

# GITHUB COPILOT FOR DEVELOPERS



# LOGISTICS



## Class Hours:

- Instructor will set class start and end times.
- There will be regular breaks in class.



## Telecommunication:

- Turn off or set electronic devices to silent (not vibrate)
- Reading or attending to devices can be distracting to other students
- Try to delay until breaks or after class

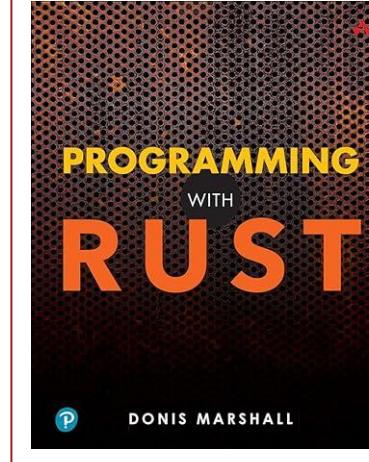
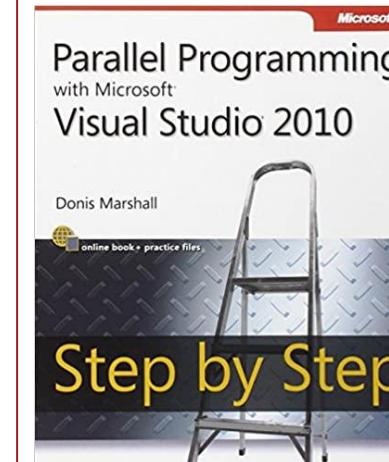
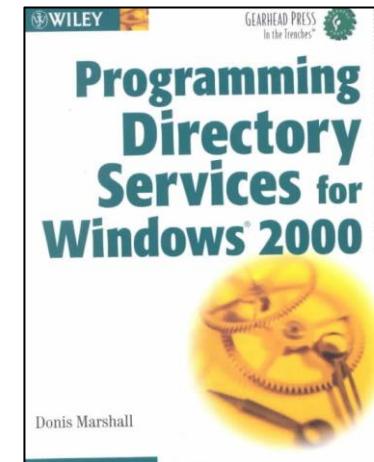
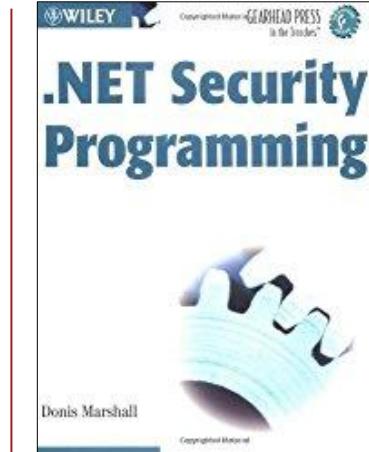
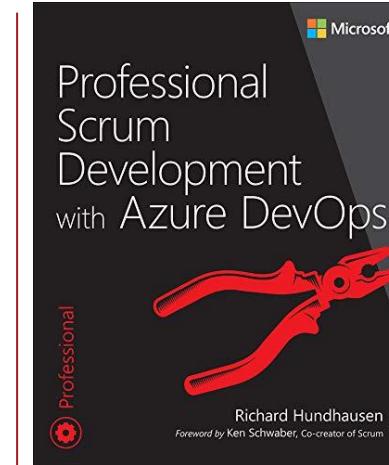
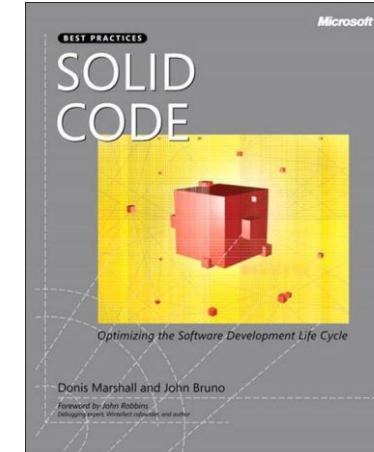
## Miscellaneous:

- Courseware
- Bathroom
- Fire drills

# DONIS MARSHALL

**Security Professional**  
**Microsoft MVP**  
**Microsoft Certified**  
**Author**

[dmmarshall@innovationinsoftware.com](mailto:dmmarshall@innovationinsoftware.com)



# INTRODUCE YOURSELF

Time to introduce yourself:

- Name
- What is your role in the organization
- Indicate Github Copilot and Vscode experience



# COPILOT



In technology, the term copilot refers to a digital assistant that augments a user's capabilities by providing real-time guidance, automation, or suggestions.

It doesn't replace human judgment but instead supplements it, offering context-aware insights and actionable recommendations.

Just as a copilot in aviation supports the pilot with navigation and decision-making, a technology copilot helps professionals carry out complex tasks more efficiently.

# GITHUB COPILOT



GitHub Copilot is an AI-powered coding assistant that integrates into development environments such as Visual Studio Code. It provides real-time code suggestions, entire functions, or even file templates as developers type.

Copilot's strength lies in its ability to understand the context of the current codebase and produce recommendations that feel relevant, saving developers time and effort.

# PLAIN ENGLISH



## GitHub Copilot

At its core, GitHub Copilot is designed to help developers write better, faster, and more consistent code. It leverages AI to understand your intent and recommend completions, functions, or entire modules that align with your project.

Its role in software development is to act as a productivity multiplier, helping developers focus on problem-solving rather than repetitive syntax or boilerplate code.

# GITHUB AND OPENAI



GitHub Copilot emerged in 2021 from a collaboration between GitHub and OpenAI, combining GitHub's massive code base with OpenAI's language model expertise. Built on the Codex model, a GPT-3 descendant fine-tuned for coding, it was designed for programming tasks.

The partnership was strategic: GitHub provided billions of code examples, and OpenAI supplied generative AI research, becoming one of the first large-scale commercial uses of generative AI in software development.

# GENERATIVE AI

Generative AI is a type of artificial intelligence that doesn't just analyze data, but creates new things—like text, images, music, or code—based on patterns it has learned. Instead of giving fixed answers, it generates original output that looks like it was made by a person.

Analogy: Imagine teaching a child to build with LEGO. At first, they copy the instructions you give them. But over time, after seeing many different LEGO sets, they start inventing their own creations—cars, castles, spaceships—by combining pieces in new ways. That's what generative AI does: it studies lots of examples, then uses that knowledge to "build" something new on its own.

## GENERATIVE AI - 2

Generative AI in GitHub Copilot works by looking at patterns in millions of pieces of code it has learned from, then creating new code that fits what you're writing. It doesn't just copy; it predicts what might come next based on context.

Simple example: Suppose you start typing in Python:

```
def add_numbers(a, b):
```

Then pause and wait momentarily.

# GENERATIVE AI - 3

Copilot might automatically suggest:

```
return a + b
```

It “guessed” that since you named the function `add_numbers` with two inputs, you probably want to return their sum. That’s generative AI at work—it takes what you’ve written, combines it with what it has learned, and generates new code for you.

# LLM



An LLM (Large Language Model) is like a giant library in the AI's brain. It has read billions of words, books, websites, and code, so it recognizes many patterns. Because it's so large, it can generate flexible, detailed answers—but it also needs more computing power. GitHub Copilot uses one or more LLMs, which is why it can suggest code in many languages and adapt to different styles.

An SLM (Small Language Model) is more like a pocket guide. It's trained on less data and has fewer "pages" in memory, so it focuses on smaller, specific tasks. It runs faster and uses less computing power, but it won't know as much or be as creative.

# LLM - SOURCES

Many assume GitHub Copilot is trained solely on GitHub repositories, but the reality is broader. Sources include:

- Public GitHub repositories
- Open-source projects hosted outside GitHub
- Programming Q&A sites
- Technical documentation and manuals
- Educational resources such as tutorials and coding books

This variety ensures Copilot can generalize across multiple programming domains and coding conventions, rather than being biased toward a single platform or community.

# CODEX

OpenAI's Codex, released in 2021, is the specialized large language model behind GitHub Copilot. As a descendant of GPT-3, it was fine-tuned for programming using natural language and billions of lines of publicly available code. Codex can take plain English instructions, map them to programming concepts, and generate code in many languages, effectively bridging human intent and machine-readable instructions.

GitHub Copilot, launched the same year, brings Codex into developer workflows through editors like Visual Studio Code. Acting as an AI pair programmer, it suggests lines or blocks of code, completes functions, and generates boilerplate from comments. More than autocomplete, it adapts to coding style and accelerates development. By combining Codex's AI with GitHub's massive code ecosystem, Copilot became one of the first major tools to make generative AI practical for everyday software development.

## LLM – STALENESS



Copilot's training data is not continuously updated. Instead, it often lags one to two years behind current coding trends. This means it may recommend outdated functions, deprecated APIs, or older coding patterns.

To counter this, developers should cross-check suggestions with up-to-date references such as official documentation, blogs, or community discussions. Incorporating these external sources ensures that projects remain aligned with current standards.

# HALLUCINATIONS



Like any generative AI, GitHub Copilot can produce hallucinations—outputs that look correct but are factually or logically wrong. This happens because the AI predicts patterns rather than “understanding” correctness.

For developers, this means Copilot suggestions must always be reviewed carefully. Blindly accepting output can introduce bugs, security risks, or inefficiencies into production systems.a

# POP QUIZ: LINES OF CODE

How many lines of code can the average developer enter before creating a logical or syntactical error?



**10 MINUTES**

# PAIR PROGRAMMING



GitHub Copilot acts as a virtual pair programmer, offering constant feedback, suggestions, and ideas without requiring another human to be physically present. This helps developers explore multiple solutions, avoid common pitfalls, and maintain coding flow.

Pair programming traditionally involves two developers working side by side, one writing code while the other reviews and suggests improvements.

Note: Like a pair programmer, Copilot doesn't always provide the correct answer.

## PAIR PROGRAMMING - 2



When using Copilot, developers must recognize their responsibility in guiding and validating output. If you consistently write poor-quality or insecure code, Copilot may mimic those patterns in its suggestions.

By following good coding standards, documenting clearly, and maintaining quality best practices, developers train Copilot to produce higher-quality suggestions in their specific projects.

# CODE REVIEW



Just as pair programming requires reviewing each other's work, Copilot-generated code must be reviewed. Developers should treat AI suggestions as drafts that require human validation, testing, and refactoring.

This ensures that the final code meets project standards, security requirements, and long-term maintainability goals.

# CODE PILOT CHAT



Copilot Chat extends the traditional completion model by enabling natural language conversations directly within the coding environment. Instead of just typing code and receiving suggestions, developers can ask Copilot questions like, "How do I implement OAuth2 authentication?" or "Explain what this function does." This conversational interface makes problem-solving more intuitive and context-driven.

The importance of Copilot Chat lies in its ability to bridge the gap between coding knowledge and documentation search.

# USE CASES

## Boilerplate and Scaffolding

Copilot accelerates development by generating repetitive setup code such as class definitions, configuration files, or API route handlers. For instance, typing `# create a Flask route for /hello` can instantly produce a functioning endpoint without you writing the boilerplate manually.

## Code Completion and Suggestions

It provides context-aware completions for functions, loops, and conditionals. Start typing `for i in range(` in Python, and Copilot may suggest the entire loop with a sensible body. This allows developers to move faster and maintain momentum.

## Error Handling and Validation

Copilot generates robust error-handling blocks, helping developers avoid common pitfalls. For example, in JavaScript, adding a comment `// handle fetch errors` can prompt Copilot to create a `try { ... } catch (error) { ... }` structure.

# USE CASES - 2

## Learning and Language Assistance

Developers working with unfamiliar frameworks or languages can rely on Copilot to propose idiomatic patterns. Typing # connect to MongoDB may yield a correct connection snippet in Node.js, Java, or Python.

## Test-Driven Development (TDD)

Copilot is excellent when writing tests first. Developers can describe intended behavior in comments or partial test code, and Copilot generates full unit tests or assertions.

# USE CASES - 3

## Finding Logic Errors and Reviewing Code

Copilot is not just a code generator; it can help detect subtle logic mistakes by suggesting corrections or alternative implementations. For example, if a loop doesn't terminate properly or a condition is inverted, Copilot often proposes the correct fix. It can also suggest more efficient or readable versions of code, acting like a lightweight peer review.

## Source Control Assistance

Copilot integrates well with GitHub workflows, helping draft commit messages, suggesting summaries for pull requests, or scaffolding git commands in scripts.

# POP QUIZ: USE CASES

What other use cases, if any, are important to mention?



**10 MINUTES**



**FLEXIBLE / ADAPTABLE**

# LLM MODELS

GitHub has expanded Copilot beyond just OpenAI's GPT family. It now also supports models from Google (Gemini) and Anthropic (Claude).

LLM Family	Examples in Copilot
OpenAI	GPT-4.1, GPT-5 mini, GPT-5, o3, o4-mini
Anthropic (Claude)	Claude Sonnet 3.5, 3.7, 4, Claude Opus 4 & 4.1
Google (Gemini)	Gemini 2.0 Flash, Gemini 2.5 Pro

# LANGUAGES SUPPORTED

GitHub Copilot supports a wide range of programming languages.

Language	Language
Python	JavaScript / TypeScript
Java	C#
Go	Ruby
PHP	C++
Rust	SQL
Shell scripting (Bash, PowerShell)	—

# IDE SUPPORTED

Copilot currently integrates with:

- Visual Studio Code
- Visual Studio (for .NET developers)
- Neovim
- JetBrains IDEs (IntelliJ IDEA, PyCharm, WebStorm, etc.)
- Eclipse

Note: Level of integration and support may vary. Be sure to check reference documentation.

# COURSE AGENDA

1. Introduction to GitHub Copilot
2. Copilot setup and configuration
3. Basic code completion
4. Context-aware code suggestions
5. Debugging with Copilot
6. Writing functions and modular code
7. Code documentation
8. Collaboration with Copilot
9. Introduction to testing
10. Basic CI/CD concepts
11. Capstone project (optional)

# VISUAL STUDIO CODE



Visual Studio Code (Vscode) is a lightweight yet powerful editor developed by Microsoft. It supports many programming languages and can be extended with thousands of extensions. Core features include IntelliSense code completion, built-in debugging, and Git integration, making it useful for .

A major strength of VS Code is its close integration with GitHub and GitHub Copilot. Developers can commit, push, and review code directly in the editor while Copilot provides AI-powered suggestions inline.

# GITHUB ACCOUNT

A GitHub account is required to use GitHub Copilot.

- Subscription management: Copilot is a paid service (with free trials for students and certain accounts). GitHub needs your account to handle billing or free access eligibility.
- Authentication: You sign in with your GitHub account in your IDE (VS Code, JetBrains Visual Studio) to connect to Copilot's cloud models.
- Settings & preferences: Your GitHub account stores Copilot settings, like language/file-type controls, telemetry, and whether Copilot Chat is enabled.
- Enterprise access: For business/enterprise users, Copilot is tied to an organization's GitHub account and licensing.

# GITHUB LICENSE

GitHub Copilot is a subscription service linked to your GitHub account. Plans vary for individuals, teams, and enterprises, each with different features and management controls.

Plan	Intended For	Key Features
Copilot Individual	Solo developers	Code completions, Copilot Chat, personal settings
Copilot Business	Teams & organizations	Includes Individual features + policy controls, seat management
Copilot Enterprise	Large organizations	Includes Business features + enterprise context in Chat, advanced management

# LABS

Learning is better when hands-on. Some of the modules have a companion lab reinforcing a kinesthetic learning experience.

- Labs review and reinforce important concepts
- Don't be surprised - many of the labs extend the topics introduced in the module
- The labs offer an opportunity for real-world experience
- Pair programming can be effective when working on the labs

# YOUR CLASS!

Yes, this is your class. What does this mean? You determine the value.

- What is the most important ingredient of class – your participation!
- Your feedback and questions are always welcomed.
- There is no protocol in class. Speak up anytime!
- We welcome your comments during and after class.  
Just email [dmarshall@innovationinsoftware.com](mailto:dmarshall@innovationinsoftware.com).

# CHATGPT



ChatGPT can be used similar to GitHub Copilot — both rely on large language models to generate context-aware code and explanations.

While Copilot is tightly integrated into the coding workflow inside IDEs like VS Code,

ChatGPT excels at broader tasks such as debugging discussions, architectural guidance, and natural language exploration of problems.

## CHATGPT - 2



GitHub Copilot's training is focused on public source code, especially GitHub repositories. It's optimized to recognize patterns in real-world examples, libraries, and frameworks, making it effective at suggesting boilerplate, API usage, and common idioms.

ChatGPT takes a broader approach, trained on both programming and natural language text (books, docs, forums). This makes it stronger for explaining concepts, providing architectural reasoning, or step-by-step debugging.

# CHATGPT - 3

Aspect	Copilot	ChatGPT
Integration	Built into IDEs (VS Code, JetBrains).	Web/app chat; extensions exist.
Focus	Inline code completion.	Broader Q&A and reasoning.
Context	Uses nearby code (FIM).	Uses conversation history.
Ease	Fast, automatic as you type.	Manual prompts, copy-paste code.
Strengths	Scaffolding, TDD, fixes.	Explaining, design, reviews.
Pros	Seamless coding flow.	Detailed answers, flexible.
Cons	Limited to coding tasks.	Less integrated with IDEs.

# LAB 1 – PROGRAMMING



# FACTORIAL

A factorial is a mathematical operation that multiplies a whole number by every positive whole number smaller than it, down to 1.

It is written with an exclamation point. For example:

5! (read “five factorial”) means:

$$5 \times 4 \times 3 \times 2 \times 1 = 120$$

3! is:

$$3! = 3 \times 2 \times 1 = 6$$

0! is defined as 1 by convention, which helps formulas work consistently.

# CONFIRM PYTHON

## 1. Open a terminal

On Windows: open Command Prompt (cmd) or PowerShell

On macOS/Linux: open the built-in Terminal app.

## 2. Check the version

`python --version`

If that doesn't work, try: `python3 --version`

You should see something like: Python 3.11.6

If you get an error such as "command not found", Python isn't installed. Download it from and install it before continuing:

[python.org/downloads](https://www.python.org/downloads/)

# CONFIRM PYTHON EXTENSION

1. Open Visual Studio Code.
2. Go to the Extensions view
3. Click the Extensions icon on the left sidebar (it looks like four squares).

Or press Ctrl+Shift+X (Windows/Linux) or Cmd+Shift+X (macOS).

Search for “Python” the official extension (Python from Microsoft)

Install the extension

# FACTORIAL-LAB FILE

Create a folder for the lab

- VS Code → File → Open Folder...
- Create and select a folder (factorial-lab).
- Within the new folder, create a new file (factorial.py)

# FACTORIAL FUNCTION

1. In the file, Import sys to access command-line arguments via sys.argv:

```
import sys
```

2. Define the factorial function. Then pause - the code implementation should appear. Tab to accept.

```
def factorial(n):  
    result = 1  
  
    for i in range(1, n + 1):  
        result *= i  
  
    return result
```

# MAIN FUNCTION

Create main function and after the function header add these comments "#" as scaffolding

1. Checks that exactly one argument (the number) is provided.
2. Converts that argument from a string to an integer.
3. Calls the factorial() function and print the result in the format n! = result.

Github Copilot should generate the main code. Tab to accept.

```
def main():

    if len(sys.argv) != 2:

        print("Usage: python factorial.py <number>")

        sys.exit(1)

    num = int(sys.argv[1])

    print(f"{num}! = {factorial(num)}")
```

# TEST AND VALIDATE

Open Vscode terminal. Make sure factorial-lab is the current directory. Test the program:

Windows (PowerShell):

```
python factorial.py 10
```

macOS/Linux

```
python3 factorial.py 10
```

Expected:

10! = 3628800 (num) }

# Lab completed



# FUNDAMENTALS

DETAILS AND WORKFLOW



# INTRODUCTION



The module outlines GitHub Copilot as a cloud-based AI coding tool. It explains how prompts are enriched with context from files, functions, and project history, enabling features like fill-in-the-middle for relevant code generation.

The workflow includes creating a prompt, collecting context, secure cloud processing, and returning results as inline suggestions or chat responses.

It also highlights privacy, licensing, and security.

# CLOUD-BASED GENERATIVE AI TOOL

GitHub Copilot is not just a local IDE plugin—it is a cloud-based service that relies on generative AI models hosted and maintained by GitHub and Microsoft. Every time a developer enters a prompt or writes code, that context is securely transmitted to cloud servers where the model processes it and returns suggestions.

Being cloud-based ensures scalability and continuous improvement. The models can be updated, retrained, and fine-tuned without requiring developers to manually install updates.

# PROMPT

Prompts include more than what you type.

When you type a comment or partial line of code, that is only part of the prompt. Copilot automatically enriches it with additional context from the surrounding code, open files, and repository history. This means your “real prompt” is much larger than the few words you typed, leading to more accurate predictions.

This is why Copilot can sometimes anticipate what you want several lines ahead—it is interpreting not just your last keystrokes but the broader coding environment.

# CONTEXT



Context is critical to how GitHub Copilot works. The tool doesn't just generate random code—it looks at the file you're writing, the function names, other files you have open, etc. This context tells Copilot what you're trying to build, so its suggestions are relevant. Without context, its output would be generic and often wrong.

For example, if you write a comment saying `# function to calculate tax`, Copilot uses that clue along with your code structure to suggest a tax calculation function instead of something unrelated.

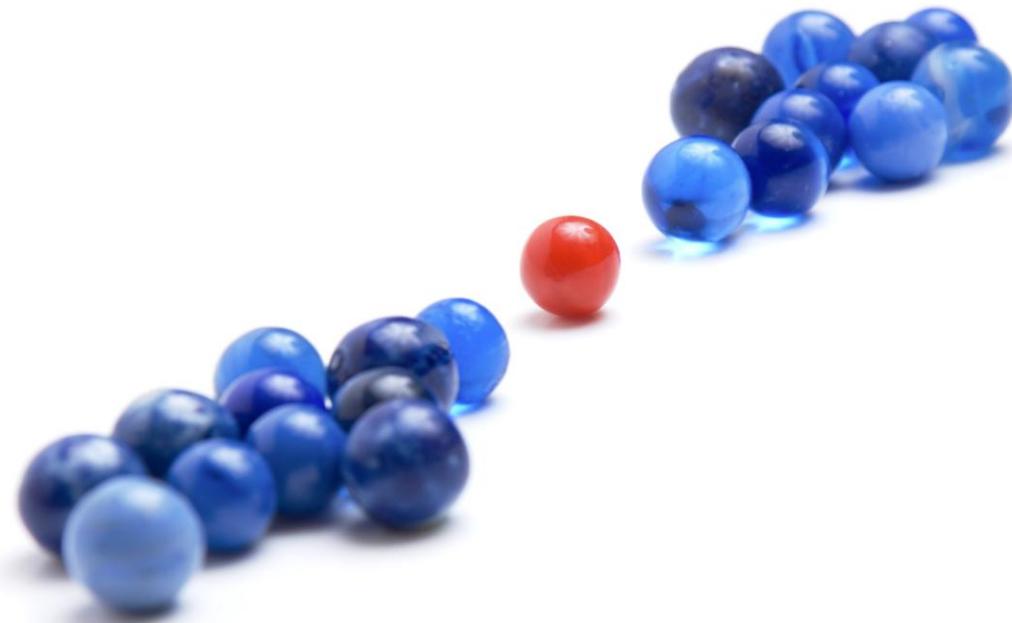
## CONTEXT - 2



Copilot gathers the context from multiple sources, some more influential than others.

