches:
               - main
```

```
jobs:
               test:
                runs-on: ubuntu-latest
                steps:
                - uses: actions/checkout@v2
                - name: Set up Python
                uses: actions/setup-python@v2
                with:
                      python-version: '3.x'
                - name: Install dependencies
                run: pip install -r requirements.txt
               - name: Run tests
               run: pytest
```

What to do next:

- Adjust runtime settings (e.g., node vs. Python, if needed).
- Try modifying your prompt to specify the language or test tool, like this:

```
# GitHub Actions workflow to build and test Node.js app
```

HTML example

Your goal is to create a responsive navigation bar.

Your tasks:

4. Open navbar.html in Visual Studio Code or your preferred IDE (or another IDE with GitHub Copilot enabled). Type the following comment:

```
<!-- Responsive navbar using Bootstrap -->
```

5. Let Copilot generate an HTML snippet.

Copilot-generated code:

```html
<!-- Responsive navbar using Bootstrap -->
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Brand</a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-tar
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarNav">
    <ul class="navbar-nav">
      <li class="nav-item active">
        <a class="nav-link" href="#">Home <span class="sr-only">(current)</spa
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Features</a>
      </li>
      <li class="nav-item">
```

HTML code for a responsive Bootstrap navbar, auto-generated using GitHub Copilot based on a descriptive HTML comment

The structure of this code demonstrates GitHub Copilot's ability to generate production-ready frontend layout code from a single descriptive comment.

What to do next:

- Add your own links or branding text.
- Refine your prompt for styling, color themes, or frameworks. For example, you might enter this prompt:

```html
<!-- Create a dark-themed responsive navbar using Bootstrap 5
```

Scenario 5: Combine multiple steps

Now you can apply everything you've learned in this chapter, from clear prompting to refinement, validation, and testing.

Your tasks:

1. Use Copilot to generate a multi-step function by giving it this Python prompt:

```
# Load data from CSV, filter rows where 'score' > 75, and ret
```

2. Split the code into logical subfunctions.
3. Add error checking for missing or malformed data.

Taking your knowledge into the real world

Practice is the key to mastering Copilot. The more you experiment, the better you'll get at writing prompts, reviewing suggestions, and creating code that's clean, correct, and context aware. You can apply the same thinking you've used here to real tasks from your own projects.

The goal isn't to rely on Copilot; it's to collaborate with it. Think of Copilot as your silent coding partner, ready to support you at every step as you lead the way.

See Also

To learn how to use Copilot for debugging and troubleshooting in real-world scenarios, go to Chapter 5: Debugging and troubleshooting code with Copilot.

Error-prone Copilot scenarios and prevention tips

While GitHub Copilot can accelerate your workflow and reduce repetitive coding, it can also quietly introduce mistakes, especially in common tasks where developers tend to move quickly or rely too heavily on default suggestions. In many cases, Copilot produces code that runs but doesn't reflect best practices or safety checks.

These small errors often go unnoticed until they cause problems in testing, production, or code reviews. This section highlights typical use cases where Copilot might give you working but suboptimal or risky code. You'll also see simple prevention tips that help you build better habits and avoid costly mistakes.

The following table provides a quick reference for common scenarios where Copilot-generated code can introduce subtle errors and outlines typical mistakes developers make in these cases as well as practical steps to prevent them.

Scenario	Common mistake	How to prevent it
Writing SQL queries	Missing WHERE clause	Always specify filter criteria
HTML generation	Unresponsive layouts	Prompt with "responsive Bootstrap" keywords
Fetching data	No retry mechanism	Use libraries with built-in retries
String formatting	Using + instead of f-strings	Use f-strings or .format() for clarity
Reading files	Failing to close a file properly	Use the open() context manager

Practicing smarter with Copilot

Practice isn't just repetition; it's intentional improvement. Use this list as you work through real-world prompts and challenges.

During or after using Copilot, consider:

- Did I break the task into smaller logical steps before prompting?
- Did I explore how slight prompt changes affect the suggestions?
- Did I edit or refactor what Copilot gave me to improve clarity?
- Have I included error handling and validation where needed?
- Did I compare my version to Copilot's and reflect on the differences?

Quick reference: Comment prompts that guide Copilot

The following table provides sample prompts to help you clearly communicate your coding goals to Copilot. Each comment is designed to produce accurate and context-aware code suggestions.

Goal	Effective comment prompt
Write a function to check even numbers	Create a function to check if a number is even
Loop through a list and print values	Loop through a list of names and print each name
Sort a list of dictionaries by a key	Sort products by price in descending order
Read a CSV file and return rows	Read data from CSV and return list of dictionaries
Format currency output	Format a number as USD currency with two decimal places

Goal	Effective comment prompt
Validate an email address	Validate email address using regex
Fetch data from an external API	Fetch JSON data from OpenWeatherMap API using requests
Generate a password	Generate a secure random password with letters, numbers, and symbols
Write unit tests for a function	Write unit tests for calculate_total() function using unittest
Create a responsive navbar in HTML	<!-- Create a responsive navbar using Bootstrap 5 -->
Connect to a SQLite database	Connect to a SQLite database and return cursor
Get top N records in SQL	Select top 5 customers by total order value
Convert list of strings to lowercase	Convert list of strings to lowercase using list comprehension
Create a GitHub Actions workflow	GitHub Actions: Run tests on push to main branch
Remove duplicates from a list	Remove duplicates from list while preserving order

Skills review

In this chapter, you learned how to:

- Identify common Copilot mistakes, such as accepting suggestions blindly, skipping validation, and letting AI write critical logic unchecked.
- Understand why these mistakes happen, including overtrusting suggestions, using unclear prompts, and believing that Copilot is a mind reader when it's really a pattern predictor.
- Refine AI-generated code by breaking it into smaller parts, renaming variables, and reviewing every suggestion for accuracy and style.
- Practice ways to guide Copilot more effectively using clear comments, function names, and structured thinking to shape better suggestions.
- Apply what you have learned in real-world scenarios, including using prompts across multiple languages, fixing errors, and experimenting with alternate phrasing.
- Develop stronger habits and apply Copilot responsibly in your daily workflow.

Practice tasks

No practice files are necessary to complete the practice tasks in this chapter.

Before you complete the following tasks, follow these steps:

1. Launch Visual Studio Code (or your preferred IDE) with GitHub Copilot enabled.
2. Create a new folder on your desktop called Copilot_Mistakes_Practice.
3. Inside that folder, create a new Python file named copilot_mistakes.py.
4. Use this file to work through the practice tasks in the following sections.

Spot and fix common Copilot mistakes

Now it's your turn to practice. In this task, you'll re-create the steps shown in the example and then improve on Copilot's output yourself. Follow these setup steps first:

1. Inside the Copilot_Mistakes_Practice folder, create a new file and name it refine_code.py.
2. Enter this comment on line 1:

```
# Create a function to calculate average from a list of r
```

3. Accept Copilot's suggestion.
4. Carefully review the code.
5. Modify the code to:

   a. Handle empty lists.

   b. Include a docstring and use meaningful variable names.

   c. Raise an error if the input is not a list of numbers.

6. Save the file.

Understand why Copilot mistakes happen

7. In this task, you'll experiment with vague and refined prompts to see firsthand how Copilot interprets your instructions. This exercise will help you understand why Copilot sometimes produces incorrect or incomplete code, and how better prompting improves the results. Inside the Copilot_Mistakes_Practice folder, create a new file and name it copilot_misunderstanding.py.
8. Enter this vague comment:

```
# sort data
```

9. Accept Copilot's suggestion.
10. On a new line, add a more detailed prompt:

```
#Sort a list of student records by 'score' in descending
```

11. Accept Copilot's new suggestion and compare the output from the two suggestions.
12. Reflect: What changed? What stayed the same?
13. Add a summary at the bottom of the file, describing what Copilot misunderstood and why.
14. Save the file.

Fix and refine AI-generated code

15. In this task, you'll see how Copilot's first draft of code can be improved through refinement. Copilot may generate a working function, but you'll practice breaking it into smaller parts, improving naming, and adding documentation to make it more readable and maintainable. Inside the Copilot_Mistakes_Practice folder, create a new file and name it refined_code.py.
16. Enter this prompt:

```
# Calculate average score and return grade
```

17. Accept Copilot's suggestion (which should be a full function).
18. Refactor the code to:

a. Use helper functions (e.g., calculate_average(), get_grade()).

b. Rename any unclear variables.

c. Add a docstring and inline comments.

19. Run a linter (like Flake8 or pylint) and fix any style issues.
20. Save the updated version of the file.

Guide Copilot effectively

This task helps you practice how phrasing your prompt differently can change Copilot's output. You'll start with a simple prompt, then refine it step by step to see how Copilot improves its suggestions with more context and detail.

1. Inside the Copilot_Mistakes_Practice folder, create a new file and name it copilot_prompting.py.
2. Enter this prompt:

```
# Check if a number is prime
```

3. Accept Copilot's output.
4. Now rephrase the prompt:

```
# Write a function that checks if a number is prime and r
```

5. Accept the new Copilot output.
6. Compare the functions and document the differences between them.
7. Add a third prompt:

```
# Include input validation and handle edge cases like neg
```

8. Accept and refine the result.
9. Save the file.

Practice real-world examples

This final task brings together everything you've learned in this chapter. You'll practice using Copilot to handle real-world coding scenarios across different problem types. For each example, try the prompt, review Copilot's suggestion, and then refine it as needed.

1. Inside the Copilot_Mistakes_Practice folder, create a new file and name it real_world_tasks.py.
2. Enter the following prompt:

```
# Load CSV file, filter rows where score > 80, and returr
```

3. Accept and review Copilot's suggestion.
4. Add improvements or fixes.
5. Add a one-line comment describing what you changed and why.
6. Save the file.
7. Repeat steps 2–6 for each of the following prompts:

```
# Generate a comma-separated string of even numbers from
# Fetch weather data from external API with error handlir
# Write a test case for the calculate_total function usir
```

10

Exploring the future of AI in software development

In this chapter

- Explore upcoming AI trends in coding
- Understand the evolution of GitHub Copilot
- Discover new developer roles in the AI era

Practice files

There are no practice files for this chapter.

Software development is undergoing one of its most profound transformations to date, powered by artificial intelligence. What was once a manual, syntax-heavy task is becoming a creative collaboration between human intent and machine intelligence. Developers no longer work alone. They now co-create with AI systems that can generate, test, optimize, and even infer solutions from patterns in code.

AI assistants like GitHub Copilot are already helping developers write cleaner code, automate repetitive tasks, and explore unfamiliar APIs. But this is just the beginning. The next wave of AI tools will enable voice-first programming, self-healing codebases, and autonomous code agents that build and deploy entire features based on natural language goals.

This changing landscape demands new skills, workflows, and mindsets. Developers must become fluent in prompt design, ethical AI usage, and system-level thinking. They must understand not only how AI generates code but how to supervise, guide, and refine its output responsibly.

This chapter explores AI-driven trends in coding, how GitHub Copilot is evolving, and how the developer's role will be redefined in an AI-assisted future.

Explore upcoming AI trends in coding

AI is no longer a distant idea in software development. It's the copilot, co-author, and, soon, the co-architect of the code we write. From automating routine tasks to enabling fully voice-guided development, AI is creating new opportunities and redefining what it means to "write code." This section explores emerging trends that are poised to reshape the developer experience in profound ways.

Understand the shift to natural language programming

One of the most exciting shifts is the evolution from code-first to intent-first development. With tools like GitHub Copilot Chat and OpenAI Codex, developers can now describe their desired functionality in plain English, and the AI generates the underlying code.

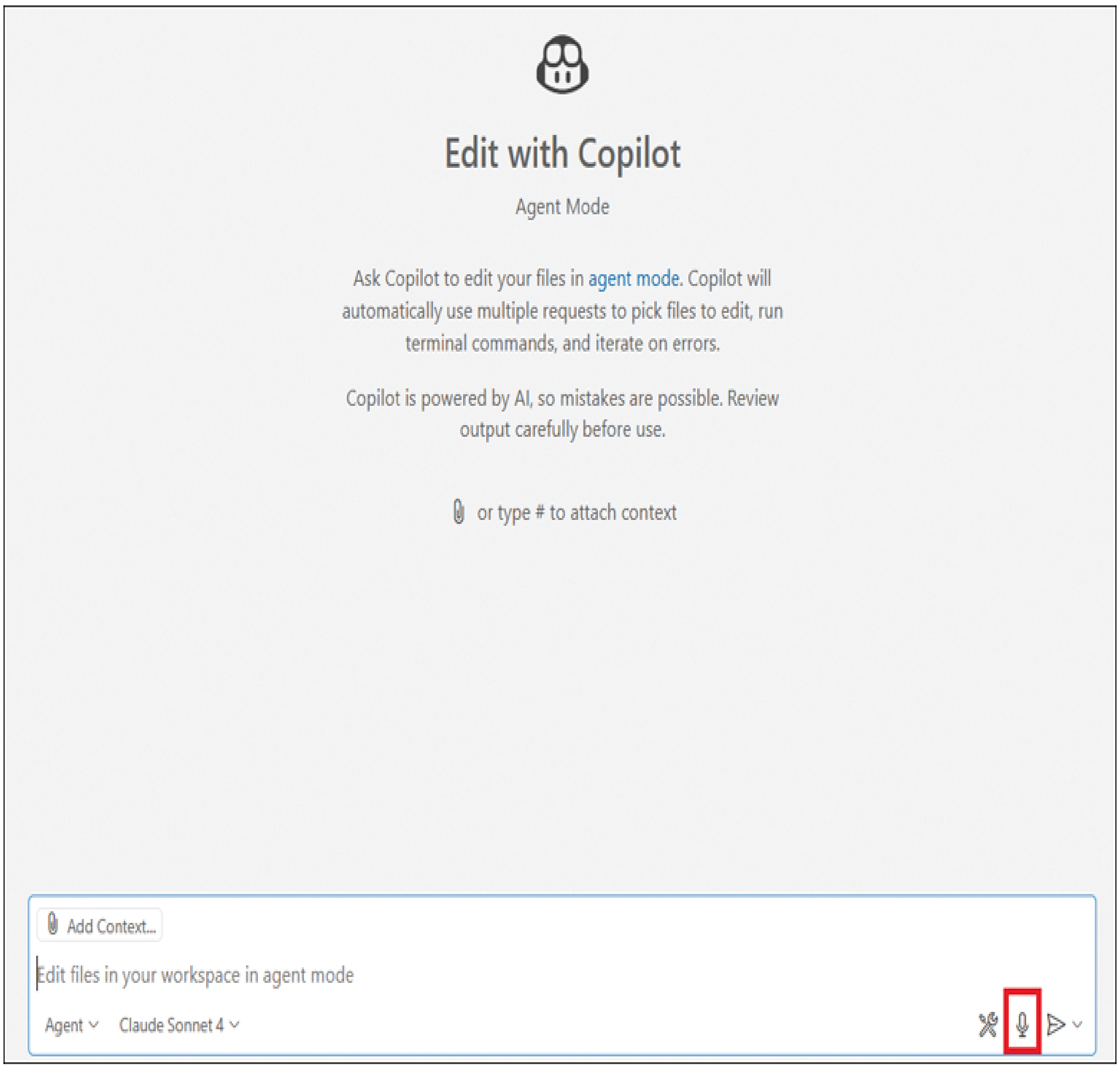
This shift transforms the developer's role from coder to orchestrator. Instead of manually writing logic from scratch, developers now describe what they want, review generated solutions, and focus on high-level architecture.

See Also

To learn how to craft effective prompts, see Chapter 3: Writing effective prompts for AI-powered coding.

Embrace voice-first coding environments

Imagine coding without touching a keyboard. With AI-powered speech-to-code systems, developers can now dictate code, ask for refactoring, and debug by simply speaking. This opens doors for hands-free programming, accessibility for developers with disabilities, and even coding during brainstorming walks.



The GitHub Copilot Agent Mode UI, highlighting voice command input (microphone icon) and context-aware editing

Emerging tools combine GitHub Copilot with speech recognition systems like Whisper and in-browser assistants. These environments are still in early development, but they represent a leap toward multimodal development experiences.

Important

Voice-first coding is promising but still limited by current speech recognition accuracy and IDE integration. Always verify generated code manually.

AI-powered coding trends vs. traditional coding approaches

The way developers interact with code is changing rapidly. Traditional coding approaches required a deep understanding of syntax, language semantics, and structure, often involving repetitive manual effort. In contrast, AI-powered development introduces a more intuitive, intent-driven workflow, where tools like GitHub Copilot assist with code generation, explanation, and optimization. The following table compares the key differences between traditional and AI-assisted coding practices to highlight how software development is evolving.

Aspect	**Traditional coding**	**AI-powered coding (Copilot)**
Input method	Manual typing	Prompts, voice, context
Code generation	Written line-by-line	Suggested in full blocks

Aspect	Traditional coding	AI-powered coding (Copilot)