- Current file content
- Function and variable names
- Comments and docstrings
- Other open files in the editor
- Project-level context (local index)
- File name and file type
- Coding conventions

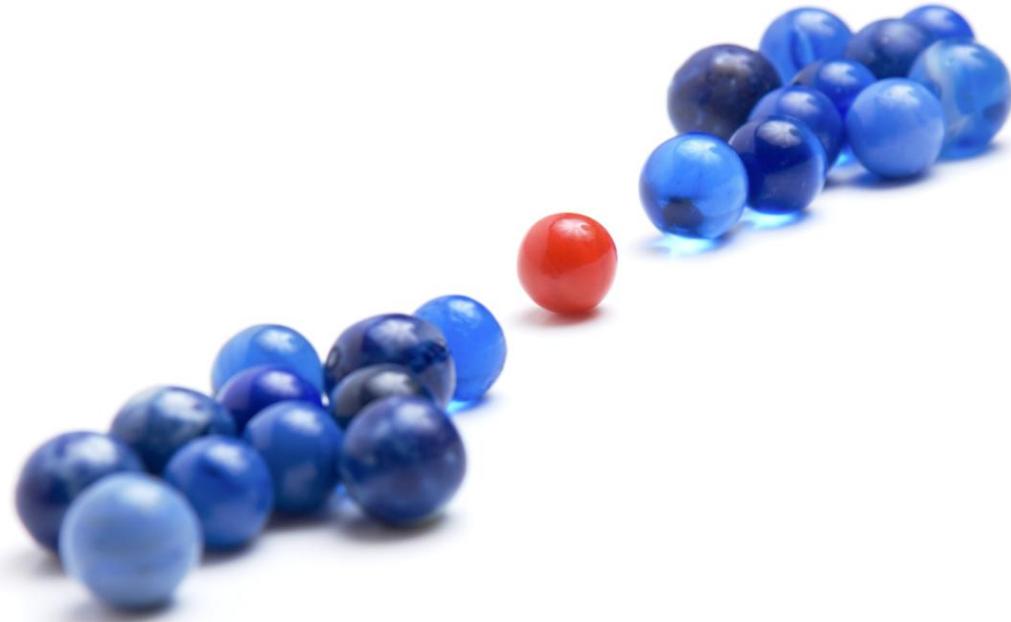
# FILL-IN-THE-MIDDLE



Copilot looks at both the code before and after the cursor position to generate a relevant suggestion. This is called Fill-in-the-Middle (FIM) and means it doesn't just rely on what you've already typed—it also pays attention to the surrounding context to "fill the gap."

For example, if you have a function stub with a comment describing what it should do, Copilot can fill in the function body based on that description and any hints from code above or below.

## FILL-IN-THE-MIDDLE - 2



FIM is particularly powerful for scaffolding code, refactoring, or when you're editing large files. Instead of writing code strictly top-to-bottom, you can leave placeholders or comments in the middle of a source file and let Copilot intelligently fill them in.

This encourages a workflow where you guide Copilot with intent signals (comments, partial code, or function headers), and it provides the in-between logic.

# FIM EXAMPLE

Here is a FIM example:

```
def add_numbers(a, b):  
    # return the sum of a and b
```

If your cursor is inside the function, Copilot may suggest:

```
return a + b
```

Because of FIM, Copilot sees the function definition above, the comment, and the empty body, and generates the missing piece.

# COPilot WORKFLOW – STEP 1 - CREATE A PROMPT

A prompt can be:

- Code you begin typing
- A function name or variable name
- A comment describing what you want
- Even a natural language question inside Copilot Chat

Copilot uses this as the main clue for what to generate next.

## STEP 2 – COPILOT COLLECTS CONTEXT

Before sending anything to the cloud, Copilot gathers context from your environment:

- The current file (especially lines near your cursor)
- Other open files in your editor
- Comments and docstrings
- The project's structure and imported libraries
- File names and file type (Python, JavaScript, HTML, etc.)

This helps Copilot understand not just what you typed, but where it fits in your project.

# STEP 3 AND STEP 4

## Step 3: Prompt + context sent securely

The prompt and gathered context are encrypted and sent to GitHub's servers. This ensures that your data is transmitted securely, never in plain text.

## Step 4: AI model generates results

On GitHub's side, one of the supported LLMs (such as GPT-4, GPT-4o, Gemini, or Claude, depending on your settings) processes the input.

- The model generates one or more code suggestions.
- Optional post-processing may occur, such as basic vulnerability checks or formatting adjustments, before sending results back.

## STEP 5 – RESULTS RETURNED TO IDE

The generated suggestion flows back into your IDE (e.g., Visual Studio Code, JetBrains). You'll see it as:

- An inline completion (gray “ghost text”)
- A list of alternative suggestions
- A full explanation or snippet in Copilot Chat

You can accept, modify, or reject the suggestions as needed.

## STEP 5 – RESULTS RETURNED TO IDE

The generated suggestion flows back into your IDE (e.g., Visual Studio Code, JetBrains). You'll see it as:

- An inline completion (gray “ghost text”)
- A list of alternative suggestions
- A full explanation or snippet in Copilot Chat

You can accept, modify, or reject the suggestions as needed.

# PRIVACY



Privacy is a major consideration when using AI-assisted coding tools. Copilot may collect snippets of your code to provide context, raising concerns about sensitive or proprietary data. GitHub provides an option in settings to disable code retention for training purposes, giving organizations more control over their intellectual property.

You should configure these options carefully to balance functionality with privacy and compliance requirements.

## PRIVACY - 2



User engagement—such as whether you accept, modify, or reject suggestions—feeds back into the learning process for Copilot. This helps the system refine its outputs over time. However, it's important to note that feedback is aggregated and anonymized.

This engagement loop ensures that Copilot becomes more useful the more it is used.

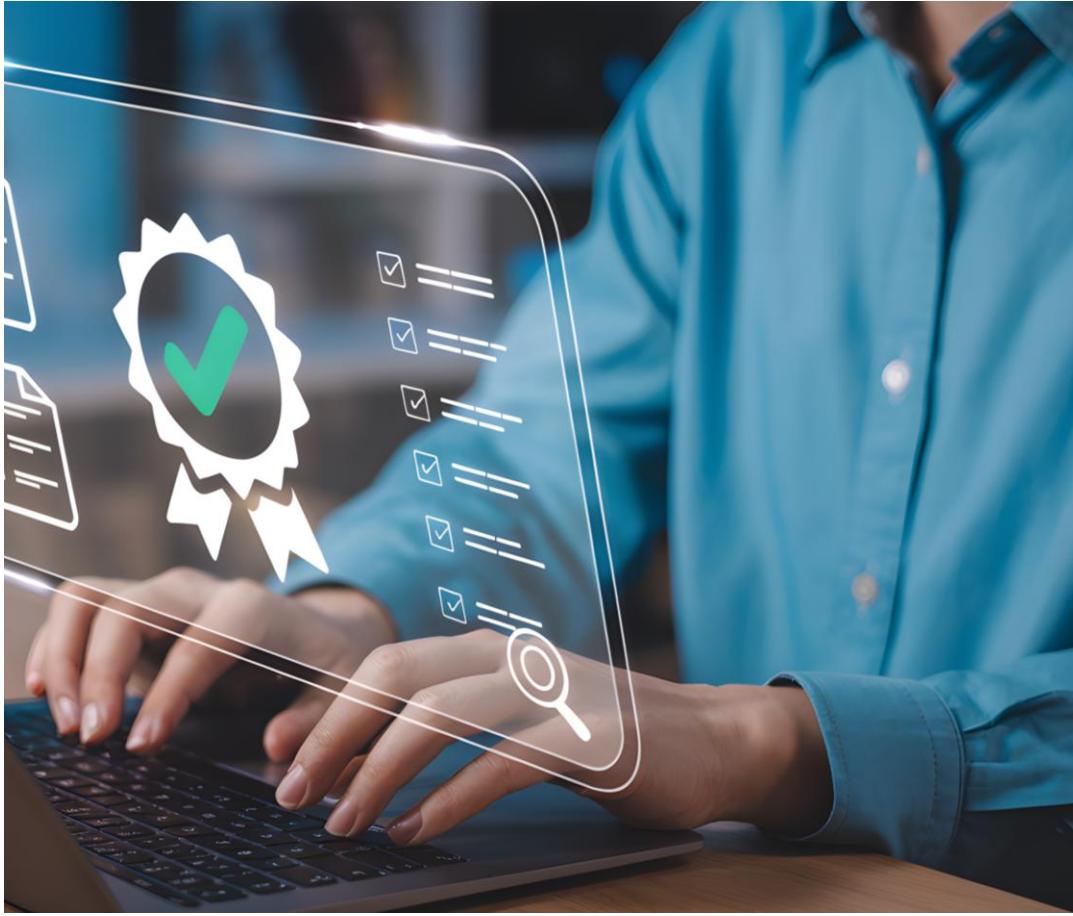
# LICENSES CONTENT



Another concern is that Copilot's training data may include code under restrictive licenses. While GitHub asserts that Copilot generates new content rather than copying directly, there is debate over whether certain outputs might inadvertently resemble copyrighted or unlicensed code.

Developers should be aware of these risks and watch for potential license compliance issues. GitHub has added features such as filtering to block suggestions that match public code verbatim.

## LICENSES CONTENT - 2



- Verbatim: blocked if identical.
- Modified: same logic, different style.
- Concept-only: inspired by the idea but applied differently.

# SECURITY

The prompt and code should never include sensitive data such as passwords, API keys, or confidential business logic. Even though Copilot transmits information securely, providing such details still increases the chance of exposure or accidental misuse. Developers should treat all prompts and context as if they may be seen outside the immediate coding session.

AI-generated code can also introduce security vulnerabilities, including insecure SQL queries without parameterization, weak authentication flows, or outdated cryptographic functions. To help, GitHub has added post-processing checks like vulnerability detection and filtering for unsafe code patterns.

These safeguards are limited. The primary responsibility for security remains with the development team, who must carefully review every suggestion, apply secure coding practices, and ensure that Copilot is used as a supportive tool rather than a replacement for sound engineering judgment.

# SCOPE OF SUGGESTIONS

This is about where and what Copilot can suggest:

## File scope:

Copilot considers the active file and lines around your cursor most heavily.

## Project scope:

It can look at other open files in your editor session. Copilot can also consider project-level context, such as imports or dependencies.

# SCOPE OF SUGGESTIONS - 2

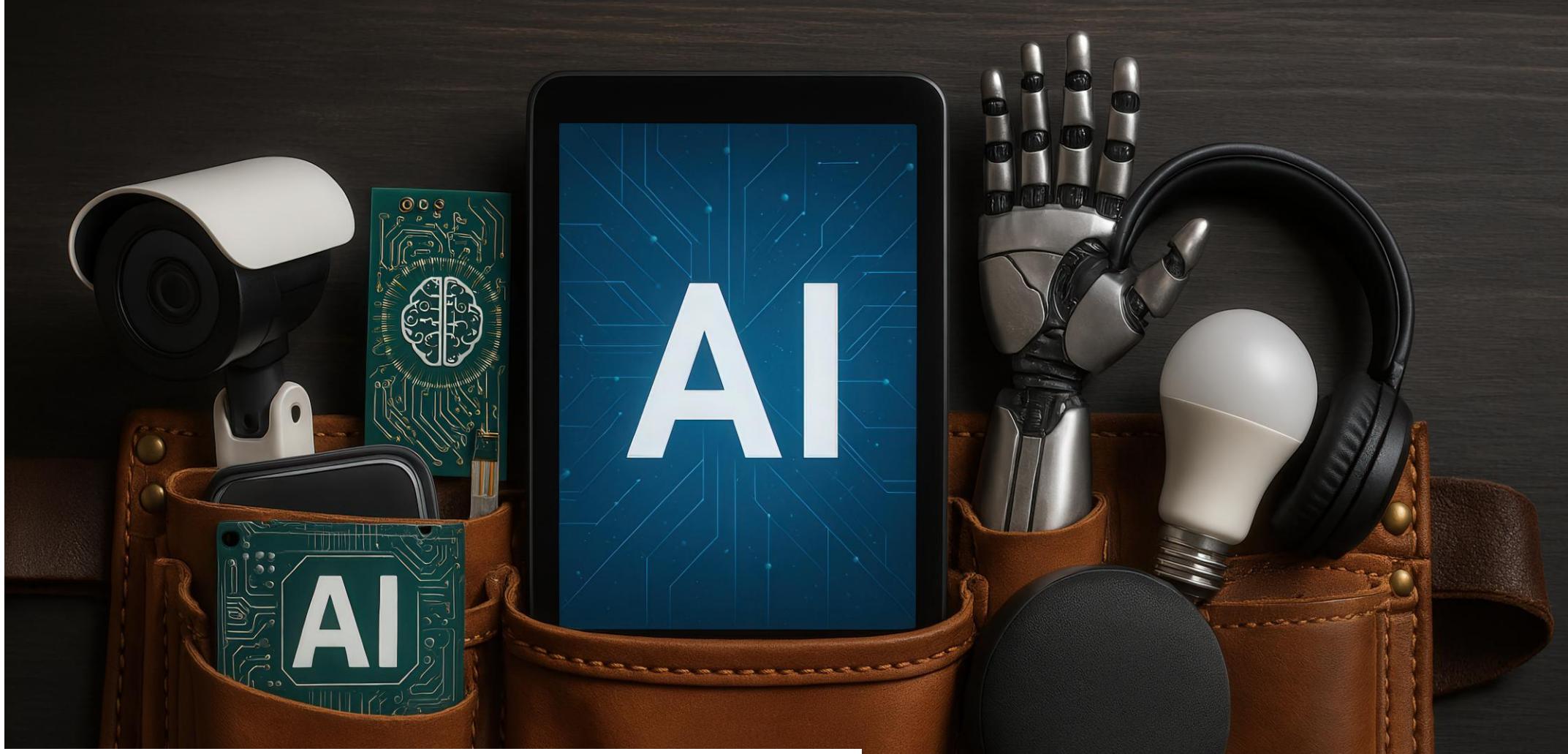
## Prompt scope:

Comments or function names you type strongly shape suggestions.

Example: typing # function to calculate factorial will cause a factorial implementation suggestion.

## Language scope:

Copilot supports many languages, but scope is controlled — you can enable or disable suggestions by language or file type.



## SETTINGS / CONFIGURATION

# PRIVACY

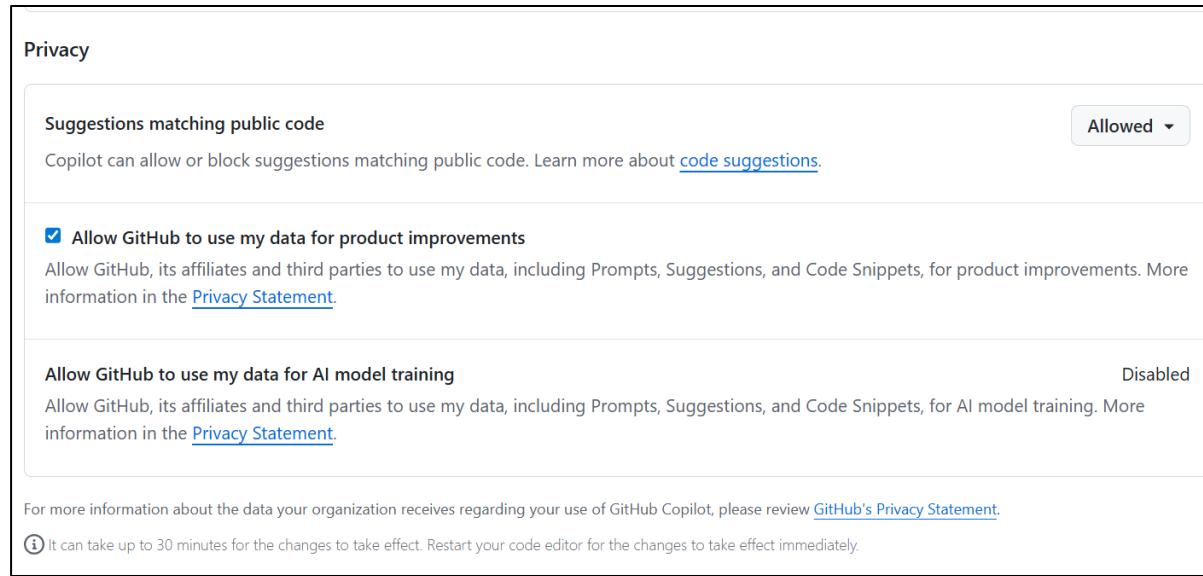
Privacy panel in GitHub Copilot settings controls how your code and interactions with Copilot are handled.

Suggestions matching public code: You can choose whether Copilot is allowed to show you completions that are very similar to existing public code on GitHub. Blocking this reduces the chance of receiving verbatim code from open repositories.

Allow GitHub to use my data for product improvements: If enabled, GitHub may use your prompts, suggestions, and snippets to analyze how Copilot is used, fix issues, and improve features.

Allow GitHub to use my data for AI model training: If enabled, your prompts and snippets could also be used to further train GitHub's AI models. With it disabled, your data won't be included in training.

# PRIVACY - 2



The screenshot shows the 'Privacy' section of the GitHub Copilot Settings. It includes two main sections: 'Suggestions matching public code' (Allowed) and 'Allow GitHub to use my data for product improvements' (Enabled). Below these are sections for 'Allow GitHub to use my data for AI model training' (Disabled) and general organization data (GitHub's Privacy Statement).

**Privacy**

Suggestions matching public code      Allowed ▾

Allow GitHub to use my data for product improvements

Allow GitHub, its affiliates and third parties to use my data, including Prompts, Suggestions, and Code Snippets, for product improvements. More information in the [Privacy Statement](#).

Allow GitHub to use my data for AI model training      Disabled

Allow GitHub, its affiliates and third parties to use my data, including Prompts, Suggestions, and Code Snippets, for AI model training. More information in the [Privacy Statement](#).

For more information about the data your organization receives regarding your use of GitHub Copilot, please review [GitHub's Privacy Statement](#).

ⓘ It can take up to 30 minutes for the changes to take effect. Restart your code editor for the changes to take effect immediately.

How to get to the Privacy panel.

- Log in to GitHub
- Go to [github.com](https://github.com) and sign in.
- Open Copilot Settings
- Click your profile picture (top-right).
- Select Your Copilot.
- Scroll down right panel to find Privacy

# ENABLE / DISABLE COMPLETION

The enable/disable completions toggle in GitHub Copilot controls whether inline code suggestions appear while you type. You can set it globally, per language, or for specific file types.

When completions are enabled, you get faster coding through boilerplate generation, context-aware help, and exposure to idiomatic patterns in unfamiliar languages. It feels like a pair-programmer suggesting alternatives. The downside is potential over-reliance, clutter from too many suggestions, and the risk of accepting inaccurate or irrelevant code.

## ENABLE / DISABLE COMPLETION – 2

When completions are disabled, you retain full control of your code with fewer distractions and better focus, which can be useful for teaching or training. However, you lose the productivity boost, brainstorming value, and language guidance Copilot provides.

You can still manually request suggestions. For VS Code, press Alt+\ (Windows/Linux) or Option+\ (macOS) to trigger Copilot inline suggestion.

You can also use Ctrl+Shift-P and Github Copilot: Open Completions Panel (Windows/Linux) or Cmd+Enter (macOS) to open the Completions panel and see multiple alternative suggestions..

# COMPLETION PANEL

The screenshot shows a code editor interface with a completion panel open. The panel displays seven different code snippets (Suggestion 4 to Suggestion 7) for implementing a palindrome check function. Each suggestion includes a 'Accept suggestion X' button below it.

```
Suggestion 4
def is_palindrome(s: str) -> bool:
    """Check if the given string is a palindrome."""
    s = ''.join(filter(str.isalnum, s)).lower() # Normalize the string
    return s == s[::-1] # Check if the string is equal to its reverse

Accept suggestion 4

Suggestion 5
def is_palindrome(s: str) -> bool:
    """Check if the given string is a palindrome."""
    s = s.lower().replace(" ", "") # Normalize the string
    return s == s[::-1] # Compare the string with its reverse

Accept suggestion 5

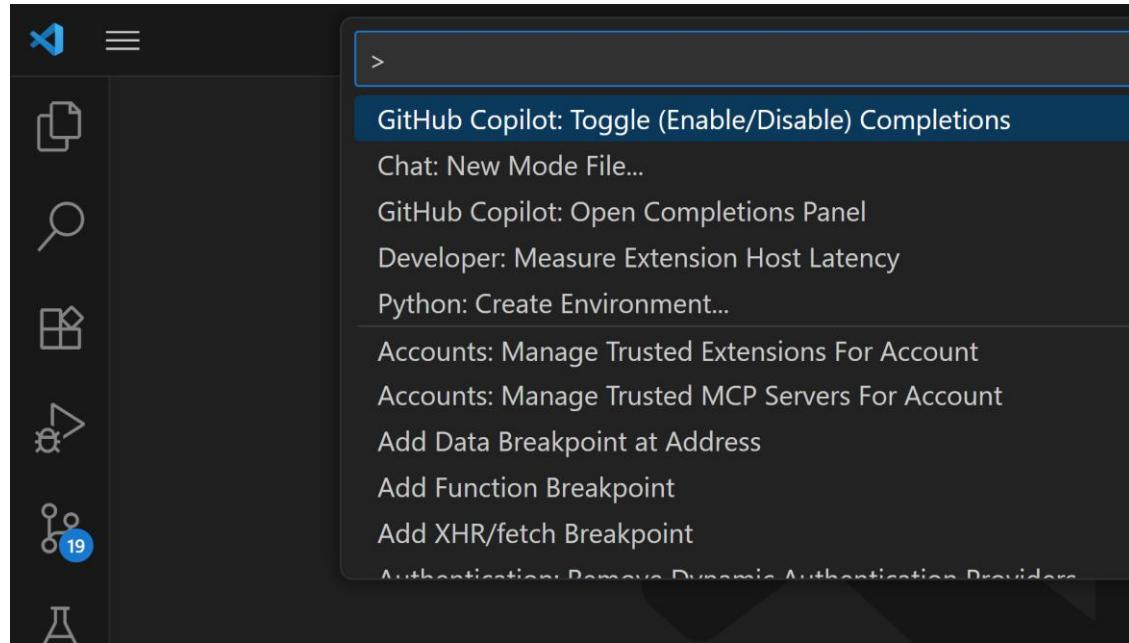
Suggestion 6
def is_palindrome(s: str) -> bool:
    """Check if the given string is a palindrome."""
    # Normalize the string by removing spaces and converting to lowercase
    normalized_str = ''.join(s.split()).lower()
    # Check if the normalized string is equal to its reverse
    return normalized_str == normalized_str[::-1]

Accept suggestion 6

Suggestion 7
```

Inline ghost text shows only one suggestion, while the completion panel offers several alternatives side by side. This helps you explore different idioms, like recursive, iterative, or library-based approaches. You can then accept, reject, or cycle through suggestions with the arrow keys for more control.

# ENABLE / DISABLE COMPLETION – 3



Ctrl-Shift-P and GitHub Copilot: Toggle  
Enable / Disable Completions

# DISABLE FOR A LANGUAGE

Disabling GitHub Copilot for certain languages is especially helpful when you need more control over security, focus, or code quality.

For sensitive or non-code files (such as .env, yaml, or markdown), it reduces the chance of leaking secrets, introducing misconfigurations, or cluttering documentation with unnecessary suggestions. In languages you already know well, it minimizes distractions and ensures that the code you write remains intentional. For teaching, learning, or secure environments, it helps avoid compliance risks, encourages independent problem-solving, and ensures AI isn't interfering where strict accuracy is required.

In short, turning Copilot off selectively gives you a better balance of productivity, safety, and intentional coding—leveraging AI where it's valuable

# DISABLE FOR A LANGUAGE

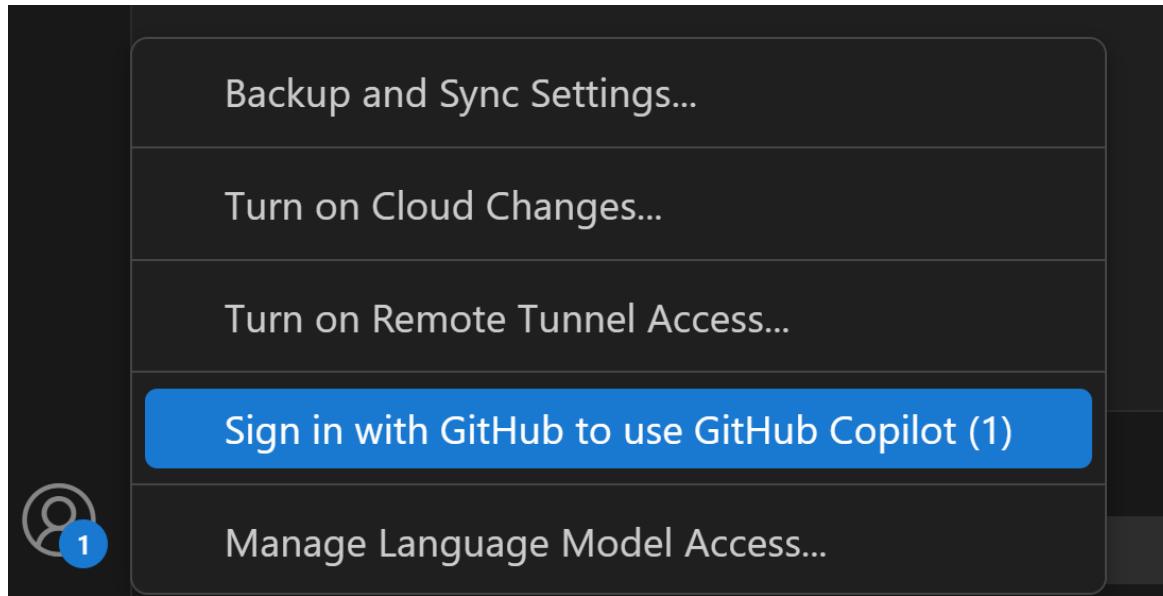
In VS Code, control this through your settings.json:

- Open Command Palette (Ctrl+Shift+P / Cmd+Shift+P).
- Type “Preferences: Open User Settings (JSON)” and select it.
- Add or adjust the github.copilot.enable section.

```
"github.copilot.enable": {  
    "*": true,           // enable by default  
    "plaintext": false, // disable for plain text  
    "markdown": false,  // disable for markdown files  
    "yaml": false,       // disable for YAML files,  
    "typescript": false // disable for typescript language  
}
```

This ensures Copilot won’t suggest completions in sensitive or non-code files.

# GITHUB ACCOUNT - LOGIN



1. In VS Code, look at the bottom left corner.
2. You'll see an Accounts icon (a little person silhouette). Click it.
3. Select Sign in with Github to use Github Copilot.
4. Alternatively, the Copilot status menu at the bottom right of the window in the status bar.

# RETROSPECTIVE



This module showed how GitHub Copilot works as a cloud-based AI assistant, using context and fill-in-the-middle to generate relevant suggestions. We walked through its workflow—from prompts and context gathering to secure cloud processing and IDE results—highlighting that Copilot is more than code completion; it adapts to the developer's environment.

We also stressed responsibility and safeguards. Copilot boosts productivity, but privacy, licensing, and security require careful configuration and adhering to best practices.

# LAB 2 – VS CODE



## SAY HELLO



Use GitHub Copilot in VS Code to generate a Python program that displays “Hello, World”. Learn how to use comments, accept Copilot suggestions, and choose from alternatives.

# SETUP

- Open Vscode
- In Vscode, login to Github using your assigned Github account for Github Copilot
- Create a new folder called:  
say-hello-world
- Create a Python File:  
say-hello-world.py

# CREATE AND RUN PROGRAM

1. In the empty file, type a Python comment to display "hello, world". Make it short, concise, and direct.
2. After typing the comment, press Enter if the suggestion (ghost text) is not automatically provided.
3. Enter tab to accept the suggestion
4. Explore Alternatives (if shown)
  - Use the arrow keys to move through alternatives.
  - Press Tab to accept the one you like.
5. Run the simple program

# EXTEND HELLO

1. Delete the code
2. As multiple discrete comments, add the instructions to display hello in four different languages (your choice of language). Use natural language.
3. The comments should be generic not technical
4. Add any other comments you feel is necessary.

Run the program.

# Lab completed



"Develop a passion for learning."

# CODE COMPLETION

## BASICS AND BEST PRACTICES



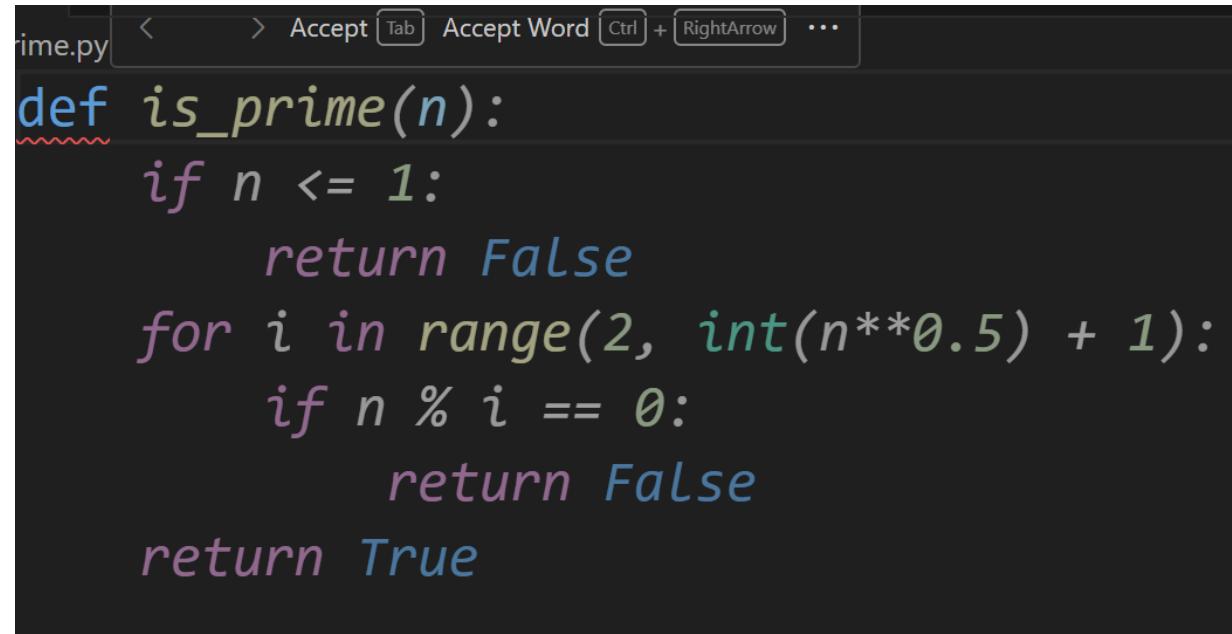
# INTRODUCTION



The deck is a practical guide to using GitHub Copilot completions effectively. It explains ghost text and “getting started” habits (clear intent, pause for suggestions, accept/cycle), then shows how to sharpen results with context: meaningful names, comments/docstrings, and even adding imports to steer libraries.

It walks through scaffolding patterns (single- and multi-line), partial code guidance, and best-practices.

# GHOST TEXT



A screenshot of a code editor window titled "prime.py". The code defines a function to check if a number is prime. The word "prime" is underlined with a red wavy line, indicating it's a suggestion. Above the code, there is a toolbar with buttons for "Accept", "Accept Word", and keyboard shortcuts "Ctrl + RightArrow" and "...". The code itself is:

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Copilot predicts what you might type next, showing it as italic gray text. It appears as faint, gray, italicized text that appears inline in your editor as a code suggestion before you accept it.

- The text isn't "real code" yet — it won't run, save, or commit until you accept it.
- Accepting with a keystroke can save you multiple lines of typing.
- Safe to ignore: If you keep typing, the italic ghost text disappears automatically.
- Options: You can cycle through alternatives (next/previous suggestion) if more than one is available.

# GETTING STARTED

Best practices for code completion:

- Start with a clear intent signal: type a descriptive comment or a function signature (e.g., `# print "hello, world"` or `def add(a, b):`).
- Pause after a newline—Copilot will show ghost text. Press Tab to accept, or use arrow keys to cycle alternatives (if offered).
- Prefer small, incremental prompts: write the next line or two, accept, run, repeat. Short loops of type → accept → test keep suggestions on track.
- Use meaningful names (files, functions, variables). Better names → better suggestions.

# GUIDANCE

Provides context:

- Function and variable names (calculate\_tax, db\_client).
- Comments/docstrings that state what and how.
- Imports and existing patterns in your file.

If suggestions drift, narrow the context:

- Add a clarifying comment.
- Rename ambiguous identifiers.
- Close irrelevant files or move code into a focused file.

# DOCSTRING

```
def division(a, b):
    """ This function divides two numbers.
        Before dividing check
        if the denominator is zero.
    """
    return a / b
```

In Python, docstrings (documentation strings) are special strings used to explain what a module, class, function, or method does. They serve as in-code documentation and are written inside triple quotes ("""" or """), immediately after the definition.

- A docstring is the first statement inside a function, method, class, or module.
- It's stored in the object's `__doc__` attribute.
- Unlike regular comments (#), docstrings are part of the object at runtime, so tools and developers can access them.

# POP QUIZ: CONTEXT

On the previous slide, how could the context be improved?



**10 MINUTES**

# IMPORT

This example may need an import to prevent ambiguity and the wrong choice. Without an Import:

```
def generate_random_number():
    """
    Generate a random integer between 1 and 10.
    """

```

Copilot may not know whether to use random, numpy, secrets, or another library. This could lead to different approaches: numpy, random, randint, secrets.rand, or random.randint.

# COMMENTS

A second comment can help GitHub Copilot by narrowing the context and reducing ambiguity. While one comment may describe the general task, adding another can specify how it should be done, what tools or techniques to use, or what constraints to follow.

```
# Calculate factorial of a number
# Use recursion
def factorial(n):
```

# SCAFFOLDING

Writing basic functions and scripts with scaffolding in comments. Wait for a moment and if not presented immediately then press Enter to invite a suggestion. For example:

```
# read a csv file and print first 5 rows
```

Guide with signatures:

```
def slugify(text: str) -> str:
```

Add a short docstring to steer edge cases (whitespace, punctuation).

# MULTI-LINE SCAFFOLDING

```
# Step 1: define a function to calculate the area of a circle  
# Step 2: use the formula area = pi * r^2  
# Step 3: import math for pi  
# Step 4: prompt user for a radius and display the result
```

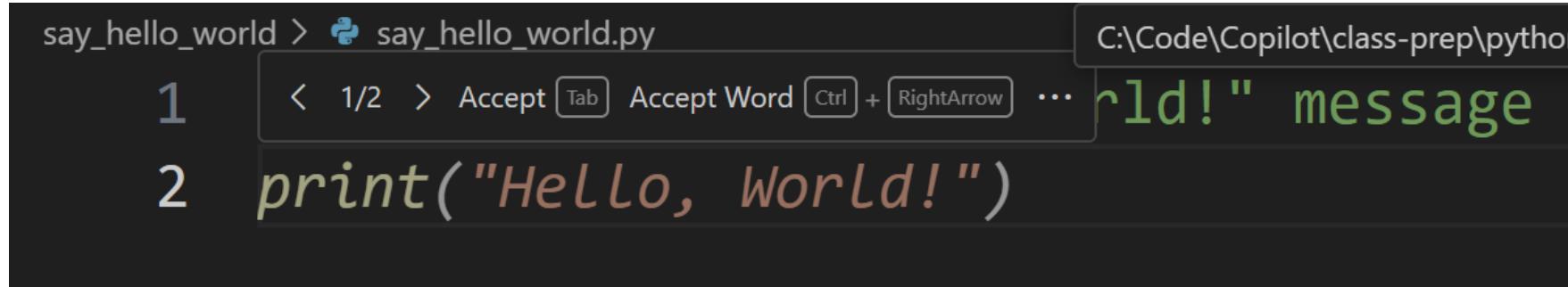
# PARTIAL CODE GUIDANCE

Partial code guidance means you don't have to fully describe or write the whole solution—just give GitHub Copilot a hint through a short comment, function signature, or partial line of code. Copilot will then try to complete it based on the context.

```
# Example 1  
# Return the largest number in a list  
def find_max(numbers):
```

```
# Example 2  
# Print numbers 1 to 5  
for i in range(
```

# ALTERNATIVES BAR



- Arrows (◀ ▶): to cycle through different completion sources (for example, between Copilot's inline suggestion, IntelliSense, or snippets).
- Accept (Tab): indicates you can press Tab (or Enter) to accept the current suggestion.
- Word: switches the suggestion source to word-based completions (from text in the open file).

# ALTERNATIVES BAR - 2



**Accept Line.** Let's you accept just the current line of Copilot's suggestion instead of the entire multi-line completion. This is handy when Copilot offers a large block of code.

**Always Show Toolbar.** Toggles whether the Alternatives Bar is always visible or only appears on hover.

**Send Copilot Completion Feedback.** Opens GitHub's feedback mechanism so you can give a quick thumbs-up/down and optionally share comments.

# ITERATE



Building iteratively with Copilot (**minimum prompt → accept → run → refine**) gives you tighter control, fewer wrong turns, and faster feedback.

Benefits: you steer suggestions with small “intent signals,” quickly test each piece, and use follow-up comments/docstrings to nudge edge cases (rounding, errors, performance) without rewriting everything.

# BEST PRACTICES

Here are some best practices for Github Copilot:

- Correctness: Does it meet your intent?
- Security: Parameterize queries, validate input, avoid secrets in code.
- Style: Matches your project conventions?
- Edit actively: If it's 70% right, accept, then fix the rest—this teaches Copilot your style.

## BEST PRACTICES - 2

- Request alternatives: Add/adjust comments or partially type the target line to reshape the suggestion.
- Keep prompts safe: Don't include sensitive data (passwords, keys, proprietary algorithms).
- Validate: Run, lint, and test. Copilot accelerates typing; you own the quality.
- Feedback loop: Good names, clear structure, and small steps improve the next suggestion.

# RETROSPECTIVE



This module showed practical ways to use GitHub Copilot completions effectively. We saw how ghost text works and how intent signals—comments, docstrings, imports, and function signatures—improve results. Techniques like scaffolding, partial code hints, and test-driven scaffolding emphasized that small, clear prompts generate better completions.

Reviewing output for correctness, security, and style, while avoiding sensitive data in prompts.

# LAB 3 – CAGR



# COMPOUND ANNUAL GROWTH RATE

Compound Annual Growth Rate (CAGR) is a way to describe growth as if it happened steadily each year, even when real results bounced around. You give it a starting value, an ending value, and how many years passed; CAGR answers, "What single yearly growth rate would get me from start to finish if it repeated every year?" It turns a jagged journey into a smooth, easy-to-compare story.

Why it's useful: it lets you compare investments or business metrics across different time periods without being distracted by big ups and downs. For example, if \$1,000 becomes \$1,500 in three years, CAGR says that's like growing about 14–15% per year, even if one year was great and another was weak. Just remember, it hides volatility, ignores extra deposits or withdrawals, and assumes you reinvest gains—so use it as a summary, not the whole picture.

# COMPOUND ANNUAL GROWTH RATE

Step 1 – In cagr.py, provide the intent

```
# Calculate CAGR (Compound Annual Growth Rate)  
def cagr(begin, end, years):
```

Copilot likely fills:

```
return (end / begin) ** (1 / years) - 1
```

Step 2 – Refine with a second hint - make begin parameter the float type

Does anything else change

# MAIN AND TEST

Step 3 – Another function by describing the intent.

```
# Calculate CAGR as a percentage and return a string
def cagr_percent(begin, end, years):
```

Step 4 – Ask copilot to create a main function to test the program with these values. Test both functions.

```
cagr(1000, 1500, 3)
```

```
cagr(2000, 3000, 5)
```

```
cagr(1500, 1000, 2)
```

```
...
```

Step 5 – Display the results in a nice format. Of course, using Copilot.

Step 6 – Run the program

# Lab completed



"Develop a passion for learning."

# COPilot CHAT

## INTEGRATION



# INTRODUCTION



The presentation introduces GitHub Copilot Chat in VS Code as a conversational coding assistant. It expands beyond inline completions by supporting natural language queries, code edits, and even multi-step workflows through Ask, Edit, and Agent modes.

It also highlights practical tools such as checkpoints to track and roll back edits, action controls to apply or discard suggestions, and inline chat for quick changes. Features like slash commands, #mentions for context, and variable awareness show how Copilot integrates smoothly into the coding workflow.

# COPILOT CHAT



Learn how to use GitHub Copilot Chat in VS Code, contrasting the full Chat View with the lightweight Inline Chat (selection-tied, accept/discard, no history).

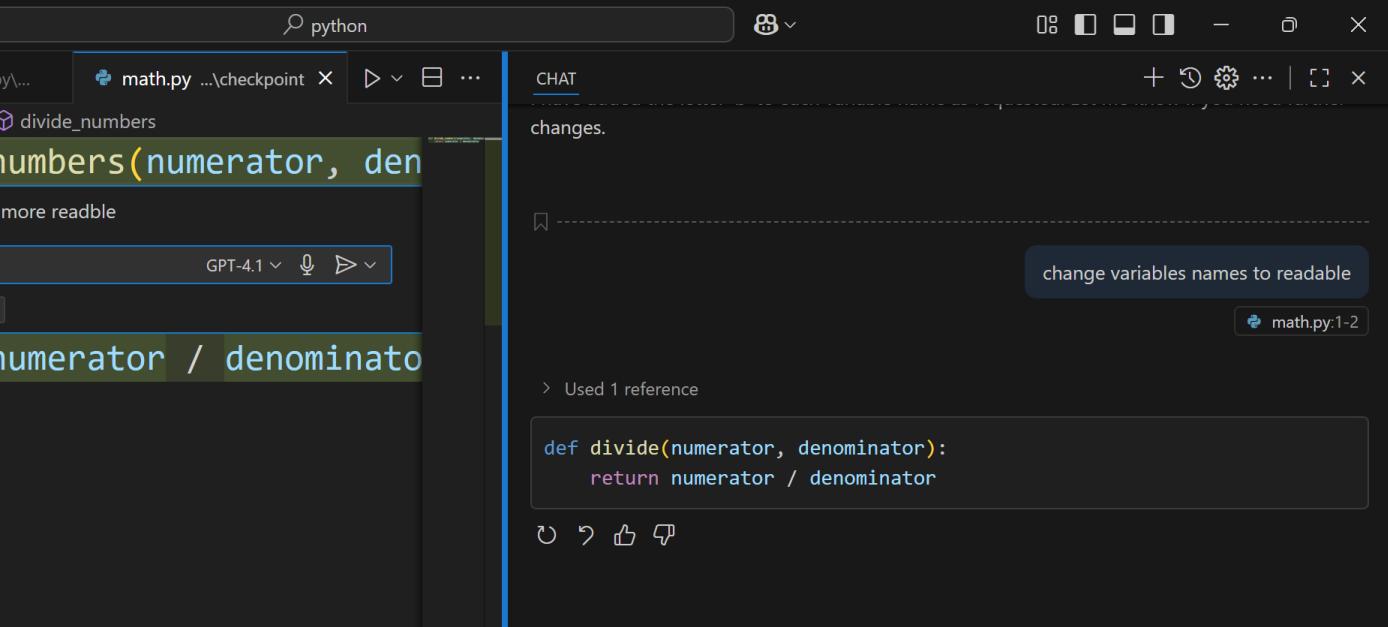
This module covers key controls—model choice, enable/disable completions, shortcuts, settings, diagnostics/logs—and two modes: Ask (Q&A) and Edit (apply changes).

# HIGHLIGHTING SOURCE

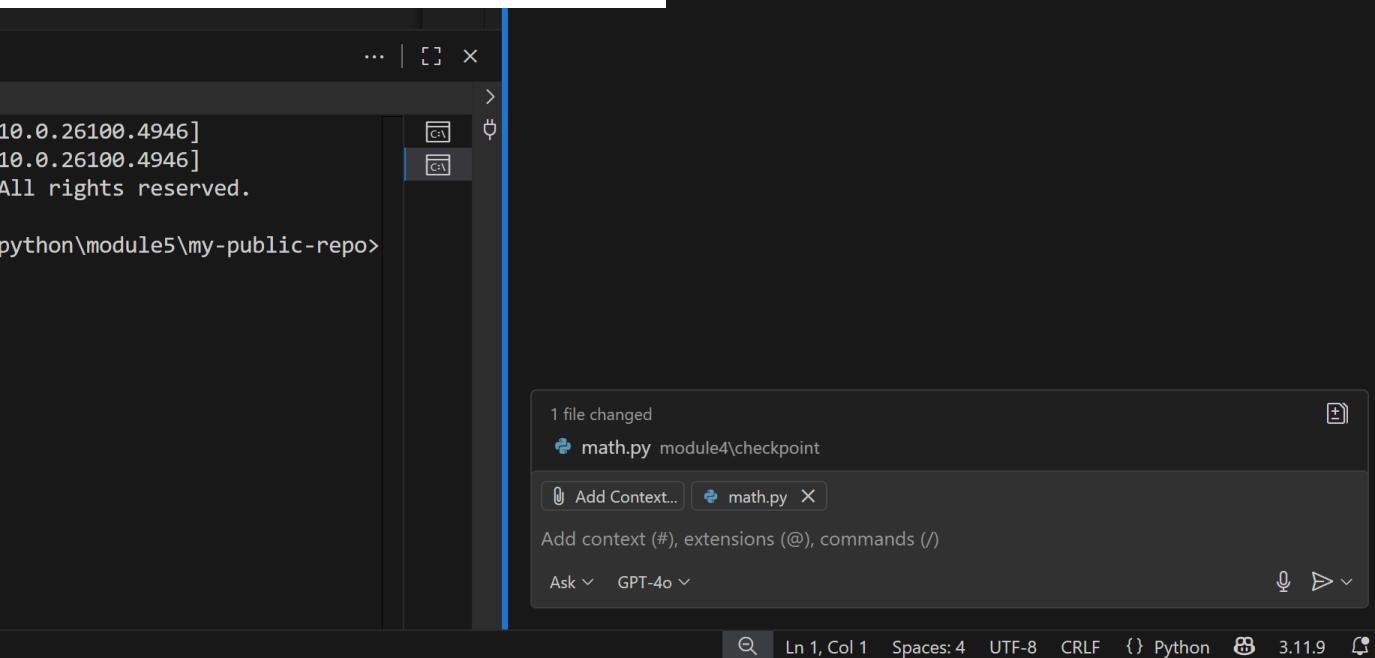


When you highlight a specific block of code, Copilot knows exactly what part of the file you want it to focus on. This narrows the context and reduces ambiguity, making its output more relevant. For example, highlighting a function and asking Copilot to “optimize for readability” ensures that only that function is considered, rather than the entire file.

In addition, safer edits and clear boundaries and leads to faster review and refined control.



## COPILOT CHAT SIDE PANEL



# COPILOT CHAT

Open the Chat panel:

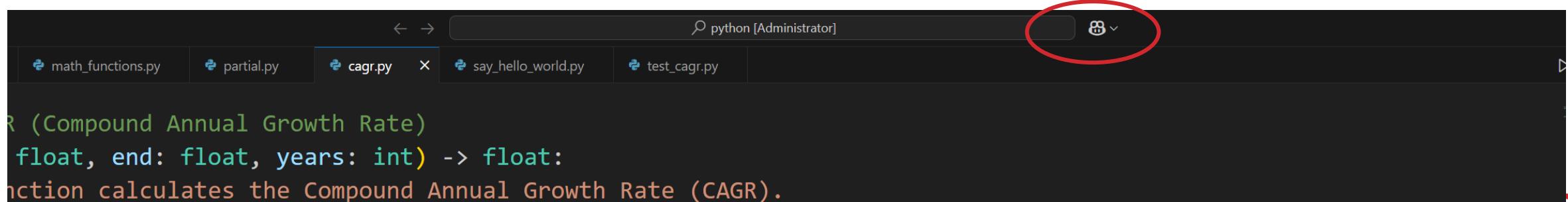
Once installed, you'll see a Copilot icon (a swirl-like symbol) in the Activity Bar on the left sidebar.