Learning curve	Steep (especially with new languages)	Flattened with in-context help
Code reuse	Manual refactoring	Suggested based on known patterns
Collaboration	Pair programming with humans	Pair programming with AI
Time to prototype	Days or hours	Minutes or seconds

Use AI as a code architect, not just a code generator

AI models are evolving from being reactive assistants to being proactive agents. Upcoming AI agents can:

- Interpret project documentation.
- Plan a feature's implementation.
- Break a project into tasks.
- Generate test cases and even deploy code.

These agents don't just respond; they initiate workflows based on goals and parameters you set. While still experimental, such agents hint at a future where developers act more like supervisors than line-by-line authors.

Tip

Follow AI open-source agent platforms like AutoGPT and Devin for sneak peeks into autonomous development.

See Also

To see how Copilot is currently helping with test generation, visit Chapter 6: Writing and automating tests with GitHub Copilot.

Explore immersive and spatial coding environments

As augmented reality (AR) and spatial computing gain traction, the concept of holographic pair programmers is moving from science fiction into the realm of practical innovation. Imagine coding with an AI assistant that is not on your screen but beside you, projected as a holographic agent that is fully aware of your coding context and workspace.

In this environment, you would wear AR glasses (such as Apple Vision Pro, HoloLens, or Meta Quest Pro) and see your code in 3D space. Instead of switching windows or dragging panels, you would be able to:

- Pin code snippets midair so you can reference them as you work.
- Swipe to accept or reject AI suggestions.
- Point to a function and ask “What’s wrong here?” to trigger debugging advice.
- Visualize call stacks or code graphs spatially.

This is pair programming reimagined, where your partner in the pair is a hyper-intelligent AI that knows your codebase, your style, and your team's architectural patterns.

Although still emerging, holographic AI assistants represent a compelling new interface paradigm where Copilot becomes not just a code generator but a visually present co-developer, reacting to your gaze, gestures, and voice in real time.

Prepare for cross-modal development

As AI models gain the ability to process not just text but images, audio, and even video, cross-modal development is becoming viable. Examples include:

- Uploading a screenshot and asking AI to generate frontend code.
- Providing a voice description and asking AI to generate a UI mockup.
- Recording a user flow and asking AI to generate automated test scripts.

These trends will redefine UX design, QA, and even DevOps, transforming them into AI-mediated workflows rather than purely manual efforts.

> **Important**
>
> Cross-modal models like GPT-4o are already in preview. Expect rapid advancement in this space over the next 12–18 months.

The rise of real-time code optimization

AI is not just for generation. It's also becoming the gatekeeper for performance. Soon, AI tools will analyze running applications in real time, detecting inefficiencies and suggesting improvements. Some experimental systems can:

- Recommend better data structures.
- Detect slow-performing loops.
- Automatically rewrite SQL queries for speed.

This trend will shift performance tuning from “after deployment” to continuous optimization, with AI providing feedback loops directly into the IDE.

See Also

To learn how GitHub Copilot currently assists with debugging and refactoring, see Chapter 5: Debugging and troubleshooting code with Copilot.

What developers need to do now

To stay ahead of these trends, developers should:

- Practice engineering prompts: Get better at describing code requirements in plain English.
- Follow AI open-source projects: Stay updated with communities that are building Copilot plugins, agent workflows, and prompt libraries.
- Invest in ethical AI awareness: Understand copyright, licensing, and security risks in AI-generated code.

Tip

Treat AI like a junior teammate. It’s helpful, fast, and creative but still needs review and supervision.

Important

You are always accountable for the code you ship, even if it was created with the help of AI. Never skip testing and review.

Understand the evolution of GitHub Copilot