Click it to open the Chat View.

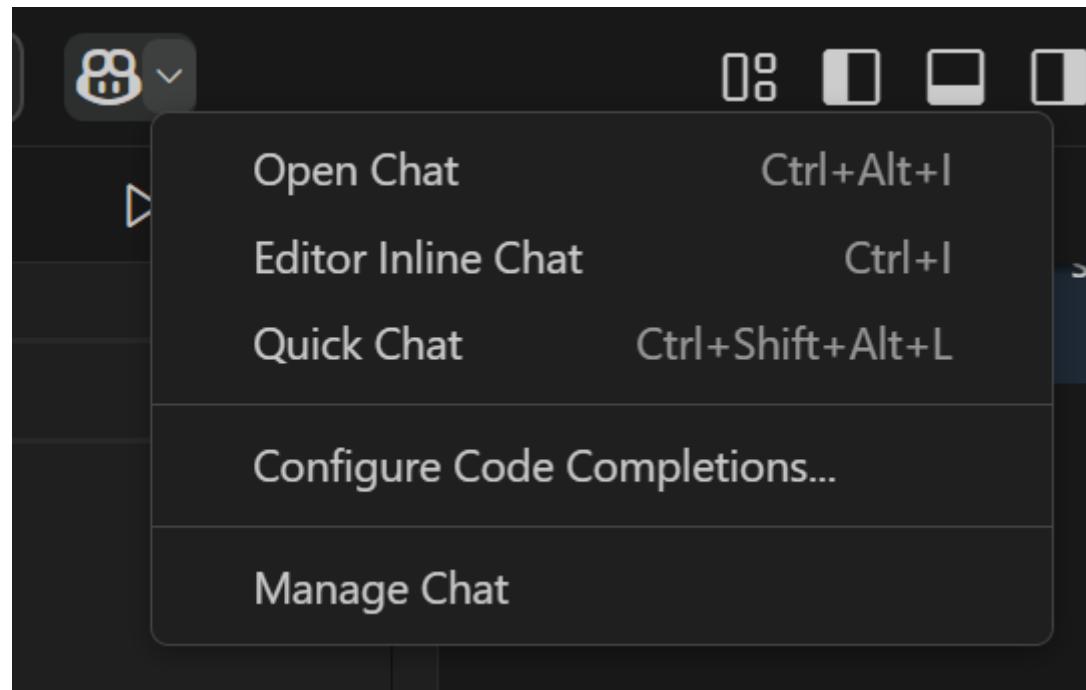
Alternatively, you can open chat directly with:

Ctrl+Alt+I (Windows/Linux)

⌃⌘I (macOS).



# COPilot CHAT MENU



Open Chat (Chat View) is the full Copilot panel for longer conversations. You can open it from the Copilot menu or with Ctrl + Shift + I (Windows/Linux) or ⌘I (macOS).

Editor Inline Chat brings Copilot directly into the editor or terminal at your cursor. Triggered by Ctrl + I (Windows/Linux) or ⌘I (macOS), it's best for quick, context-specific tasks.

# MANAGE CHAT

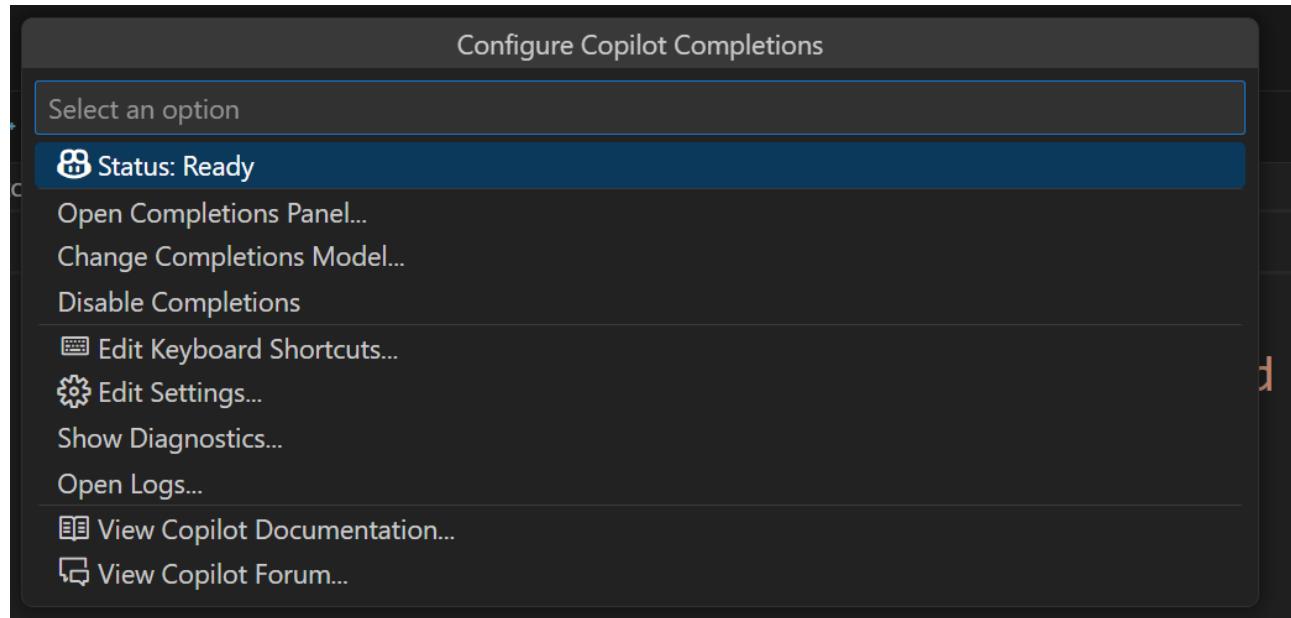
In GitHub.com, opens the Copilot settings (Features) for the current GitHub account.

The screenshot shows the GitHub Settings interface for the user 'DonisMarshall2'. The main header says 'Chat with Copilot'. On the left, there's a sidebar with various account management options like Public profile, Account, Appearance, Accessibility, Notifications, Billing and licensing, Emails, Password and authentication, Sessions, SSH and GPG keys, Organizations, Enterprises, and Moderation. The 'Copilot' section is highlighted. The main content area is titled 'GitHub Copilot' and displays information about the Copilot Pro subscription status, usage statistics (Premium requests at 0.0%), and feature status (Editor preview features and Copilot in GitHub.com both enabled). Buttons for 'Copilot in your IDE', 'Copilot in the CLI', 'Chat in GitHub Mobile', and 'More features' are also present.

## MANAGE CHAT

Quick Chat is a mini Chat side panel, accessed with Ctrl + Shift + Alt + L (Windows/Linux) or ⌘+⌥+⌘+L (macOS). It opens a small box for single questions or snippets. Unlike Open Chat, it doesn't keep conversation history—ideal for quick lookups or short code suggestions.

# CONFIGURE COPILOT COMPLETIONS MENU



The Configure Copilot Completion from the Copilot Chat menu is focused on how inline suggestions (ghost text) behave while you type, including the current status.

- Status – Displays whether Copilot completions are currently active. It shows if completions are enabled globally.
- Open Completions Panel – Opens a dedicated panel where you can see multiple alternative .

## CONFIGURE COPILOT COMPLETIONS MENU - 2

- Disable Completions – Provides a quick way to temporarily turn off inline completions. Useful if you don't want Copilot generating ghost text while you type.
- Edit Keyboard Shortcuts – Opens the Keyboard Shortcuts editor with Copilot commands pre-filtered, so you can assign or change hotkeys for things like triggering completions or cycling through suggestions.
- Edit Settings – Opens the Copilot-related settings (in the GUI or `settings.json`) where you can control behavior such as enabling completions by language, automatic vs. manual triggers, and inline vs. panel display.

## CONFIGURE COPILOT COMPLETIONS MENU - 3

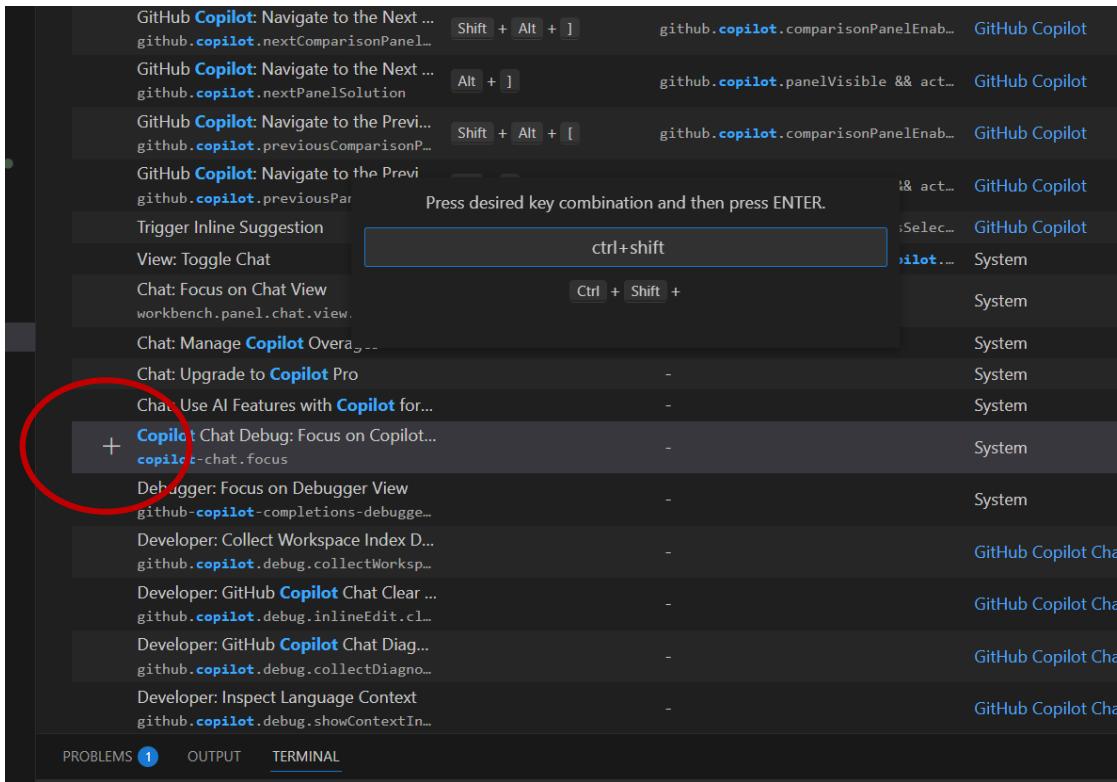
- Show Diagnostics (static) – Displays diagnostic information about Copilot's state and connectivity. This can help confirm whether the extension is connected properly and if requests are being sent and received.
- Open Logs (real time) – Opens the Output panel in VS Code filtered to GitHub Copilot logs. These logs capture startup messages, network errors, or authentication issues and are helpful for troubleshooting.

# EXAMPLE - EDIT KEYBOARD SHORTCUTS

The Edit Keyboard Shortcuts option in the Copilot Chat menu opens VS Code's Keyboard Shortcuts editor with Copilot commands pre-filtered. From there, you can assign, change, or remove shortcuts

Command	Keybinding	When	Source
GitHub Copilot: Accept Comparison P... github.copilot.acceptCursorCompari...	Ctrl + Shift + /	github.copilot.comparisonPanelEnab...	GitHub Copilot
GitHub Copilot: Accept Panel Suggest... github.copilot.acceptCursorPanelSo...	Ctrl + /	github.copilot.panelVisible && act...	GitHub Copilot
GitHub Copilot: Debug Last Terminal ... github.copilot.chat.rerunWithCopil...	Ctrl + Alt + .	github.copilot-chat.activated && t...	GitHub Copilot Cha...
GitHub Copilot: Navigate to the Next ... github.copilot.nextComparisonPanel...	Shift + Alt + ]	github.copilot.comparisonPanelEnab...	GitHub Copilot
GitHub Copilot: Navigate to the Next ... github.copilot.nextPanelSolution	Alt + ]	github.copilot.panelVisible && act...	GitHub Copilot
GitHub Copilot: Navigate to the Previous Comparison Panel github.copilot.previousComparisonP...	Shift + Alt + [	github.copilot.comparisonPanelEnab...	GitHub Copilot
GitHub Copilot: Navigate to the Previous Panel Solution github.copilot.previousPanelSoluti...	Alt + [	github.copilot.panelVisible && act...	GitHub Copilot

# COMMON CUSTOM KEY BINDINGS



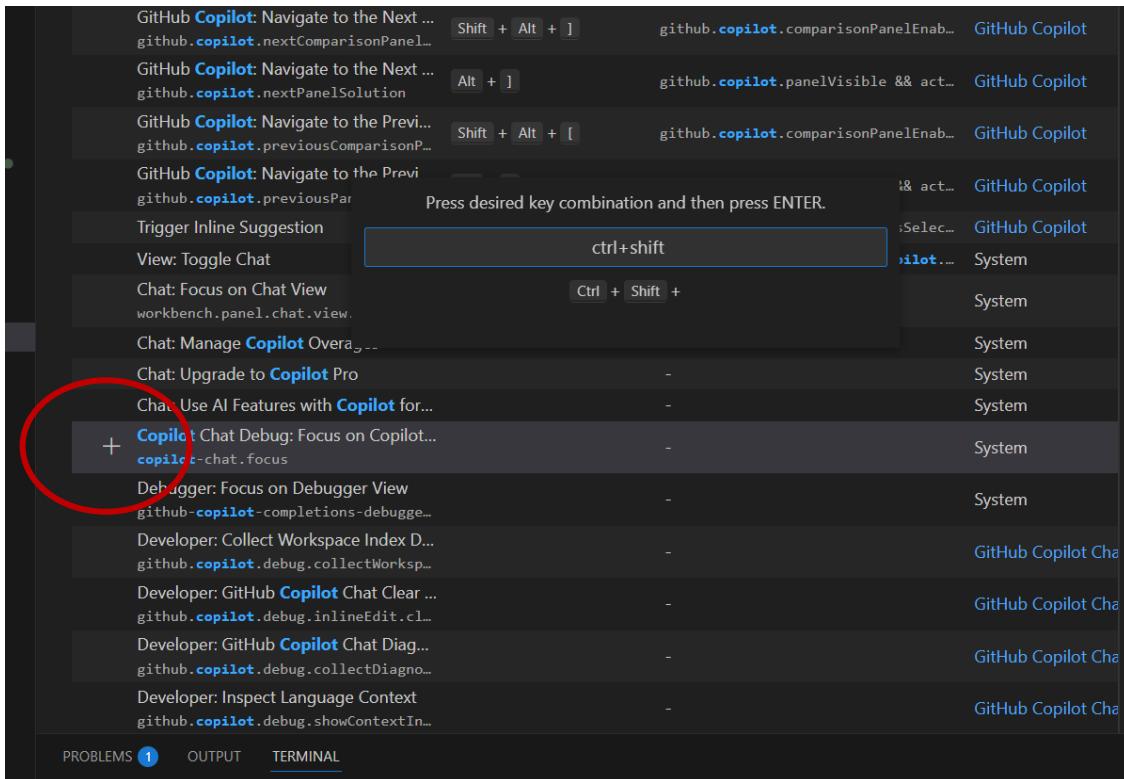
GitHub Copilot: Enable/Disable Completions – Toggle inline suggestions globally.

GitHub Copilot: Explain with Copilot – Ask Copilot to explain highlighted code.

GitHub Copilot: Accept Suggestion – Accept the current inline completion.

GitHub Copilot: Trigger Inline Suggestion – Manually request a suggestion on demand.

# COMMON CUSTOM KEY BINDINGS - 2



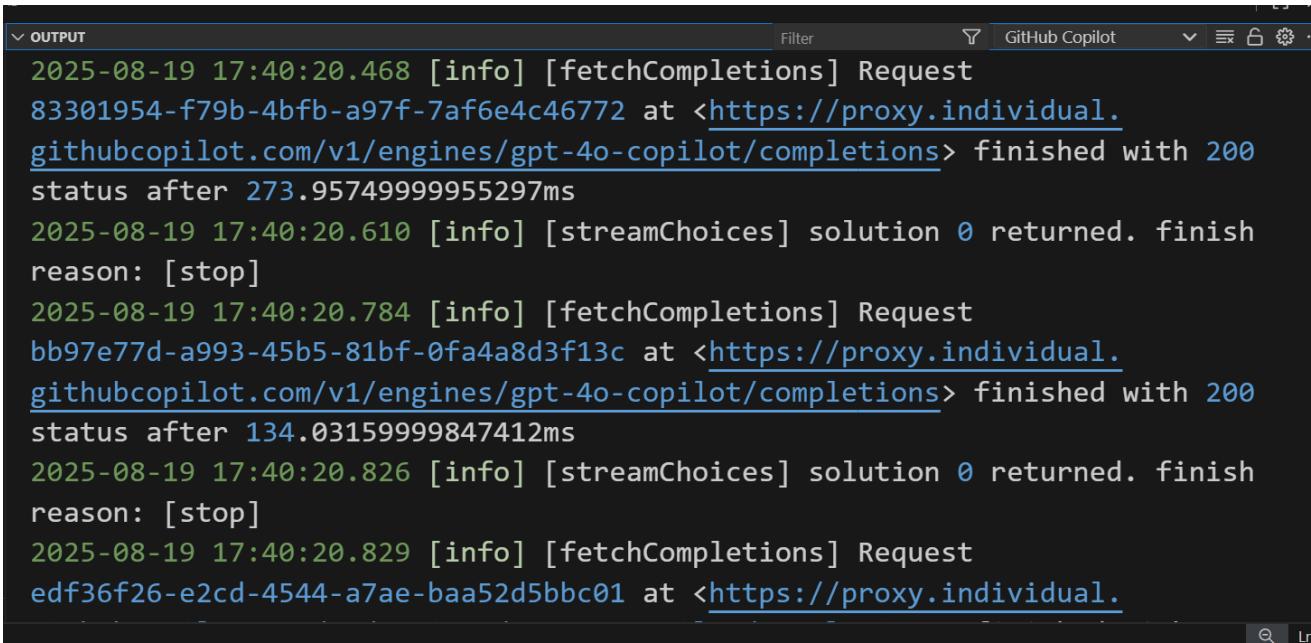
GitHub Copilot: Generate Tests – Ask Copilot to generate unit tests for selected code.

GitHub Copilot: Insert at Cursor / Apply to Selection – Choose how Copilot applies generated code.

# EXAMPLE - OPEN LOG

The Copilot log in VS Code is a diagnostic record of what the extension is doing behind the scenes. It captures events like when Copilot starts up, sends or receives a completion request, or encounters an error. You can view it by opening the Output panel and selecting GitHub Copilot from the dropdown list.

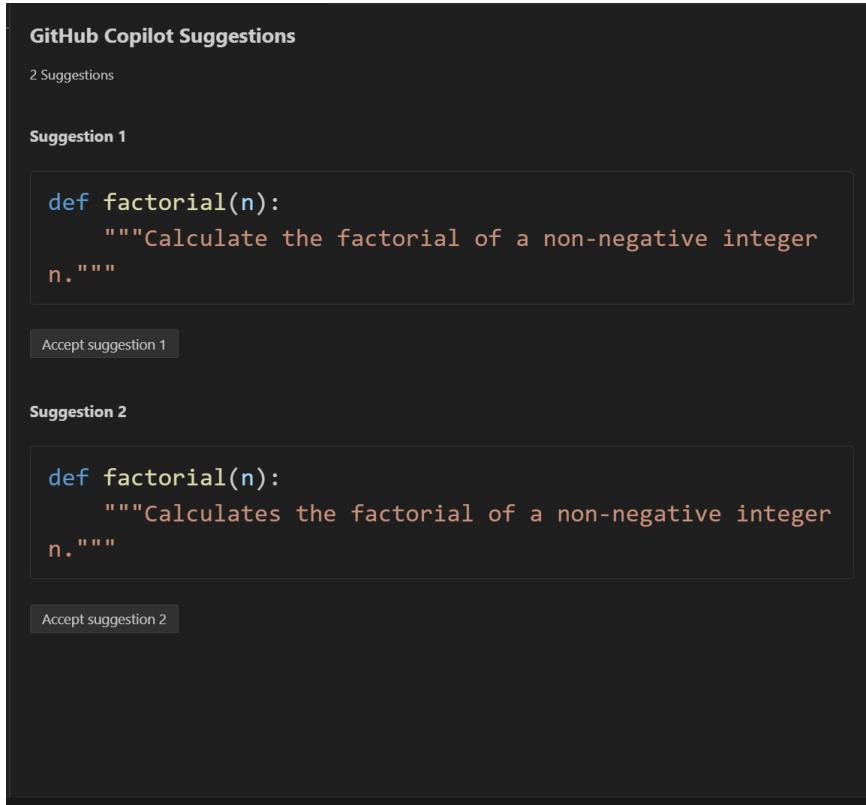
These logs are mainly for troubleshooting. If Copilot isn't generating suggestions or fails to connect, the log can reveal issues such as authentication errors, network timeouts, or extension crashes.



The screenshot shows the VS Code Output panel with the title bar "OUTPUT" and a dropdown menu showing "GitHub Copilot". The panel displays a log of events:

```
2025-08-19 17:40:20.468 [info] [fetchCompletions] Request  
83301954-f79b-4bfb-a97f-7af6e4c46772 at <https://proxy.individual.  
githubcopilot.com/v1/engines/gpt-4o-copilot/completions> finished with 200  
status after 273.9574999955297ms  
2025-08-19 17:40:20.610 [info] [streamChoices] solution 0 returned. finish  
reason: [stop]  
2025-08-19 17:40:20.784 [info] [fetchCompletions] Request  
bb97e77d-a993-45b5-81bf-0fa4a8d3f13c at <https://proxy.individual.  
githubcopilot.com/v1/engines/gpt-4o-copilot/completions> finished with 200  
status after 134.03159999847412ms  
2025-08-19 17:40:20.826 [info] [streamChoices] solution 0 returned. finish  
reason: [stop]  
2025-08-19 17:40:20.829 [info] [fetchCompletions] Request  
edf36f26-e2cd-4544-a7ae-baa52d5bbc01 at <https://proxy.individual.
```

# EXAMPLE - OPEN COMPLETIONS PANEL...



Select the header for the Factorial function and choose the Open Completions Panel.

# CHAT MODES

Vscode supports multiple built-in chat modes tailored to different tasks. Here are the main modes:

## Ask Mode (Default)

Ideal for asking questions or explanations about code, technology, best practices, or debugging.  
Responses may include code snippets or concepts; you apply changes manually

## Edit Mode

Suited for applying code changes across multiple files.  
You supply context, and Copilot will generate edits. You can review and accept changes

## Agent Mode

For autonomous, multi-step workflows. Copilot determines necessary tasks, executes commands  
(e.g., build, tests), applies and iterates until completion

## ASK MODE

In GitHub Copilot Chat, the default interaction mode is Ask mode.

That means when you open Copilot Chat in VS Code and type a question or prompt, you're by default in a conversational Q&A flow — asking Copilot about code, documentation, debugging, or general programming topics. Other modes, like /explain, /tests, or /fix, are invoked explicitly with slash commands, but Ask mode doesn't require a prefix — you just type naturally.

## EDIT MODE

In GitHub Copilot Chat, Edit mode lets you directly modify code in your editor using natural language instructions. Instead of just asking Copilot questions (Ask mode), you highlight a block of code, then tell Copilot what you want changed. Copilot will generate a replacement or modification, and you can preview and accept it.

This mode uses NLP (natural language processing) to interpret your intent. For example, you might say “optimize this function for readability” or “add error handling for division by zero,” and Copilot edits the selected code accordingly.

Syntax: /edit *chat instruction*

# ASK VERSUS EDIT MODE

In the current version of GitHub Copilot Chat, there is little distinction between Ask mode and the /edit command. When you type an instruction in Ask mode, Copilot automatically interprets your intent based on context. If you highlight code, it treats your request as an edit; if no code is selected, it generates explanations, examples, or new snippets. This makes Ask mode flexible and able to handle both conversational queries and direct code edits without requiring a special command.

The /edit command still exists, but its function has been streamlined into the same workflow as Ask mode. Using /edit explicitly signals that you want an edit, yet the outcome is effectively identical to giving an instruction in Ask mode. Both methods produce inline previews with options to accept or discard, making the experience consistent and simpler. In practice, you can rely on Ask mode alone for most tasks, since it now covers the same ground as /edit.

# CHECKPOINTS

In the Copilot Chat side panel, when you use the /add or /edit commands, the history of your interactions is preserved right in the chat thread. Each request and Copilot's corresponding suggestion are shown in sequence, making it easy to scroll back and review prior edits or additions without losing context. This gives you a running log of how your code evolved through chat-based instructions.

At the same time, checkpoints are placed directly in your code editor at small dotted line markers. These indicate where Copilot made a change. If you hover over a dotted line, you'll see the checkpoint, which allows you to quickly roll back to the earlier version of your code. This provides a convenient safety net, ensuring that edits suggested through the chat panel are reversible.

# COMMANDS

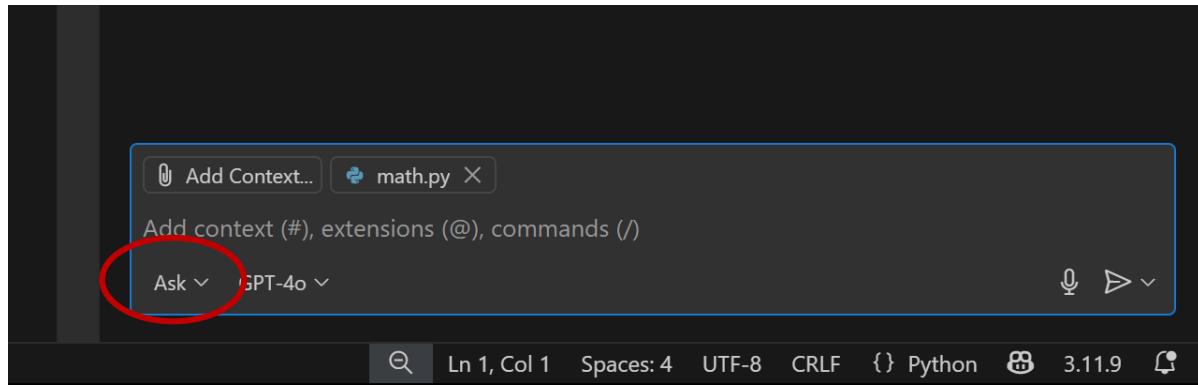


You can use the slash commands to steer Copilot into specific behaviors.

## Common Copilot Chat Commands

- **/ask:** Default mode. Ask a question about code, concepts, or documentation.
- **/edit:** Edit selected code directly with your instruction.
- **/fix:** Suggest fixes for errors or issues in the selected code.
- **/explain:** Explain what the selected code does in plain language.
- **/tests:** Generate unit tests for the selected code.

# ASK MODE



At the bottom of the chat window, there is the command pane, where commands are entered. You can change the mode here also.

- Ask
- Edit
- Agent (discussed later)

As shown, the default is Ask, which is more advisory or view mode.

# CHECKPOINTS

Here we have applied two Add commands to a divide function. Both are shown in the history.

The checkpoints are also displayed as dotted lines. However, over the lower checkpoints enables the Restore Checkpoint button.

```
change the variable names to something more readable  
math.py
```

```
> Used 1 reference
```

```
def divide(numerator, denominator):  
    return numerator / denominator
```

```
Restore Checkpoint
```

```
add exception handling  
math.py:1-2
```

```
> Used 1 reference
```

```
def divide(numerator, denominator):  
    try:  
        return numerator / denominator  
    except ZeroDivisionError:  
        return "Error: Division by zero is not allowed."  
    except TypeError:  
        return "Error: Both numerator and denominator must be numbers."
```

```
⌚ ⚡ ⏪ ⏴
```

```
Add Context... math.py:1-2
```

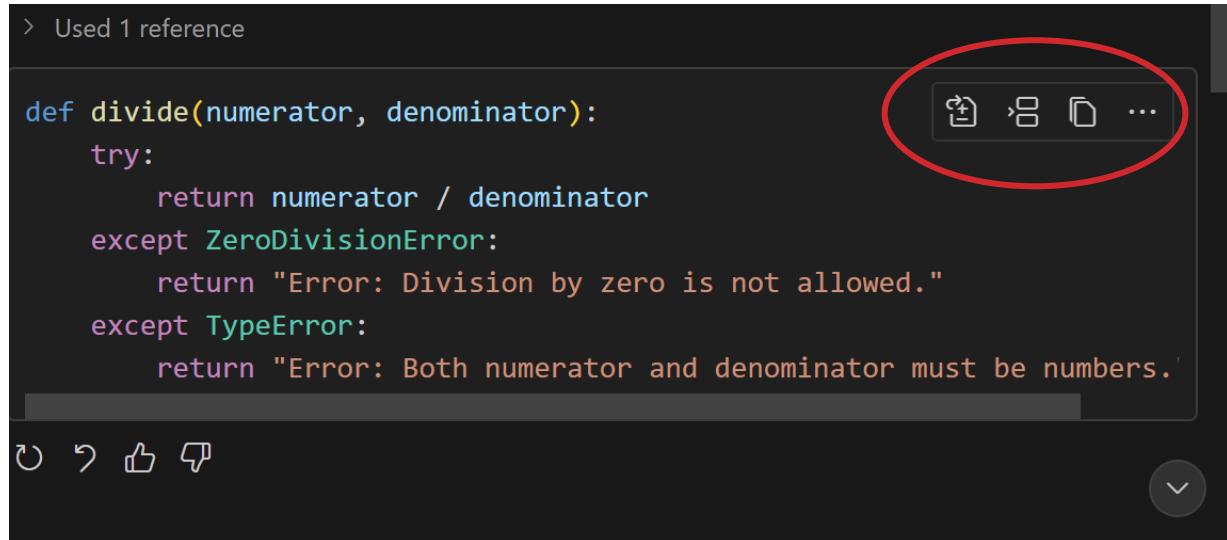
```
Add context (#), extensions (@), commands (/)
```

```
Ask GPT-4o
```

```
134
```

# ACTION BAR

When you run an /ask or /edit command in Copilot Chat, Copilot's suggestion appears in the chat side panel. Hovering over that suggestion reveals four action buttons that let you decide how to use the generated code.



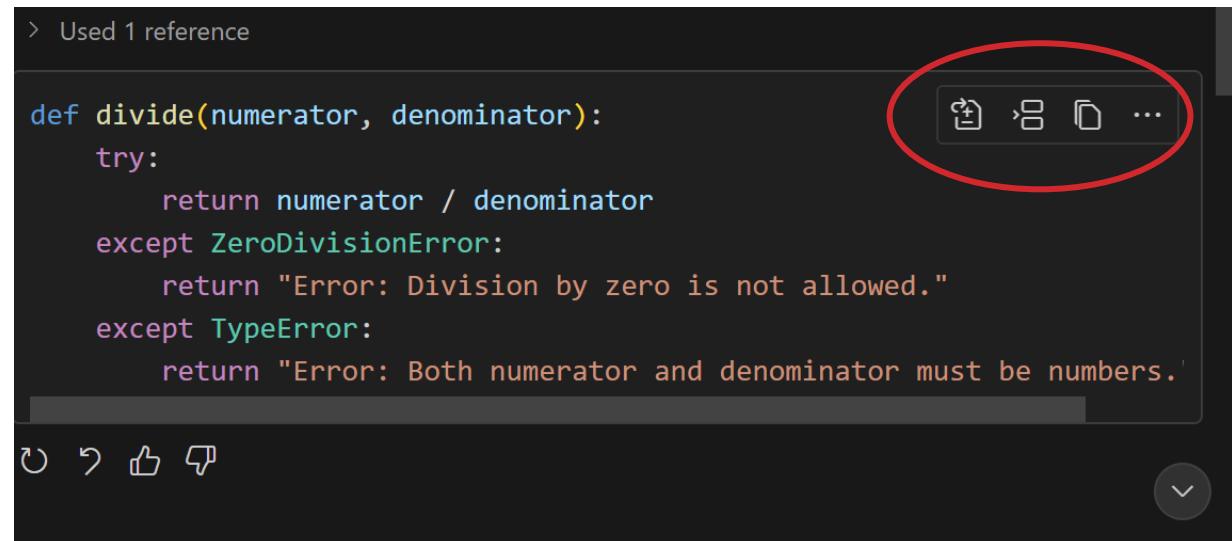
# ACTIONS

Apply To: Applies the suggestion directly to the code you had selected in the editor.

Insert At: Lets you insert the suggestion at your current cursor location in the open file.

Copy: Copies the suggestion to your clipboard.

Short menu: Insert into Terminal and Insert into New File.



The screenshot shows a code editor window with the following Python code:

```
> Used 1 reference

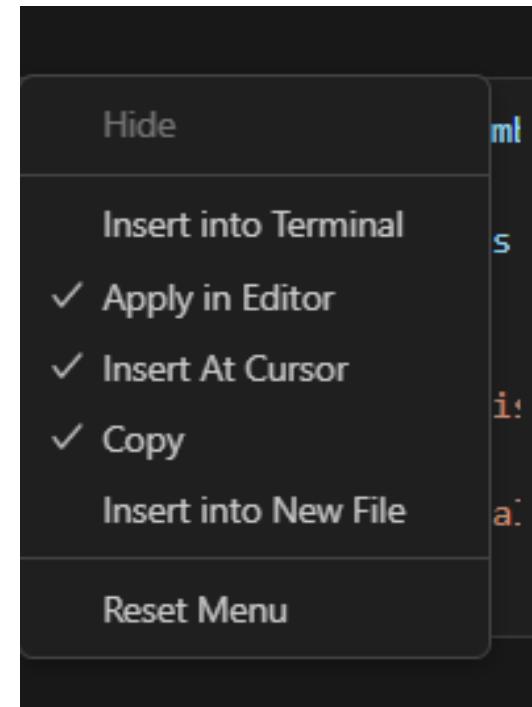
def divide(numerator, denominator):
    try:
        return numerator / denominator
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."
    except TypeError:
        return "Error: Both numerator and denominator must be numbers."
```

A context menu is open at the end of the code block, with the ellipsis ('...') option highlighted by a red oval. The menu also includes options for 'Used 1 reference', 'Copy', 'Insert into Terminal', and 'Insert into New File'.

## ACTIONS - 2

Insert into Terminal: add display to terminal window.

Insert into New File: create a new untitled file and insert

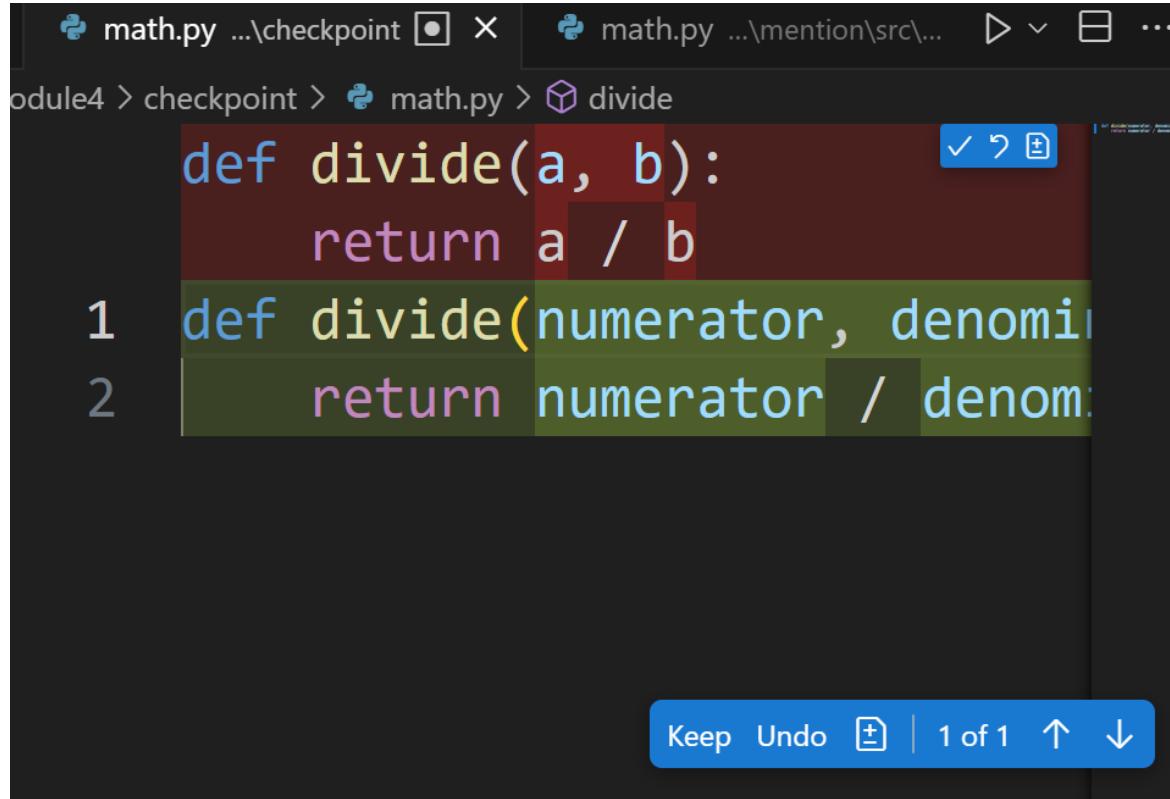


## DIFF VIEW

If you decide to apply the suggestion, then the suggestion is presented within a diff view, with the suggestion shown below the original. Here are the allowable actions:

- Keep the suggestion
- Reject the suggestion
- Toggle editor marks
- Scroll through suggestions

# KEEP CONTROL



A screenshot of a code editor interface. At the top, there are two tabs: "math.py ...\\checkpoint" and "math.py ...\\mention\\src\\...". Below the tabs, the file path "module4 > checkpoint > math.py > divide" is shown. The code editor displays two definitions of the "divide" function:

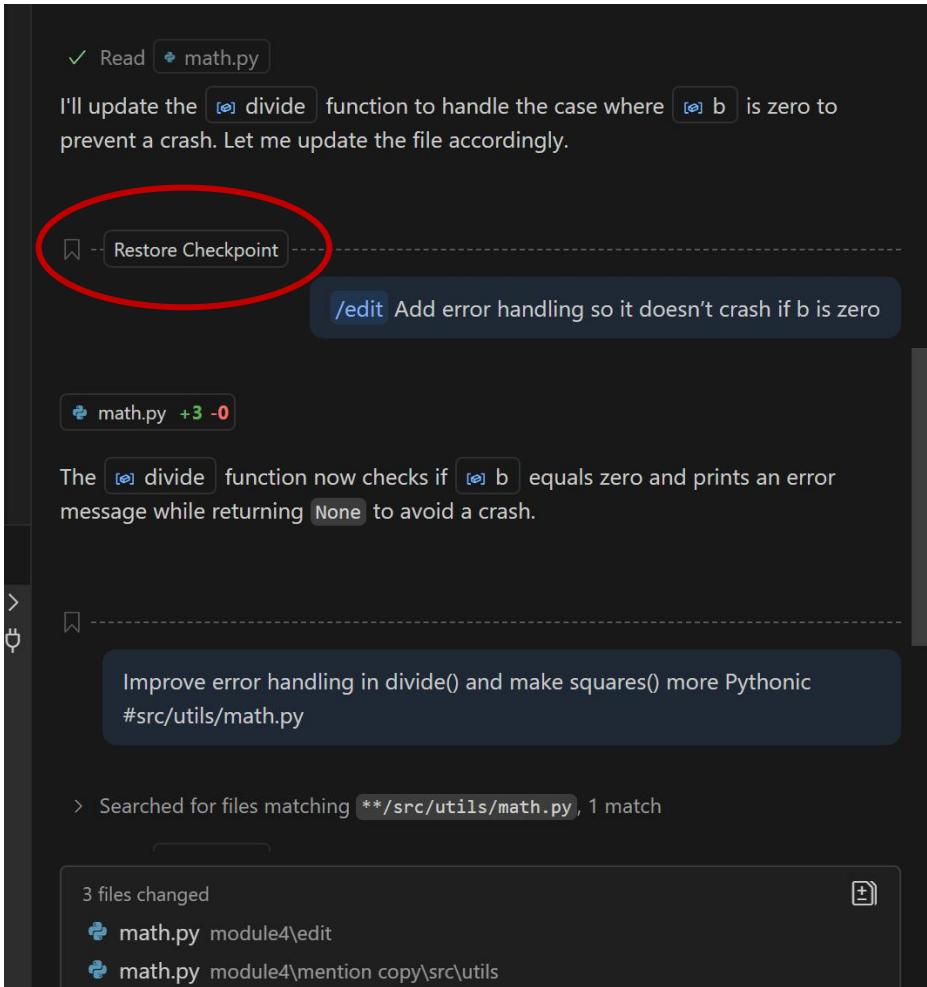
```
def divide(a, b):
    return a / b
1 def divide(numerator, denominator):
2     return numerator / denominator
```

The second definition is highlighted with a green background. A blue callout box labeled "Keep" is positioned over the first definition. In the bottom right corner of the code area, there is a small blue button with a checkmark icon. At the bottom of the editor window, there is a toolbar with buttons for "Keep", "Undo", and "Redo", followed by the text "1 of 1" and navigation arrows.

The Keep control allows:

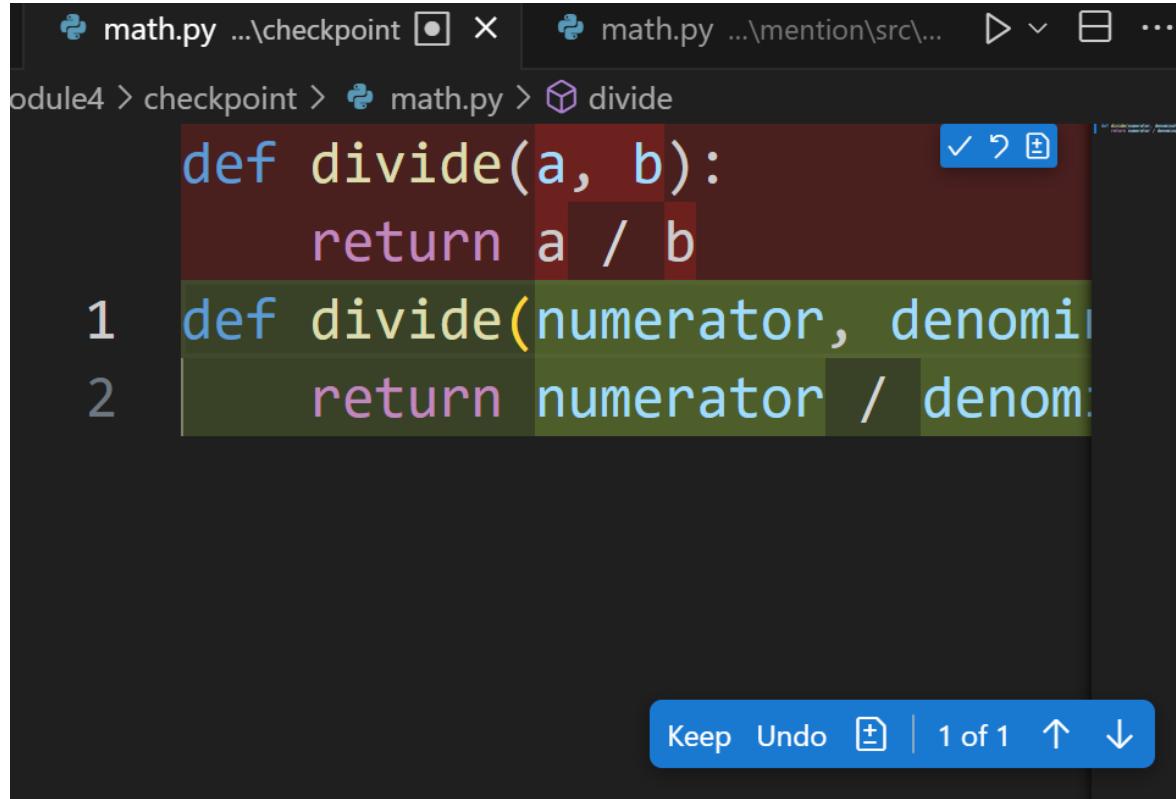
- Keep a suggestion
- Undo (reject) a suggestion
- Arrow through available suggestions

# CHAT PANE



In the Copilot Chat pane, checkpoints will also appear when hovering over the related action.

# EDIT MODE



The screenshot shows the GitHub Diff view in Edit mode. The top bar displays file paths: 'math.py ...\\checkpoint' and 'math.py ...\\mention\\src\\...'. Below the bar, the commit message 'module4 > checkpoint > math.py > divide' is shown. The main area contains two code snippets:

```
def divide(a, b):
    return a / b
1 def divide(numerator, denominator):
2     return numerator / denominator
```

The first snippet is in a dark red background, indicating it's the current file being edited. The second snippet is in a green background, indicating it's the target file. A blue status bar at the bottom right shows 'Keep Undo ⌘ | 1 of 1 ↑ ↓'.

With the edit mode, you skip the staging (i.e., Add mode) and the suggestion is immediately presented in the Diff view.

You can then accept or reject the suggestion with the Diff Editor control.

# INLINE EDITOR

A screenshot of a code editor interface. At the top, there's a navigation bar with icons for back, forward, search, and user profile. Below it is a tab bar showing two files: "math.py ...\\checkpoint" and "math.py ...\\mention\\src\\...". The main area displays a Python function:

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

Below the code, there's a button labeled "Ask or edit in context" and a dropdown menu showing "GPT-4.1". To the right, a sidebar shows a message from GPT-4.1: "change the variable name". Below that, it says "Used 1 reference" and shows a file icon with "+2 -2". The sidebar also includes a "Used 1 reference" section and a "math.py +2 -2" entry.

A screenshot of a code editor interface showing a terminal window. The title bar of the terminal window reads "Microsoft Windows [Version 10.0.26100.4946] C:\>". The terminal window is empty, showing only the command prompt. To the right of the terminal, there's a sidebar with a message: "I have updated the variable names to be more descriptive. Let me know if you need further modifications." Below that are upvote, downvote, like, and dislike buttons. The sidebar also shows a "1 file changed" message with "math.py module4\\checkpoint" and "Add Context..." buttons. At the bottom, there's an "Edit" dropdown set to "GPT-4o".

# INLINE CHAT WINDOW

The inline overlay is a lightweight tool meant for quick edits or explanations directly in your code. It has no conversation history — once closed, the exchange is gone — and doesn't support context tools like #file or #codebase. It generates a simple preview with Accept/Discard, tied only to the code you highlighted. Checkpoints are limited here: you can undo changes using VS Code's native undo, but you don't get the richer checkpoint markers or history that appear in the side panel.

- Much simpler: just type instructions in a small inline box.
- No conversation history — once you close it, it's gone.
- Generates a preview with Accept/Discard only, tied to the code you highlighted.
- No extra context tools (#codebase, #file, etc.) in this view.

# INLINE CONTROLS

```
1 def divide(a, b):  
    change the name of the variables  
    to be more readable  
2     return a / b
```

The first button allows you to change the LLM.

After entering a command, accept the command with the Enter key or the Send and Dispatch button.