GitHub Copilot began as a revolutionary AI-powered autocomplete tool. Today, it is evolving into a full-fledged coding partner, offering not just code suggestions but explanations, documentation, testing, and even natural language chat. Its journey reflects the broader evolution of developer tooling, from passive text editors to intelligent assistants that learn, adapt, and collaborate.

Understanding Copilot's evolution helps developers appreciate its growing potential and responsibly integrate it into modern development workflows. In this section, we'll explore how GitHub Copilot has transformed over time, from basic inline suggestions to contextual chat, proactive workflows, and organization-wide adoption.

From autocomplete to AI pair programming

When GitHub Copilot was first introduced in 2021, it resembled a powerful autocomplete engine. Powered by OpenAI Codex, it could suggest code as you typed, completing functions and loops and even generating documentation from comments.

Early Copilot could handle in-line code completion based on current file context and also comment-to-code transformation (e.g., // sort list). It also provided support for multiple languages, including Python, JavaScript, and TypeScript. While its suggestions were impressive, early versions lacked understanding of project context, external documentation, or multi-file awareness.

> **Tip**
>
> Copilot works best when comments are specific. Use action verbs and expected output in your prompts to guide its suggestions.

Copilot Chat: Context-aware conversations

The introduction of Copilot Chat brought a game-changing enhancement to the Copilot experience. Developers could now engage in natural language conversations with Copilot, asking questions like:

- "What does this function do?"
- "Can you write a test case for this class?"
- "Why is this SQL query slow?"
- "Can you provide project-wide context, not just for the current file?"
- "Can you explain this code and help me debug it?"
- "Can you refactor this to async or perform other task-oriented changes?"

> **Important**
>
> Copilot Chat relies on active GitHub authentication and higher-tier subscriptions. Be sure you're on the Pro or Pro+ plan to access advanced features.

> **See Also**
>
> For help setting up Copilot and selecting the right plan, revisit Chapter 2: Setting up GitHub Copilot for your development workflow.

The evolution of GitHub Copilot capabilities

As GitHub Copilot has matured, it has evolved from being a simple code completion engine into a full-featured AI assistant capable of engaging in contextual conversations, understanding entire projects, and integrating across multiple IDEs. Each iteration has expanded its awareness, utility, and value in professional environments. The following table outlines the key stages in Copilot's evolution and how its capabilities have expanded over time:

Phase	Key features	Context awareness	IDE support
Initial Release (2021)	Inline code suggestions	Limited to current file	VS Code only
Copilot Chat (2023)	Explanations, debugging, prompting	File and project	VS Code, JetBrains
Copilot for Business	Team policies, telemetry	Project and team settings	Multi-IDE
Copilot Enterprise (Preview)	Use of internal repos as context	Organization-level and custom codebases	GitHub Codespaces

The rise of AI documentation and learning assistants

GitHub Copilot isn't just your coding partner; it's your personal documentation assistant and real-time tutor. As AI becomes more embedded in development environments, developers are discovering its potential to generate helpful comments, explain complex logic, and even teach new technologies in context.

Generate inline documentation

Instead of writing docstrings or comments manually, developers can now:

- Add a comment header like # explain this function, and Copilot generates a Python-style docstring.

- Use prompts like // Add JSDoc for this function in JavaScript to create structured annotations.
- Write summaries for classes, APIs, and modules without needing to repeat boilerplate patterns.

Inline documentation enhances code readability and accelerates onboarding for new team members.

> **Tip**
>
> Use descriptive function names and inline comments to prompt better documentation suggestions from Copilot.

Learn by doing with AI as your mentor

New developers often struggle to understand existing code. Copilot Chat bridges the gap, allowing users to ask questions like these:

```
"What does this loop do?"
"Why is map() used here instead of for?"
"Convert this SQL query to Python using pandas"
```

It's like having a senior engineer on demand, willing to explain anything, anytime in simple language.

> **Important**
>
> Copilot explanations are probabilistic and may simplify technical detail. Always verify suggestions before relying on them for critical tasks.

Integration across the toolchain

GitHub Copilot was originally limited to Visual Studio Code but now integrates with multiple development environments, including these:

- Visual Studio 2022+
- JetBrains IDEs (IntelliJ IDEA, PyCharm, and WebStorm)
- Neovim
- Azure Data Studio
- GitHub Codespaces

In addition, Copilot is being embedded into the broader GitHub ecosystem, supporting pull request comments, code reviews, and even issue triage with smart suggestions.

> **Tip**
>
> Try using Copilot in JetBrains IDEs for full-stack projects. The language model adapts well to backend frameworks like Spring or Django.

Copilot for teams and enterprises

GitHub Copilot is no longer just for individuals. It now supports team-wide configurations and enterprise-grade governance.

New features include:

- Copilot for Business: Offers centralized billing, telemetry controls, and policy settings
- GitHub Copilot Enterprise: Uses your internal codebase as training context (with access controls)
- Compliance mode: Helps prevent suggestions based on public code with ambiguous licenses

> **Important**
>
> Organizations should enforce policies to ensure responsible AI usage, particularly in regulated industries and when using proprietary data.

AI and team collaboration: Beyond the individual developer

While GitHub Copilot is often seen as a personal tool, its true power emerges at the team level. AI-assisted development enhances team workflows, aligns coding standards, and improves communication without sacrificing individual creativity.

Teams that embrace Copilot benefit in a number of ways:

- Shared code intelligence: Everyone gets access to the same AI-powered knowledge base.
- Consistent naming and structure: Copilot learns team patterns and reinforces them.
- Faster reviews and iterations: AI can suggest improvements even before human reviewers see the code.

Copilot use cases by team role

GitHub Copilot serves a variety of roles across development teams. From backend development to documentation, testing, and infrastructure, Copilot offers tailored benefits depending on how it's applied. The following table summarizes common use cases by team role, along with the key benefits each type of user can expect.

Role	Copilot use case	Benefit

Role	Copilot use case	Benefit
Backend developer	Generate API endpoints	Accelerated scaffolding
Frontend developer	Suggest UI logic and component props	Faster iteration
DevOps engineer	Write YAML/Terraform/Docker	Reduce config errors
Junior developer	Learn by asking Copilot Chat	Improve self-sufficiency
Tech lead	Use Copilot for code reviews	More efficient feedback

Accelerate collaboration across disciplines

AI bridges the gap between technical and non-technical roles by:

- Translating requirements into starter code or configs.
- Generating readable summaries for non-developers.
- Assisting in collaborative prototyping sessions (e.g., pair coding with a product owner).

For instance, a data analyst could describe an SQL task in plain English, and a developer could then prompt Copilot to generate the initial logic, reducing back-and-forth clarification.

Tip

Encourage cross-functional teammates (such as QA folks, designers, and analysts) to participate in prompt crafting during working sessions. Copilot can help fill technical gaps.

Enhancing code reviews and pull requests

Copilot's integration with GitHub pull request (PR) workflows is growing. Developers can:

- Use Copilot to generate summaries of PRs or commits.
- Get refactoring suggestions during code review.
- Speed up reviews by having Copilot explain unfamiliar patterns or identify anti-patterns.

> **Important**
>
> Copilot can assist in reviews but should not replace human accountability. Always validate logic, especially in sensitive or critical components.

Strengthening team culture and onboarding

New team members can ramp up faster with:

- Auto-generated docstrings for unfamiliar code.
- In-context explanations of business logic.
- Team-custom Copilot suggestions through GitHub Copilot Enterprise.

By strengthening team culture and onboarding, Copilot reduces mentoring load and promotes knowledge sharing without bottlenecks.

> **See Also**
>
> To explore how Copilot can assist in real-world project workflows, refer to Chapter 8: Using AI-powered development workflows in real-world scenarios

Integration of Copilot into CI/CD and DevOps pipelines

Modern development doesn't end with writing application code. It also involves building pipelines, automating deployments, defining environments, and ensuring system reliability. GitHub Copilot can assist DevOps engineers and full-stack developers by generating infrastructure as code scripts, CI/CD configurations, and deployment templates.

Write infrastructure and configuration code

Copilot can help with the following infrastructure and configuration code:

- Dockerfiles (e.g., # Dockerfile for a Flask app)
- Kubernetes YAML (e.g., # Define a Kubernetes deployment)
- Terraform modules (e.g., # Terraform configuration for an S3 bucket)
- GitHub Actions (e.g., # Workflow to deploy app on push)

> **Tip**
>
> Use descriptive comments to generate repeatable, version-controlled configuration files. Add validation after code generation by using linters like yamllint, tflint, or actionlint.

Automate the build, test, and deploy steps

In GitHub Actions and other CI/CD platforms, you can use Copilot to:

- Scaffold workflows for different languages and stacks.
- Add conditional logic for test stages (e.g., run integration tests on main branch only).
- Insert environment variables, secrets, and rollback steps.

For example, you could use a prompt like this with GitHub Actions:

```
# GitHub Actions workflow to build, test, and deploy a Node.j
```

Copilot will generate a ready-to-use multi-step CI pipeline that you can refine.

> **Important**
>
> AI-generated config files may include default or outdated practices. Always align AI-generated code with current platform recommendations.

Supporting end-to-end DevOps workflows

Developers can also use Copilot to support DevOps workflows. For example, they can:

- Create .env files, .gitignore, and .dockerignore files.
- Automate monitoring scripts by using Bash or PowerShell.
- Generate Helm charts or Ansible playbooks for infrastructure orchestration.

Copilot allows teams to standardize infrastructure with less manual setup while maintaining version control and traceability.

Continuous improvement and the future roadmap

GitHub Copilot is constantly evolving. GitHub and OpenAI regularly release updates to:

- Improve suggestion accuracy.

- Support more frameworks and languages.
- Reduce hallucinations and bias.
- Improve performance for large projects.

A number of experimental features are currently in development, including:

- Natural language–driven code navigation.
- Automated test generation based on commit messages.
- AI-assisted project bootstrapping and planning.

> **See Also**
>
> To explore real-world use cases powered by GitHub Copilot, refer to Chapter 8: Using AI-powered development workflows in real-world scenarios.

Discover new developer roles in the AI era

As AI becomes more integrated into the software development lifecycle, the role of developers is evolving. Developers are no longer just builders of systems; they are curators of AI prompts, validators of machine-generated output, and ethical stewards of automation. Rather than replacing jobs, AI is reshaping responsibilities, requiring developers to adapt and elevate their skills in new directions.

A common misconception is that AI will replace developers. In reality, AI is becoming one of the most powerful creative partners in a developer's toolkit. AI doesn't take away the role of developers. Rather, it amplifies their creativity, accelerates their problem solving, and extends their reach across unfamiliar domains.

Think of GitHub Copilot not as a silent tool but as a thought collaborator. When you're blocked on logic, you can use it to get alternatives. When you're writing tests, you can ask it to scaffold boilerplate code instantly.

When you’re cleaning up legacy code, you can have it refactor syntax while preserving intent.

AI is not doing the work for you; it’s doing the work with you. Much like a musical partner in a jam session, it listens, responds, improvises, and adjusts to your style. This dynamic encourages a more fluid, exploratory, and joyful development process. Here are some examples of AI-augmented creativity:

- Generating multiple approaches to solve a single problem (e.g., iterative, recursive, or functional styles)
- Refactoring complex code into more readable segments
- Suggesting improvements in naming, structure, or performance
- Translating functions across languages or paradigms (e.g., imperative to functional)

AI-augmented developer roles

The rise of AI in software development is transforming the role of the developer. Whereas developers traditionally focused primarily on writing and debugging code, today developers who are equipped with AI are guiding intelligent systems, curating prompt-based workflows, and supervising machine-generated output. The following table contrasts traditional development responsibilities with development responsibilities in AI-powered environments.

Role area	Traditional development	AI-augmented development
Focus	Manual code writing	AI output supervision and guidance
Primary skills	Syntax and tooling	Prompt engineering and validation

Role area	Traditional development	AI-augmented development
Testing	Manually scripted tests	AI-generated tests
Documentation	Manual documentation	AI-assisted generation and explanation
Learning	Forums, trial-and-error	In-IDE AI chat, real-time examples

From coder to AI supervisor

In this section, we'll look at the emerging roles that developers are stepping into in the age of AI-assisted coding, as well as the competencies and mindsets that will define future-ready professionals.

Developers traditionally focused on writing, testing, and deploying code. In the AI era, developers act more like AI supervisors, guiding AI-generated code, validating logic, and ensuring that AI suggestions align with project goals.

Key responsibilities of developers who are working with Copilot include:

- Reviewing Copilot suggestions for accuracy, security, and maintainability.
- Writing better prompts to generate usable output.
- Refining or rewriting Copilot-generated code for specific business logic.

Tip

Think of GitHub Copilot as a talented junior developer, who is fast but needs careful guidance and quality checks.

See Also

To learn about common issues in AI-assisted development, refer to Chapter 9: Avoiding common pitfalls with GitHub Copilot.

Becoming prompt engineers

Prompt engineering is now a core competency. Developers must learn how to clearly and efficiently communicate intent to AI systems.

Effective prompt engineers:

- Use structured language to convey goals.
- Break down tasks into logical steps.
- Experiment and iterate when responses aren't ideal.

Important

Prompt engineering isn’t just a skill; it’s a mindset. Developers must move from “telling the computer what to do” to “guiding the AI to do what’s needed.”

See Also

For structured guidance on creating prompts, revisit Chapter 3: Writing effective prompts for AI-powered coding.

AI-powered collaboration facilitators

As AI improves collaboration between team members of different technical levels, developers are becoming AI translators and facilitators. They use AI to:

- Convert natural language user stories into technical tasks.
- Help non-developers test and interact with code via Copilot Chat.
- Generate visual prototypes and explain code to stakeholders.

AI can foster better cross-functional teamwork and accelerate delivery across the board.

> **Tip**
>
> Use Copilot Chat to explain code blocks during handoffs or walkthroughs. It bridges the gap for designers, QA engineers, and clients.

Responsible AI champions

In AI-assisted teams, someone must take responsibility for the ethical and legal aspects of using AI. Developers are increasingly stepping into roles where they:

- Assess the compliance of generated code with applicable licenses.
- Document the use of AI tools in a project's audit trail.
- Ensure transparency in how and why AI-related decisions were made.

> **Important**
>
> AI-generated code can sometimes include snippets based on public code that is under restrictive licenses.

Continuous learners and AI integrators

Finally, successful developers in the AI era are defined by continuous learning and experimentation. They:

- Stay current with AI models, updates, and usage patterns.
- Contribute to prompt libraries, open-source AI tools, or community playbooks.
- Learn how to integrate AI into existing CI/CD, testing, or DevOps pipelines.

Use cases for Copilot in DevOps and CI/CD

GitHub Copilot is useful not just in application development but also in DevOps and infrastructure automation. It can assist in writing scripts, creating configuration files, and generating CI/CD workflows. The following table presents examples of prompts and typical outputs that showcase how Copilot can simplify and accelerate DevOps tasks across common platforms.

DevOps Artifact	Copilot prompt	Output
Dockerfile	Dockerfile for Node app	Ready-to-use base image with port and CMD
GitHub Actions	Workflow to deploy Python app on push	CI/CD config with build, test, and deploy stages
Kubernetes	K8s deployment for nginx	YAML spec for deployment and service

DevOps Artifact	Copilot prompt	Output
Terraform	Create S3 bucket	Resource block for AWS S3 with permissions

Tip

Follow GitHub, OpenAI, and Microsoft Learn to stay current on GitHub Copilot updates, prompt improvements, and new tooling integrations.

See Also

For a broader understanding of how AI tools integrate with workflow optimization, refer to Chapter 7: Using GitHub Copilot for code reviews and collaboration.

Embracing vibe coding and AI flow states

The emergence of AI-powered development isn't just changing what we build; it's transforming how we build it. As developers collaborate with AI in real time, the act of coding becomes less mechanical and more expressive. Welcome to the age of vibe coding.

Vibe coding is a new mental model for development where you don't write every line of code but instead compose solutions through prompts, interactions, and flow-state improvisation.

This model captures the shift from traditional, rigid, keystroke-by-keystroke logic into a more fluid and iterative process. Developers using GitHub Copilot often say they feel as if they are:

- Moving faster than they can type.
- Discovering unexpected approaches from AI suggestions.
- Navigating between code and creativity in seamless rhythm.

You can write a comment like this:

```
# Build a dashboard with metrics for user retention, churn ra
```

And in seconds, Copilot generates scaffolding for charts, APIs, or queries. You might accept the first draft, tweak the second, and discard the third, but the collaborative rhythm keeps flowing.

> **Tip**
>
> Treat Copilot's responses as creative sparks. Even "wrong" suggestions can lead to better solutions by revealing new perspectives.

The vibe coding experience encourages:

- Exploration over perfection: Let the first draft be fast and then refine it.
- Playful iteration: Try multiple prompts to compare styles.
- Human–machine flow: Shift between prompting and editing with minimal friction.

> **Important**
>
> While flow-state development is powerful, it's important not to skip critical review. Use Copilot for drafting code but rely on your own expertise to finalize secure, maintainable, and performant code.

See Also

For more on real-world scenarios where Copilot enhances rapid development, refer to Chapter 8: Using AI-powered development workflows in real-world scenarios.

Skills review

In this chapter, you learned how to:

- Explore upcoming AI trends in coding
- Understand the evolution of GitHub Copilot
- Discover new developer roles in the AI era
- Use GitHub Copilot for documentation and learning
- Enhance team collaboration with AI tools
- Integrate GitHub Copilot into CI/CD and DevOps workflows

Practice tasks

No practice files are necessary to complete the practice tasks in this chapter. Follow the steps below to reinforce the concepts and tools introduced in each section.

Use an integrated development environment (IDE) such as Visual Studio Code with GitHub Copilot enabled and signed in. The following practice tasks use Python as the language. You can adapt them to other supported languages as needed.

Explore upcoming AI trends in coding

To explore upcoming AI trends in coding, open Visual Studio Code with GitHub Copilot enabled, and then perform the following tasks:

1. Enter this natural language prompt:

   ```
   Create a function to calculate the factorial of a number
   ```

   Observe the code that Copilot generates.

2. Experiment with prompt variations to generate the same logic in different ways.
3. Try a voice-to-text tool that integrates with your IDE and test how Copilot responds to transcribed spoken input.

Understand the evolution of GitHub Copilot

Open your IDE with GitHub Copilot enabled and create a new blank project (or file) where you can try code in at least two different programming languages.

1. Use Copilot to complete code in two different programming languages and compare the quality of the suggestions.
2. Open Copilot Chat and ask it to explain the logic behind a selected block of code in your project.
3. Explore the GitHub Copilot documentation to identify new features introduced in the latest release.

Discover new developer roles in the AI era

Open your development environment and either create a new project or use an existing one you've worked on recently:

1. Use Copilot to generate a function or component and then refine the output to meet your team's conventions or requirements.
2. Practice writing multiple prompt styles for the same feature and evaluate which version produces the best results.
3. Reflect on how your responsibilities have changed while using Copilot and write down at least two new roles you've taken on (e.g., AI reviewer, prompt engineer).

Index

Symbols and Numerics

A

B

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q-R

S

T

U

V

W

X-Y-Z