# CHECKPOINT EXAMPLE

Highlight the divide function, press Ctrl+I (Windows/Linux) or Cmd+I (macOS), and type:

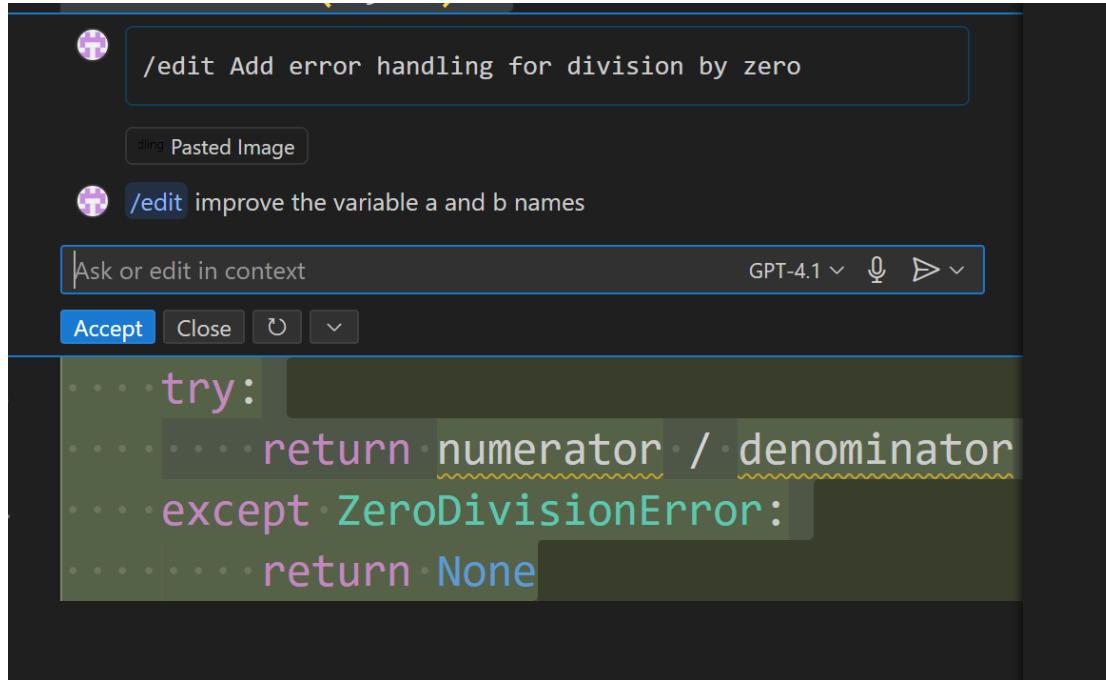
Add error handling for division by zero

Copilot Suggestion:

```
def divide(a, b):  
    if b == 0:  
        return "Error: Division by zero"  
    return a / b
```

A checkpoint is automatically created, storing the original version before applying this change.

# CHECKPOINT EXAMPLE



The screenshot shows a code editor window with a dark theme. At the top, there's a GPT-4.1 interface with a blue header bar containing a purple icon, the text '/edit Add error handling for division by zero', and a 'Pasted Image' button. Below this is another purple icon followed by the text '/edit improve the variable a and b names'. A search bar says 'Ask or edit in context' and a status bar says 'GPT-4.1'. At the bottom of the interface are 'Accept', 'Close', and other buttons. The main code area shows Python code:

```
try:  
    return numerator / denominator  
except ZeroDivisionError:  
    return None
```

You'll see a small "checkpoint" marker above the edited section.

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

Ask or edit in context GPT-4.1 ▾ ⟲ ⟳

change the variable name

Used 1 reference

math.py +2 -2

## OTHER FEATURES

```
C:\>
```

I have updated the variable names to be more descriptive. Let me know if you need further modifications.

1 file changed  
math.py module4\checkpoint

Add Context... math.py X

Add context (#), extensions (@), commands

Edit ▾ GPT-4o ▾

# VARIABLES

Copilot fully understands variables you've declared in the codebase.

Example:

```
total = price * quantity
```

If you then write a comment like

```
# calculate tax on total
```

Copilot knows total is a variable and will suggest code like:

```
tax = total * 0.08
```

Copilot's completions are context-aware. It looks at variable names, types, and surrounding code when generating suggestions.

# VARIABLES - 2

Copilot can sometimes get confused if you have several variables with similar names (like user, username, userData). To disambiguate in a Copilot prompt, you give it extra clues.

## 1. Mention the Type or Purpose

If you have both user (an object) and username (a string), you can guide Copilot by saying:

Instead of:

"Use the user variable"

Say:

"Use the username string variable, not the user object"

# VARIABLES - 3

## 3. Use Comments Near the Code

Copilot pays attention to comments. Right above where you're working, add:

```
# Use the username variable, not the user object  
print(username)
```

Copilot will now autocomplete correctly because you've clarified in plain English.

# VARIABLES - 4

## 4. Use Explicit Prompt Language

Instead of just saying “fix this,” be very specific:

Good: “Loop through customer\_list (the array of names), not customer (the object).”

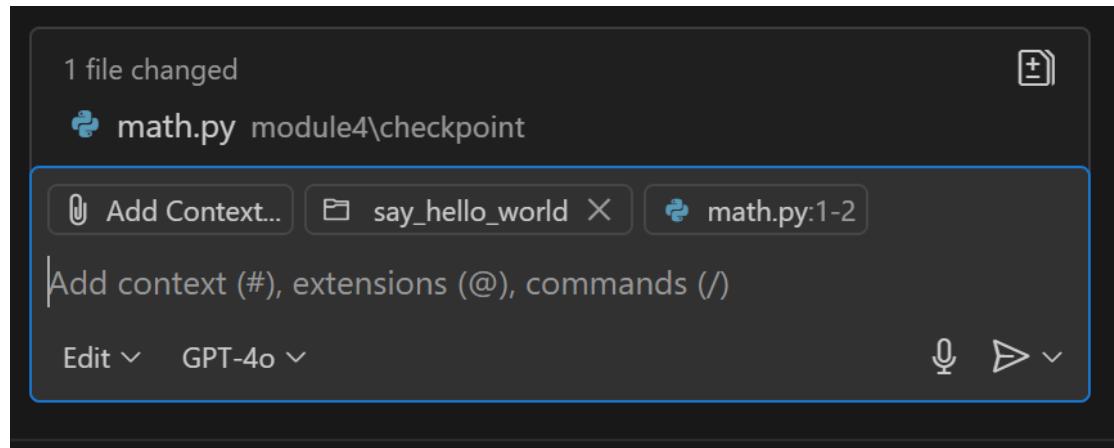
Bad: “Loop through customers.” (too vague)

# #MENTION



The “# mentions” method lets you manually give Copilot Chat extra context by referencing files, your whole workspace, or external resources directly in your prompt. This is useful if you don’t see the Add Context button in your version of VS Code

# CONTEXT BUTTON



The Add Context button, which is a paperclip, is provided to add items to the context, such as files.

It will open a dialog listed resources that can be selected.

## # MENTION

The “# mentions” method lets you manually give Copilot Chat extra context by referencing files, your whole workspace, or external resources directly in your prompt. This is useful if you don’t see the Add Context button in your version of VS Code.

- Add # followed by the context you want:
- #codebase: makes Copilot aware of the entire workspace/project.
- #file:<*filename*> points to a specific file, e.g. #app.js.
- #file:styles.css: adds supporting files like CSS or config files.

# MENTION EXAMPLE

Here's a short example of using file path mentions in Copilot Chat: Here is the project structure:

```
project/  
  src/  
    utils/  
      math.py  
  styles/  
    main.css
```

Prompt in Copilot Chat:

Explain how the functions are organized #file:src/utils/math.py.

Suggest improvements to button styling #file:styles/main.css.

Result:

Copilot looks at math.py to describe your utility functions. It also uses main.css to suggest design tweaks.

# #SELECTION

```
def process_data(records):
    cleaned = []
    for r in records:
        if r is not None and r != "" and r.strip() != "":
            cleaned.append(r.strip().lower())
    # ... more logic for filtering, transforming, exporting ...
    return cleaned
```

You can focus suggestions on selected code with `#selection`, such as a portion of a function.

For example:

```
#selection optimize
```

# #SELECTION #STYLE EXAMPLE

```
<!DOCTYPE html>

<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Button Example</title>
  <!-- Later we will replace style.css with new-styles.css -->
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <button class="primary-btn">Submit</button>
  <button class="secondary-btn">Cancel</button>
</body>
</html>
```

```
/* Base button template */

button {
  font-family: Arial, sans-serif;
  font-size: 14px;
  padding: 10px 20px;
  margin: 5px;
  cursor: pointer;
}
```

# NEW-STYLES.CSS

```
/* Styles for the primary button */
.primary-btn {
    background-color: blue;
    color: white;
    border: none;
    border-radius: 8px;
    padding: 10px 20px;
    font-size: 16px;
    cursor: pointer;
}

/* Styles for the secondary button */
.secondary-btn {
    background-color: gray;
    color: black;
    border: none;
    border-radius: 0;
    padding: 10px 20px;
    font-size: 16px;
    cursor: pointer;
}
```

#selection

#file:style.css

Create new-styles.css for index.html with styles for .primary-btn (blue, white text, rounded) and .secondary-btn (gray, black text, square).

# # CODEBASE

When you type #codebase in Copilot Chat, you're explicitly telling Copilot:

"Use my entire project as context for this request."

Normally, Copilot only looks at:

- The current file,
- Recently opened files, or
- What's in its local/workspace index.

By mentioning #codebase, you override that and attach the whole project folder (everything in your workspace index) as extra context.

# /EXPLAIN

```
def calculate_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
```

Explains the code you have selected or the file that's active in the editor. Great for onboarding to an unfamiliar area or validating logic before changes.

# /FIX

```
def divide_numbers(a, b):  
    return a / b  
  
print(divide_numbers(10,  
0))
```

Analyzes the selected code and proposes a patch for errors or suspicious patterns. You'll get a diff you can accept or tweak.

## /FIXTESTFAILURE

```
def parse_int_list(s: str) -> list[int]:  
    return [int(x) for x in s.split(",")]
```

Targets a failing test and suggests code changes to make it pass. Use when your test runner (or CI) is red and you want a focused remedy.

# /TESTS

```
def factorial(n: int) -> int:  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n-1)
```

Generates unit tests for your active file or selection, including typical and edge cases. You can specify frameworks or constraints.

# POP QUIZ: /TESTS COMMAND

Why is unittest imported in the test\_factorial.py file?



**10 MINUTES**

# /EXPLAIN

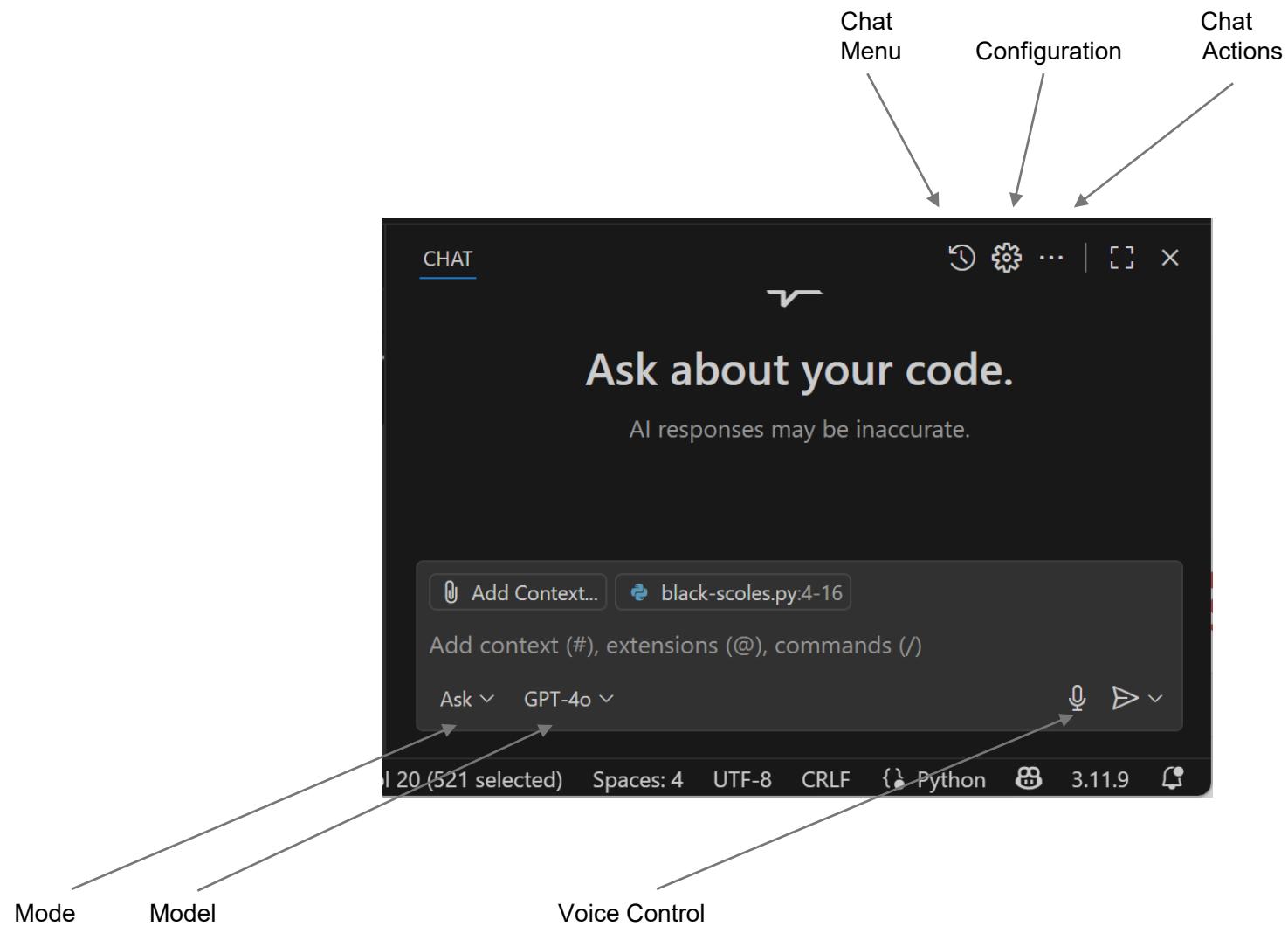
```
import math
from scipy.stats import norm

def black_scholes_call(S, K, T, r, sigma):
    try:
        if S <= 0 or K <= 0 or T <= 0 or sigma <= 0:
            raise ValueError("All inputs must be
positive.")

        d1 = (math.log(S / K) + (r + 0.5 * sigma**2)
* T) / (sigma * math.sqrt(T))
        d2 = d1 - sigma * math.sqrt(T)
```

```
call_price = S * norm.cdf(d1) - K *
math.exp(-r * T) * norm.cdf(d2)
return round(call_price, 4)
except Exception as e:
    print(f"Error calculating Black-Scholes
price: {e}")
return None
```

# CHAT INPUT BOX



# SHORTCUT KEYS

Action	Windows/Linux	macOS
Open Copilot Chat panel	<code>Ctrl + I</code>	<code>Cmd + I</code>
Open Inline Chat (editor overlay)	<code>Alt + Enter</code>	<code>Option + Enter</code>
Accept suggestion	<code>Tab</code>	<code>Tab</code>
Cycle inline suggestions	<code>Alt + [ / Alt + ]</code>	<code>Option + [ / ]</code>
Submit prompt in chat	<code>Enter</code>	<code>Enter</code>
Newline in chat prompt	<code>Shift + Enter</code>	<code>Shift + Enter</code>
Open chat with context	<code>Ctrl + Shift + \\"</code>	<code>Cmd + Shift + \\"</code>
Open chat from selection	<code>Right-click &gt; Copilot: Explain this code</code>	Same

# RETROSPECTIVE



This module explains how to use GitHub Copilot Chat in VS Code across Chat View, Inline Chat, and Quick Chat, and it helpfully walks through the Copilot Chat/Completions menus (status, open completions panel, disable completions, shortcuts, settings, diagnostics/logs).

It also covers modes (Ask, Edit, Agent), core slash commands (/fix, /tests, /explain, /edit), checkpoints and the action bar

# LAB 4 – MOBY DICK



The goal of this lab  
is file processing –  
starting with Moby  
Dick.

# MOBY DICK

Call me Ishmael. Some years ago—never mind how long precisely—having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people's hats off—then, I account it high time to get to sea as soon as I can. This is my substitute for pistol and ball...

# FILE PROCESSING

Goal: Use Copilot Chat to perform text processing

1. Create a lab directory. In the lab directory, add a new file called text\_summary.py.
2. Add a comment:
  - # Read a file and summarize the number of lines, words, and characters
  - Accept Copilot's function suggestion.
3. Copy mobydick.txt from the Lab directory to your directory, if not the same.
4. Run the script and verify output.

# REVISING

Ask Copilot Chat:

- Suggest improvements for handling empty files
  - Add error handling if moby dick.txt is missing
  - Display the run time of the script in milliseconds
2. Refactor the script to support command-line arguments so different file names can be provided.
  3. Use the /tests command to create unit tests
  4. Run the unit test(s)

# Lab completed



# COMMON TASKS

ERROR HANDLING, SYNTAX CHECKING, AND MORE



# INTRODUCTION



The presentation introduces GitHub Copilot Chat in VS Code as a conversational coding assistant. It expands beyond inline completions by supporting natural language queries, code edits, and even multi-step workflows through Ask, Edit, and Agent modes.

It also highlights practical tools such as checkpoints to track and roll back edits, action controls to apply or discard suggestions, and inline chat for quick changes. Features like slash commands, #mentions for context, and variable awareness show how Copilot integrates smoothly into the coding workflow.

# COPILOT CHAT



This module introduces GitHub Copilot Chat as a practical debugging partner: it spots and explains common mistakes (e.g., removing items while iterating a list), scaffolds real error handling with clear try/except patterns, helps fix syntax slips across languages, and—most importantly—catches “silent” logic errors that still run but produce wrong results.

# EXAMPLE

```
def remove_negatives(numbers):  
    for n in numbers:  
        if n < 0:  
            numbers.remove(n)  
    return numbers  
  
data = [3, -1, 5, -2, 7]  
print(remove_negatives(data))
```

# POP QUIZ: PROBLEM

From the previous slide, do you know the problem in this code?



**10 MINUTES**

# COPilot CHAT – THE FIX

```
def remove_negatives(numbers):  
    return [n for n in numbers if n >= 0]  
  
data = [1, -1, -2, 3, -4, -5, 6]  
  
print(remove_negatives(data))
```

The issue is that you're modifying the list while iterating over it. When you remove an element during iteration, Python shifts the remaining items left, so the loop skips over some elements.

# ERROR HANDLING



Error handling is an area where Copilot shines by quickly generating try-except or try-catch structures. Many developers overlook robust error handling in the early stages of coding.

Copilot can encourage better practices by suggesting error handling blocks automatically.

# EXAMPLE

```
def compute_average(filename):  
    with open(filename, "r") as f:  
        lines = f.readlines()  
        total = sum(int(line.strip()) for line in lines)  
        avg = total / len(lines)  
  
    return avg
```

# WITH ERROR HANDLING

```
def compute_average(filename):  
    try:  
        with open(filename, "r") as f:  
            lines = f.readlines()  
            if not lines:  
                raise ValueError("The file is empty.")  
            total = sum(int(line.strip()) for line in lines)  
            avg = total / len(lines)  
            return avg  
    except FileNotFoundError:  
        print(f"Error: The file '{filename}' was not found.")  
    except ValueError as ve:  
        print(f"Error: {ve}")  
    except Exception as e:  
        print(f"An unexpected error occurred: {e}")
```

Error handling is an example of repetitive code, such as try-except or try-catch structures, that GitHub Copilot handles exceptionally well.

# SYNTAX ERRORS

Syntax errors are among the most common issues developers encounter, from missing colons in Python to mismatched brackets in JavaScript. Copilot helps resolve these by offering corrected syntax based on language rules. For example, if you mistakenly write `if (x > 10)` in Python, Copilot may recognize the missing colon and suggest `if x > 10:`. Similarly, if you open an HTML tag and forget to close it, Copilot often generates the closing tag for you.

Another example is in Java: if you write `System.out.println("Hello World")` without a semicolon, Copilot can highlight the issue by suggesting the corrected line. These corrections are particularly useful for beginners or for developers working in a language they do not use frequently.

# MUST BE GOOD!



This is especially important for syntax errors that do not cause compilation errors. In languages like Python, certain mistakes still produce valid code, so the program runs but behaves incorrectly.

These errors are much harder to detect because they don't trigger obvious failures — they just produce wrong results.

# EXAMPLE

```
def check_value(x):
    message = ""
    magic_number = 10
    if (y := magic_number):
        if x > y:
            message = f"x is greater than {y}"
        elif x < y:
            message = f"x is less than {y}"
        else:
            message = f"x is equal to {y}"
    else:
        message = "Condition failed"
    return message
```

```
print(check_value(5))
print(check_value(10))
print(check_value(20))
```

# FIXED

```
def check_value(x):  
    message = ""  
    magic_number = 10  
  
    if x > magic_number:  
        message = f"x is greater than  
{magic_number}"  
    elif x < magic_number:  
        message = f"x is less than  
{magic_number}"  
    else:  
        message = f"x is equal to  
{magic_number}"  
  
    return message
```

GitHub Copilot would quickly find and fixed the syntax / logical error.

## *GENERAL DEBUGGING*

Copilot can also assist in debugging simple programs by suggesting possible reasons for logic errors and recommending alternative approaches. For example, if your loop is written incorrectly, such as for i in range(1, 10) when you intended to include 10, Copilot might suggest adjusting the loop or using range(1, 11). Similarly, if you implement a sorting function but forget to return the sorted list, Copilot may propose adding a return sorted\_list statement at the end of the function.

This becomes more powerful when combined with descriptive comments. Writing # fix the bug in this function above a function definition may lead Copilot to review the logic and suggest corrections.

# EXAMPLE

Copilot can assist in debugging simple programs by suggesting possible reasons for logic errors and recommending alternative approaches. For example, if your loop is written incorrectly, such as for i in range(1, 10) when you intended to include 10, Copilot might def fibonacci(n):

```
def fibonacci (n) :  
    return fibonacci (n - 1) + fibonacci (n - 2)
```

# RETROSPECTIVE



This module clearly shows how Copilot Chat helps you catch and fix everyday mistakes: mutating a list while iterating, undefined or incorrect calls, and common syntax slips; it then levels up practice with real error handling (file I/O guards, empty-file checks via try/except) and highlights “silent” logic errors

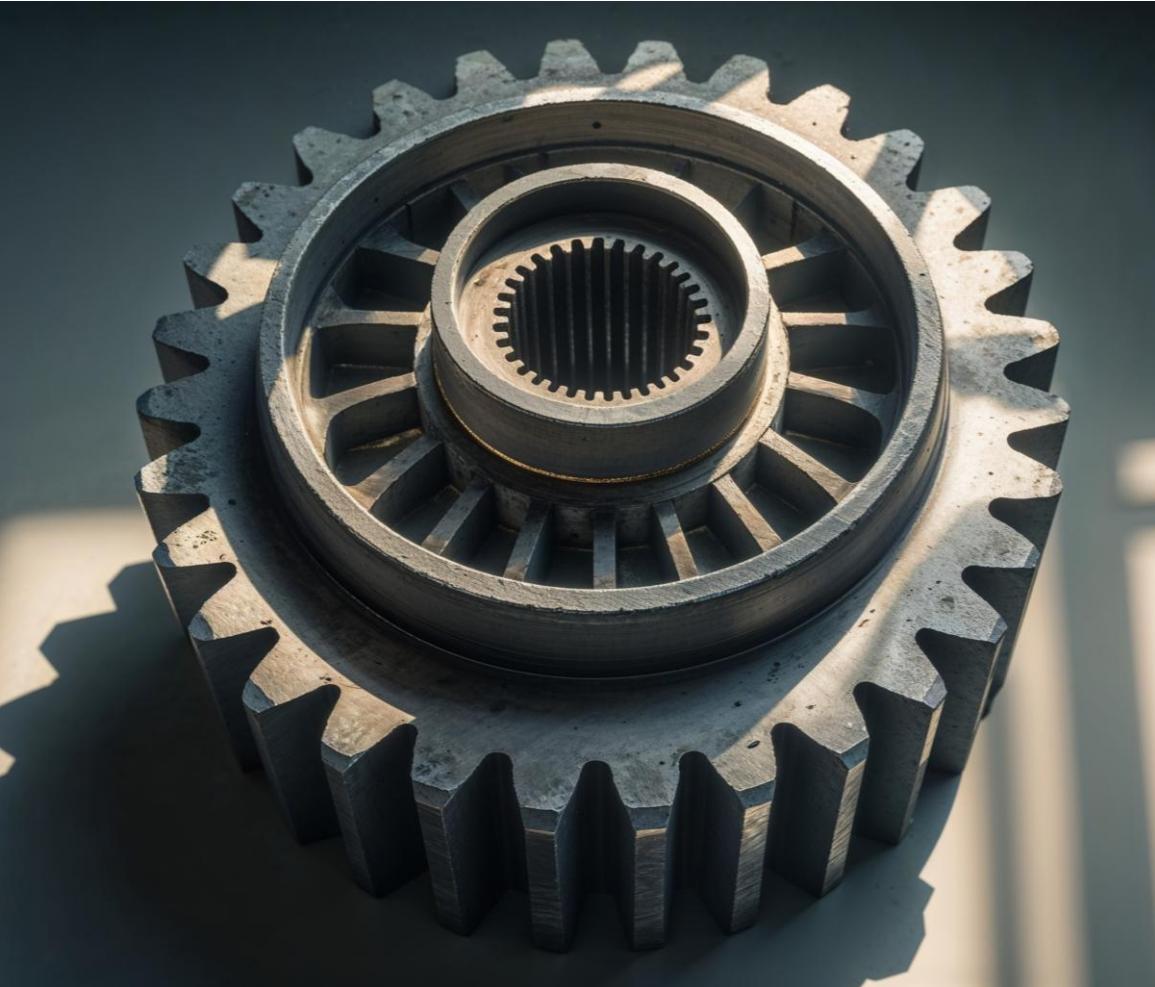
# LAB 5 – COMMON TASKS



For Python applications,  
complete common tasks  
using GitHub Copilot.

Lab instructions also in  
the `readme.md` file in the  
same directory.

# SETUP



In Vscode, open the Lab folder for Module 5.

There are several files in the lab that require some assistance from GitHub Copilot.

The goal is to successfully execute each of the programs, in some manner.

# PART A

## A1 — part\_a1.py

Run python part\_a1.py and observe the failure.

Use Copilot to introduce the missing data so the greeting can be constructed correctly.

Re-run until it prints a valid welcome message.

## A2 — part\_a2.py

Run python part\_a2.py and observe the error.

Use Copilot to correct the function call so it supplies appropriate arguments.

Re-run and verify the formatted name output looks correct.

## PART B

Create a grades.txt file in this folder. Try with it empty; then with some invalid and then valid lines.

Valid grades.txt file:

```
90  
85  
95  
100  
80
```

Run python part\_b1.py and note different failure modes.

Use Copilot to add robust error handling for file/parse issues and add a safe guard to avoid dividing by zero.

Re-run with valid numeric data and verify an average is produced.

# PART C

## Part C — Syntax Fixes

### C1 — part\_c1.py

Run `python part_c1.py` to see syntax failures.

Use Copilot to fix all syntax issues and indentation so the program runs.

Re-run and verify it prints a message and a small sequence of numbers.

# PART D

## D1 — part\_d1.py (Factorial)

Run python part\_d1.py and inspect the output/behavior.

Use Copilot to correct the logic so the function returns the correct factorial result.

(Optional) Ask Copilot for a non-recursive version to improve efficiency.

## D2 — part\_d2.py (Fibonacci Sequence)

Run python part\_d2.py and examine the length/content of the output.

Use Copilot to adjust the logic so the function returns exactly n terms.

Re-run and verify the sequence length matches the input.

# Lab completed



"Develop a passion for learning."

# CODE DOCUMENTATION



# INTRODUCTION



This module is a hands-on guide to using GitHub Copilot Chat for fast, consistent documentation and clearer code understanding. You'll practice context-aware generation of docstrings and short README snippets, refine drafts through quick iterative prompts, and learn simple prompt patterns that keep Copilot focused

# DOCUMENTATION



Copilot now uses context-aware generation, meaning it considers the function signature, surrounding code, and even related files to suggest meaningful comments or documentation.

For example, when you begin typing a docstring in Python or a Javadoc comment in Java, Copilot can propose a full template that includes a description, parameter explanations, and return values.

# SAMPLE CODE

```
def calculate_area(width: float, height: float) -> float:  
    """
```

Calculate the area of a rectangle.

Args:

width (float): The width of the rectangle.

height (float): The height of the rectangle.

Returns:

float: The calculated area of the rectangle.

```
"""
```

```
return width * height
```

# ITERATIVE

Copilot Chat further enhances this process by letting developers ask questions such as “Generate docstrings for all functions in this file” or “Explain what this class does in one sentence.” Copilot analyzes the file and responds with suggested documentation that can be inserted directly into the code.

This workflow makes documentation creation interactive—developers can refine or regenerate explanations until they meet project standards, all without leaving the editor.

# VEHICLE CLASS

```
class Vehicle:  
    """Base class representing a general vehicle."""  
  
    def __init__(self, brand: str, year: int):  
        self.brand = brand  
        self.year = year  
  
    def start(self):  
        """Start the vehicle."""  
        return f"{self.brand} is starting."  
  
    def stop(self):  
        """Stop the vehicle."""  
        return f"{self.brand} is stopping."
```

# CAR CLASS

```
class Car(Vehicle):  
    """Intermediate class representing a car, inherits from Vehicle."""  
  
    def __init__(self, brand: str, year: int, doors: int):  
        super().__init__(brand, year)  
        self.doors = doors  
  
    def honk(self):  
        """Honk the car horn."""  
        return f"{self.brand} goes 'beep beep!'"
```

# ELECTRICCAR

```
class ElectricCar(Car):  
    def __init__(self, brand: str, year: int, doors: int,  
                 battery_capacity: int):  
        super().__init__(brand, year, doors)  
        self.battery_capacity = battery_capacity  
  
    def charge(self):  
        """Recharge the electric car's battery."""  
        return f"{self.brand} is  
               charging with {self.battery_capacity} kWh capacity."
```

# POP QUIZ: QUESTIONS

What questions and documentation do you have of the previous code?



**10 MINUTES**

# OBJECT ORIENTED PROGRAMMING

In multilevel inheritance, each class in the hierarchy can add attributes, override methods, or change initialization logic. By the time you reach the most descendant class (the deepest child), its state is a mix of:

- Properties defined in the base class.
- Properties added by intermediate parent classes
- Overrides or redefinitions in subclasses.

# QUESTIONS

This layering makes it difficult to immediately know:

- Which attributes exist (some may be shadowed or overwritten).
- Where each attribute was introduced (base vs. intermediate vs. child).
- What initialization code actually ran (since constructors in Python, Java, C++, etc. may call super() in different ways).

# DOCUMENTATION - BEST PRACTICES

- Write the docstring first: start a docstring/Javadoc header and let Copilot scaffold params/returns, then edit for brevity.
- Constrain scope with context: use #selection (only selected code) or @file/@workspace to feed the right amount of code; smaller context → tighter docs.
- Seed with style hints: a first line like “Use imperative mood, ≤80-char lines, include examples” steers output.

# PROMPT - BEST PRACTICES

- Iterate, don't accept wholesale: accept a draft, then ask in Chat: "tighten language," "remove redundancies," "convert to bullets."
- Template reuse via prompt files: keep a docstyle.md (tone, sections, examples). Reference it in prompts so Copilot stays consistent.
- Ask for contrasts/examples: "Add a 1-line example and a common pitfall" produces practical docs without bloat.

# PROMPT - BEST PRACTICES - 2

docstring:

- #selection @file:utils.py Create a concise docstring: 1-line summary, args, returns, edge cases. Use imperative mood.

README snippet:

- @workspace Generate a minimal README section: install, run, test. Keep steps numbered and under 7 lines.

Refine for brevity:

- Tighten this to half the length; keep technical accuracy and example.

# PROMPT - BEST PRACTICES - 3

Write the docstring first:

- Start a docstring/Javadoc header and let Copilot scaffold params/returns, then edit for brevity.

Constrain scope with context

- Use #selection (only selected code) or @file/@workspace to feed the right amount of code; smaller context → tighter docs.

Seed with style hints:

- A first line like “Use imperative mood, <80-char lines, include examples” steers output.

# PROMPT - BEST PRACTICES - 4

Iterate, don't accept wholesale:

- Accept a draft, then ask in Chat: "tighten language," "remove redundancies," "convert to bullets."

Write the docstring first:

- Start a docstring/Javadoc header and let Copilot scaffold params/returns, then edit for brevity.

Constrain scope with context

- Use #selection (only selected code) or @file/@workspace to feed the right amount of code; smaller context → tighter docs.

Seed with style hints:

- A first line like "Use imperative mood, ≤80-char lines, include examples" steers output.

# PROMPT - BEST PRACTICES - 5

Template reuse via prompt files:

- Keep a docstyle.md (tone, sections, examples). Reference it in prompts so Copilot stays consistent.

Ask for contrasts/examples:

- “Add a 1-line example and a common pitfall” produces practical docs without bloat.

# RETROSPECTIVE



This module showed how to use GitHub Copilot for fast, consistent documentation: it demonstrated context-aware generation of docstrings that consider signatures, surrounding code, and related files.

# LAB 6 – ESCAPE VELOCITY

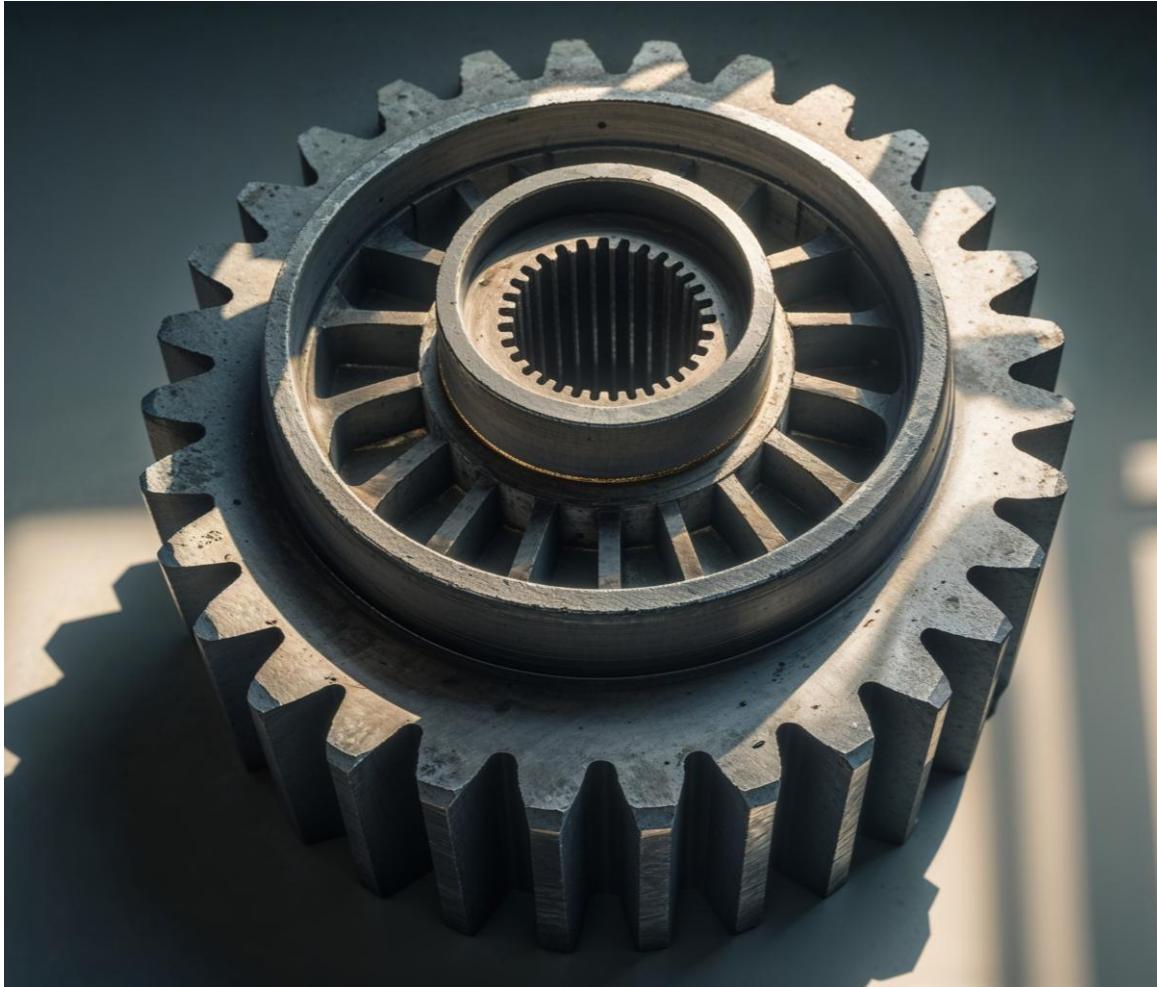


Document an application that computes escape velocity at a given altitude.

Estimates propellant mass to achieve that  $\Delta v$  via the rocket equation,

Uses Copilot to add docstrings and to create a README.md with a function table and a data table of examples.

# SETUP



In Vscode, open the Lab folder for Module 5.

There are several files in the lab that require some assistance from GitHub Copilot.

The goal is to successfully execute each of the programs, in some manner.

# SMOKE TEST

From the project folder, run in the module5/lab/space-lab directory:

```
python main.py
```

You should see three lines (escape speed in m/s,  $\Delta v$  in m/s, propellant in kg).

If it fails, fix runtime issues first (paths, Python version, etc.).

```
C:\Code\Copilot\class-prep\python\module6\lab\space-lab>python main.py
Escape speed: 11015 m/s
Δv (with margin): 12667 m/s
Propellant required: 66747 kg
```

Familiarize yourself with the files in the directory.

# DOCSTRINGS

Goal! Add short docstrings to existing code and generate a simple README with a function table and a tiny data table — using Copilot Chat.

## 1) Docstring for AstroMath.escape\_velocity

Do: Add a short, clear docstring.

Action: Open astro.py, select only the escape\_velocity function.

Enter in Copilot Chat (A then B then C):

- Write a short docstring. One sentence.
- Add Args and Returns with units.
- Add formula text:  $v = \sqrt{2\mu/r}$ . Max 6 lines total.

# DOCSTRINGS - 2

## 2) Docstring for AstroMath.propellant\_needed

Do: Add a short docstring that explains the rocket equation result.

Action: Select only the propellant\_needed function.

Enter in Copilot Chat (A then B then C):

- Short docstring. One sentence about the rocket equation.
- Add Args: dv m/s, isp s, dry\_mass kg. Returns: propellant kg.
- Keep under 6 lines. No derivation.

# DOCSTRINGS - 3

## 4) Docstring for MissionPlanner (class)

Do: Describe the pipeline and the loss margin.

Action: Select the class header and init together.

Enter in Copilot Chat (A then B then C):

- Short class docstring. One or two sentences.
- Say steps: compute escape speed, add dv margin, compute propellant.
- Explain loss\_margin in plain English. Max 5 lines.

# DOCSTRINGS - 4

Create README.md (sections + table + example)

Do: Generate a short README with simple words.

Action: Open empty README.md.

Enter in Copilot Chat (A then B then C):

- Create README with sections: Overview, Functions and Methods, Example. Use simple words.
- Add a table: Name | Purpose | Key args | Returns. Rows: AstroMath.escape\_velocity, AstroMath.propellant\_needed, MissionPlanner.fuel\_to\_escape.
- Keep under 30 lines.

## TEST AND EXAMINE

Test the application. Naturally, it should still run.

View the files again to observe the requested changes.

# Lab completed



"Develop a passion for learning."

# ADVANCED

SETUP, REFERENCE, AND MORE



# INTRODUCTION



To support different needs, Copilot Chat includes features such as prompt files (reusable macros), instruction files (persistent guidelines), tool sets (collections of tools for agent mode), and modes (customized behavior profiles).

It also connects to MCP servers, which act like plugins to provide access to external tools and data. Together, these elements allow you to shape Copilot into a flexible coding assistant that adapts to your project's standards, workflows, and integrations

# PREMIUM REQUESTS

Every time you interact with Copilot—whether via chat, agent mode, code review, or extensions—you’re making a request. Some of these are considered premium requests, meaning they consume from a monthly quota and may incur additional charges if you exceed your limit.

# PREMIUM REQUESTS - 2

## How Many Premium Requests Does Each Plan Include?

Here's how the plans stack up:

Plan	Monthly Premium Requests Included	Cost for Extra Requests
Free	50 per month	Cannot purchase more—must upgrade <a href="#">GitHub Docs +3</a> <a href="#">Reddit</a>
Pro	300 per month	\$0.04 USD per extra request <a href="#">GitHub Docs +2</a>
Pro+	1,500 per month	\$0.04 USD per extra request <a href="#">GitHub Docs +2</a>

# PREMIUM REQUESTS

You open Copilot Chat in VS Code and type:

Action:

"Refactor this Python function to improve performance and explain the changes."

Since this is Copilot Chat (not just inline code completion), it counts as a premium request.

The model you choose determines how many premium requests it consumes:

If you use GPT-4.1 (included model), cost = 0 premium requests.

If you switch to Claude Sonnet 3.5 (multiplier = 1×), cost = 1 premium request.

If you pick Claude Opus 4 (multiplier = 10×), cost = 10 premium requests.

# CHAT MESSAGE



A chat message is the unit of interaction inside Copilot Chat.

It's literally each prompt you type and each AI reply.

Example:

You: "Explain this function." is 1 chat message

Copilot reply is 1 chat message

Total = 2 chat messages for that exchange.

# CHAT MESSAGE VERSUS PREMIUM REQUEST

## Comparison Table

Feature	Chat Message?	Premium Request?	Explanation
Inline code completion (ghost text / suggestion while typing)	No	No	Standard Copilot feature, included in base subscription; unlimited.
Copilot Chat prompt (your input)	Yes	Yes	Each prompt is a chat message and consumes premium request units.
Copilot Chat reply (AI output)	Yes	Yes	Each reply is a chat message and also consumes premium request units.
Copilot Agents (/fix, /new, /testfailure, etc.)	No	Yes	Invokes advanced AI features; not a chat but billed as premium requests.
PR review / summary	No	Yes	Uses Copilot to analyze diffs; premium request consumption.
Copilot Extensions (queries to third-party services via Copilot)	No	Yes	Consumes premium requests, but not counted as chat messages.

# TOKENS

The unit of text the model processes. Both input (what you type + hidden context like system instructions, recent chat, file summaries) and output (the reply) are counted in tokens. Rough rule: ~4 characters ≈ 1 token.

- Cost/quotas: Premium usage is measured in tokens; longer prompts/replies → more tokens → more premium budget used.
- Context limits: Models have max token windows. If you exceed them, Copilot truncates context, which can hurt answer quality.

# LOCAL INDEX

Means Copilot has made a quick list of the files you're using right now on your laptop. Typically just the files opened in Vscode and sometimes those you've recently touched.

Copilot is saying: "I've gone through your home bookshelf, and I know where the books are."

Private — never leaves your machine.



# WORKSPACE INDEX

The workspace simply means everything in your project folder. It's the full collection of files that make up your codebase — not just the one you're currently editing.

The workspace index is like Copilot's table of contents for that entire folder. By creating this index, Copilot knows what files exist, where they are, and what they contain at a high level. This catalog allows Copilot to connect the dots between different parts of your project, so it can provide more useful completions and answer broader questions about your code.

# REMOTE INDEX

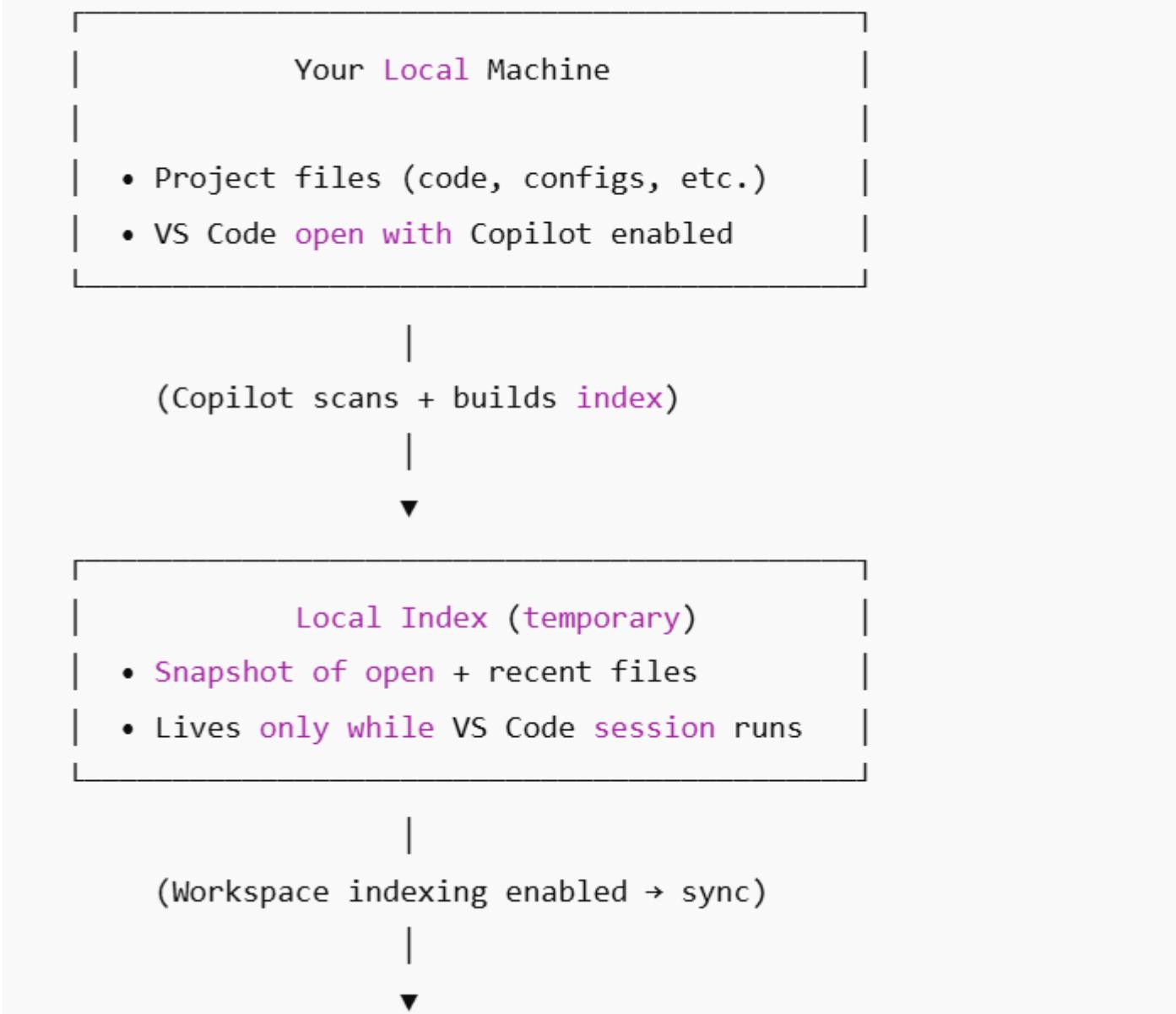
Remote is just Copilot keeping a cloud copy of what you have locally indexed, so it can provide context-aware help across the project.

The remote index is built from your local project (the files on your machine in VS Code). Copilot scans those and sends an index (a kind of table of contents) up to GitHub's servers.

That way, even if you close a file locally, Copilot can still "remember it" because the index lives remotely.



# INDEXING FLOW



# INDEXING FLOW - 2

## Remote Index (on GitHub)

- Cloud copy of your project's index
- Built from files on \*your machine\*
- Lets Copilot "remember" the repo even when files are closed locally

(Used for completions + chat queries)



## Copilot Suggestions

- Smarter completions
- Answers about any file in the repo
- Cross-file understanding

# WHY CONFIGURE INDEXING?

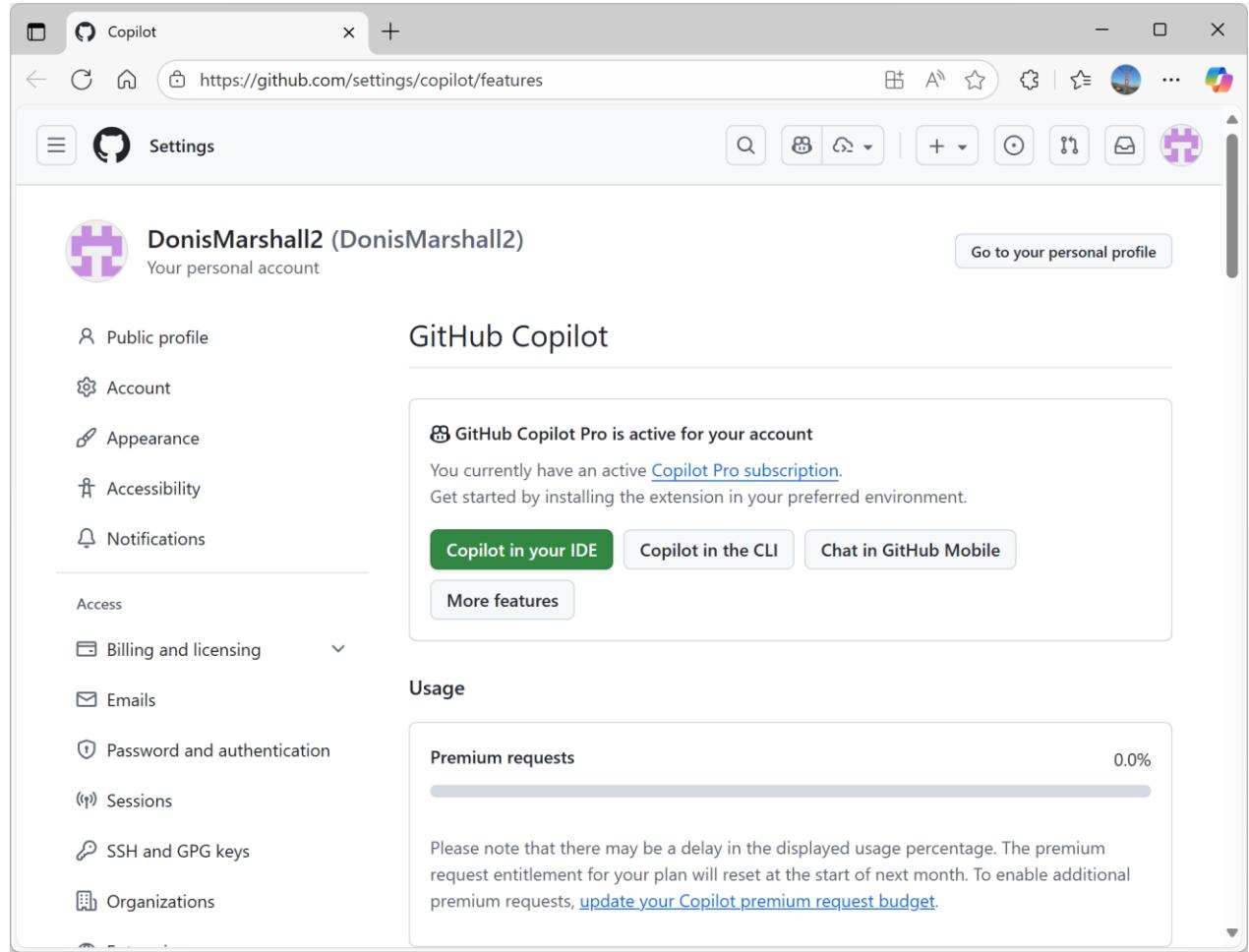
- Local Index Only (safer, private):

Use this if you're working with sensitive files (like client data, finances, or company secrets) and don't want anything analyzed in the cloud.

- Workspace + Remote Index (smarter, but less private):

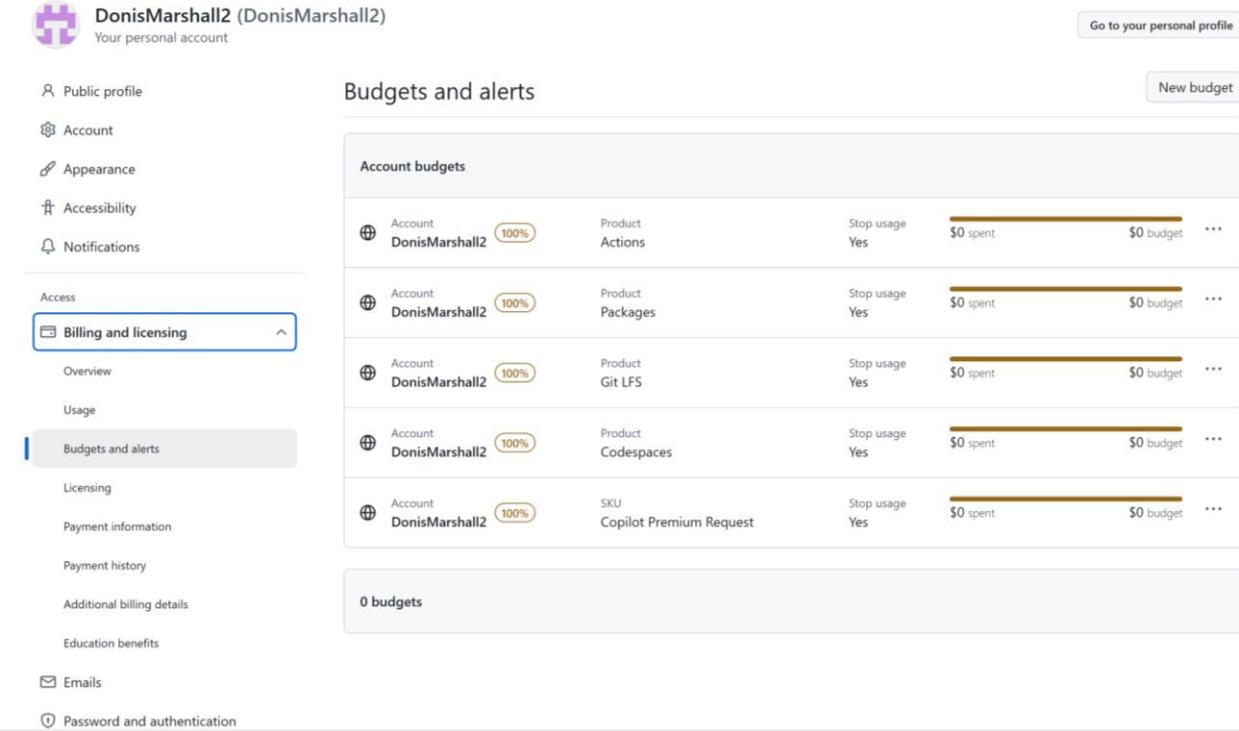
Use this if you want Copilot to be really good at answering across your whole project — like when your project has many files that depend on each other.

# COPilot USAGE



The Copilot Usage menu item opens the GitHub settings for Copilot located at the Github web portal.

# MANAGE PAID PREMIUM REQUESTS



The screenshot shows the GitHub Billing and licensing interface. On the left, there's a sidebar with account details (DonisMarshall2), navigation links (Public profile, Account, Appearance, Accessibility, Notifications, Access, Billing and licensing, Usage, Budgets and alerts, Licensing, Payment information, Payment history, Additional billing details, Education benefits, Emails, Password and authentication), and a 'Go to your personal profile' button. The 'Billing and licensing' section is currently selected. In the main area, under 'Budgets and alerts', there's a table titled 'Account budgets' showing five entries: Account DonisMarshall2 (100%) for Product Actions, Product Packages, Product Git LFS, Product Codespaces, and SKU Copilot Premium Request. Each entry includes a 'Stop usage Yes' button, '\$0 spent' and '\$0 budget' fields, and a '...' button. Below the table, it says '0 budgets'.

For the current GitHub account, opens the Budgets and alerts window in GitHub.com. From here, you can set the budget for premium requests.

# HOW TO CONFIGURE INDEXING?

To build the index you want, type and run one of these commands (Ctrl+Shift+P):

- GitHub Copilot: Build Remote Workspace Index — makes a cloud (remote) index for your GitHub-hosted project.
- GitHub Copilot: Build Local Workspace Index — creates a private index on your machine (if the project has up to 2,500 files).

# REFERENCES ( @ )

Use when you want Copilot to look at built-in context from your workspace, project, or IDE.

Reference	What It Does	Example Prompt	🔗
@file	Points to a specific file in your project	"Summarize @app.py"	
@functionName	Points to a specific function/class/variable	"Explain how @calculateInterest works"	
@problems	Looks at the Problems/Errors panel in your editor	"Fix the issues in @problems"	
@workspace	Tells Copilot to consider the whole project	"Generate documentation for @workspace"	
@pr	References the current Pull Request (in GitHub)	"Summarize the changes in @pr"	

# MENTIONS ( # )

Use when you want to bring in or highlight something manually for Copilot.

Reference	What It Does	Example Prompt
#file	Adds the contents of a file you specify	"Write tests for #file:customer.py"
#selection	Uses only the code you've highlighted	"Refactor this loop #selection"
#codebase	Consider the entire codebase (if indexed)	"Find security issues in #codebase"
#notebook	Adds context from a Jupyter notebook	"Summarize results from #notebook"

## @ VERSUS #

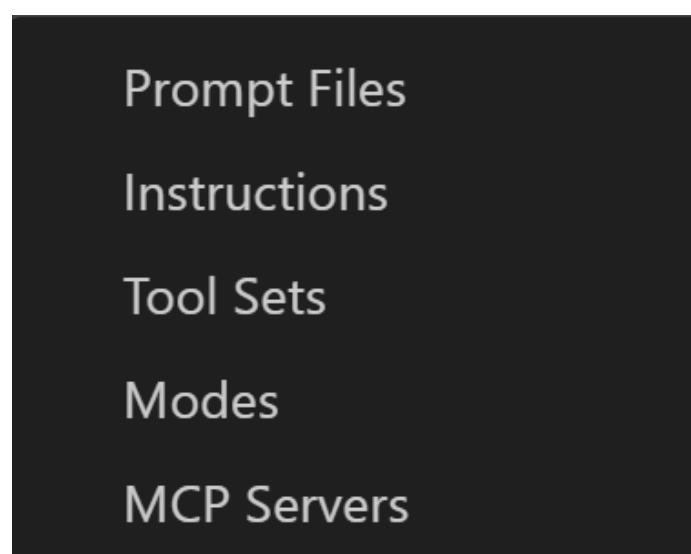
@ = who answers. Pick the model/agent to handle your request.

Example: @gpt-4o write a Jest test for this

# = what to look at. Point Copilot to context (selection, file, codebase, tool, prompt).

Examples: #selection explain this function · #file summarize · #codebase find usages · #tool ping run

# COPILOT GEAR SETTINGS MENU



The Copilot Gear Settings menu is at the top-right of the GitHub Copilot Chat side panel in Vscode. The menu items you mentioned—Prompt Files, Instructions, Tool Sets, Modes, MCP Servers, and Generate Instructions—each offer targeted ways to customize Copilot Chat to your workflow.

# PROMPT FILES

A Prompt File is a Markdown file (.prompt.md) that stores a reusable prompt template.

Instead of retyping the same instructions in Copilot Chat, you put them into a .prompt.md file and run it whenever needed.

They can include metadata (front-matter at the top) that controls how Copilot should run that prompt: what mode, which model, which tools, and even a short description.

Think of them like saved macros for conversations with Copilot, but written in plain text/Markdown.

# PROMPT FRONT-MATTER

Front-matter metadata (optional, YAML style, between --- lines):

---

```
title: "Refactor for performance"
```

```
description: "Optimize code for runtime efficiency"
```

```
mode: agent
```

```
model: gpt-4
```

```
tools: [filesystem, terminal]
```

---

```
title: human-friendly name for the prompt
```

```
description: short explanation (shows up in menus)
```

```
mode: Copilot Chat mode (e.g., ask, edit, agent)
```

```
model: preferred AI model
```

```
tools: which Copilot/extension/MCP tools can be used
```

## PROMPT

After the final --- of the front matter, add the actual prompt.

Please analyze the given code for performance bottlenecks.  
Suggest improvements without changing functionality.

# WHERE TO PLACE?

Workspace level: Place .prompt.md files in a special folder like:

.github/prompts/

This makes the prompt available to everyone working on that repo/workspace.

User level: You can also keep them in your user config, so they're available across all projects.

# HOW TO USE

There are different ways to execute the prompt (macro):

- Open the Copilot Chat panel
  - gear menu
  - Prompt Files
  - New/Open/Run.
- Use the Command Palette:
  - Chat: New Prompt File → create one
  - Chat: Run Prompt → pick and execute one
- Trigger directly in chat with a slash command:  
`/refactor-performance`

# INSTRUCTIONS

Instructions are similar to prompt files, except setting guidelines and directives for Copilot, instead of an actual prompt.

It sets persistent guidelines—so you don't have to keep reminding Copilot in every prompt.

Think of it like a style guide or coding policy document that Copilot automatically follows.

It's typically named `.copilot-instructions.md` (or similar) and placed at the root of your repo or inside `.github/`.

# FRONT MATTER

An instruction file is plain Markdown, often with a YAML front-matter block (optional) at the top – similar to a Prompt file:

---

title: "Project Instructions"

description: "Guidelines Copilot should always follow in this repo"

# RULES / GUIDELINES

After the Front Matter, you write the rules in Markdown:

## # General Guidelines

- Always write code in TypeScript, not JavaScript.
- Use async/await instead of callbacks.
- Follow Airbnb style guide for naming and formatting.
- Add Doc comments for every exported function.

## # Testing

- All functions must include a Jest unit test.
- Use `describe`/`it` blocks, not `test`.

# RULES / GUIDELINES – 2

## # Security

- Never hardcode secrets or API keys.
- Recommend environment variables (`process.env`) for config.

# TOOL SETS



A Tool Set is a collection of tools that Copilot Chat can use when running in Agent Mode.

- Tools can come from:
- Vscode extensions (e.g., GitHub Issues, PRs, filesystem, terminal).
- MCP servers (Model Context Protocol servers you or others configure).
- Built-in Copilot capabilities.

A Tool Set lets you group those tools together, so you can enable/disable them as a unit.

# MODES

A Mode File is a configuration file (.chatmode.md) that defines a custom chat mode for GitHub Copilot Chat.

It tells Copilot how it should behave when you switch into that mode:

- Which instructions to follow.
- Which tools / tool sets it's allowed to use.
- Which model to run on (GPT-4, GPT-4o, etc.).
- What description/title to show in the UI.

Think of it as a preset personality + capabilities profile for Copilot Chat.

# MCP SERVER

A Model Context Protocol (MCP) Server is an external program that exposes tools, data, or APIs to GitHub Copilot Chat in a standardized way.

Copilot acts as the MCP client, and when you enable a server, it can call that server's tools in Agent Mode.

Think of it like plugins for Copilot that run as independent processes that Copilot can talk with.

# MCP SERVER WHERE?

GitHub ships some built-in ones (filesystem, terminal, GitHub Issues/PRs).

You can write your own MCP servers in any language that upholds a published interface and standards.



# MCP – THE PROTOCOL

MCP is like *USB-C* for AI apps.

It gives AI apps (e.g., VS Code Copilot, Claude Desktop) one standard plug to talk to outside programs called servers. Those servers offer tools (do something), data (read something), and prompts (templates to ask the AI).

- The app plugs in to a server.
- It sees what's available (which tools/data/prompts exist).

It uses them in a consistent way. Why this matters: you don't need a different, custom integration for every app + tool. If both sides speak MCP, they just work—one standard instead of many adapters.

# FASTMCP

FastMCP is a developer-friendly library that makes it easy to stand up an MCP server without having to write all the boilerplate code.

Instead of manually handling JSON-RPC, message passing, and lifecycle management, you just define your tools (functions, APIs, resources), and FastMCP exposes them through MCP automatically.

Think of it like *Fast API* but for MCP servers: minimal setup, fast iteration, built-in scaffolding.

# FASTMCP - 2

## Features of FastMCP:

- Quick server creation: register functions as MCP tools with decorators.
- Auto-wiring: handles protocol messages, introspection, and tool listing automatically.
- Python-native: you define tools in Python and FastMCP takes care of wrapping them in MCP-compliant responses.
- Integration with clients: servers built with FastMCP can be added to GitHub Copilot Chat, ChatGPT (with MCP support), or other MCP-capable assistants.

# SAMPLE CODE

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("hello-mcp")

@mcp.tool()

def ping() -> str:
    """Returns 'pong'."""
    return "pong"

if __name__ == "__main__":
    mcp.run(transport="stdio")
```

Creating a MCP Server is outside the scope of this class. Importantly, MCP is language-agnostic—it's just JSON-RPC over transports like stdio or HTTP/SSE—so you can write a server in any language that can read/write JSON on those pipes.

# SAMPLE CODE – CONFIGURATION FILE – MCP.JSON

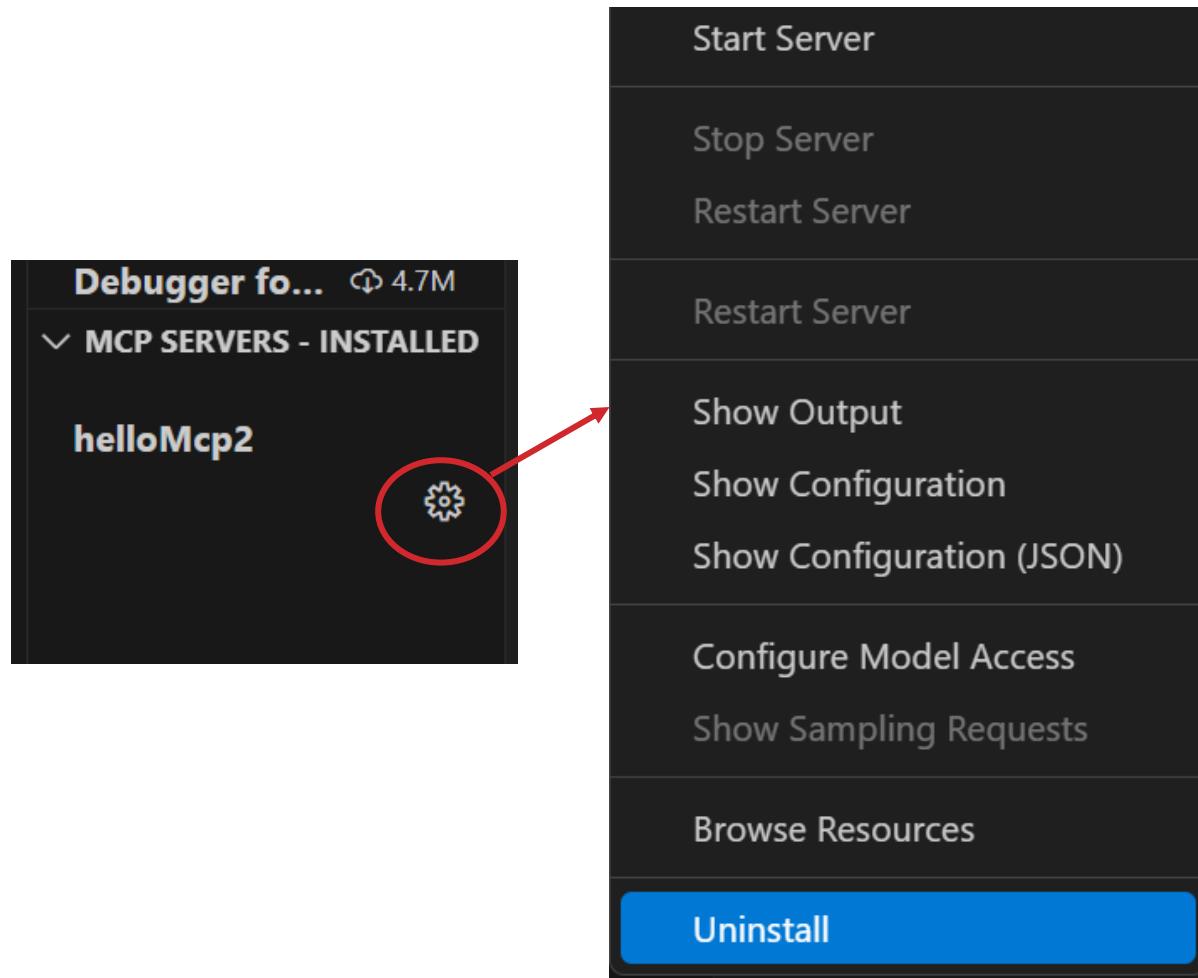
```
{  
  "servers": {  
    "helloMcp2": {  
      "type": "stdio",  
      "command": "python",  
      "args": ["${workspaceFolder}/server.py"]  
    }  
  }  
}
```

Tells VS Code/Copilot which MCP servers to use and how to start them: choose the transport (stdio or http), set the launch command and args (or a remote url), and optionally define inputs (e.g., API keys).

Where to put it:

./vscode/mcp.json

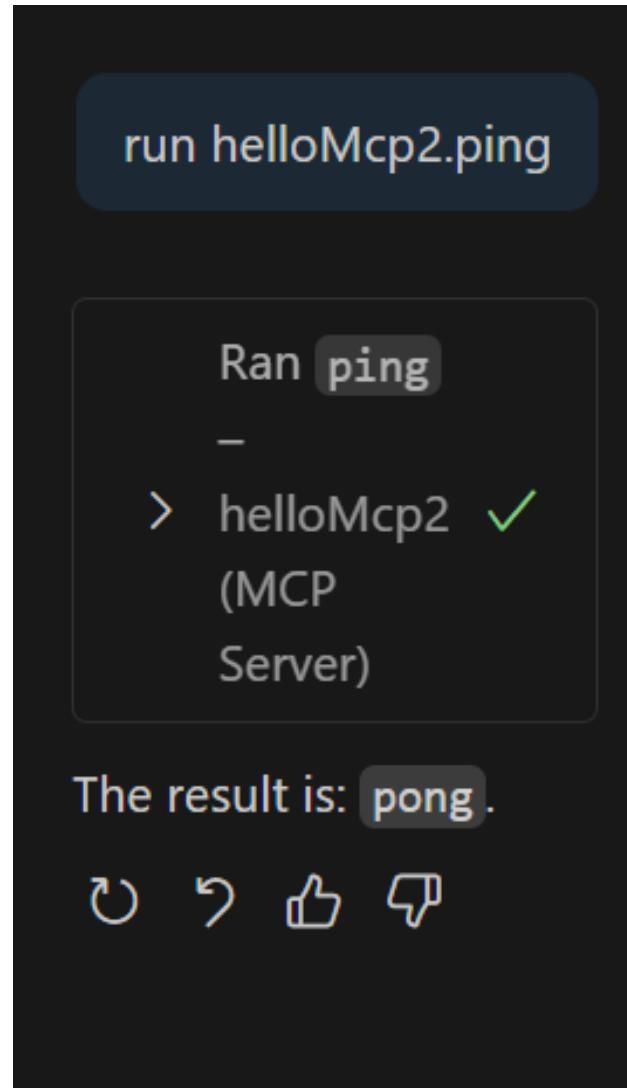
# MCP SERVER PANE



For the Chat Gear Menu, the MCP Servers menu items opens the MCP Server Pane.

To start a MCP server, select the gear menu adjacent to the server name.

# MCP COMMAND



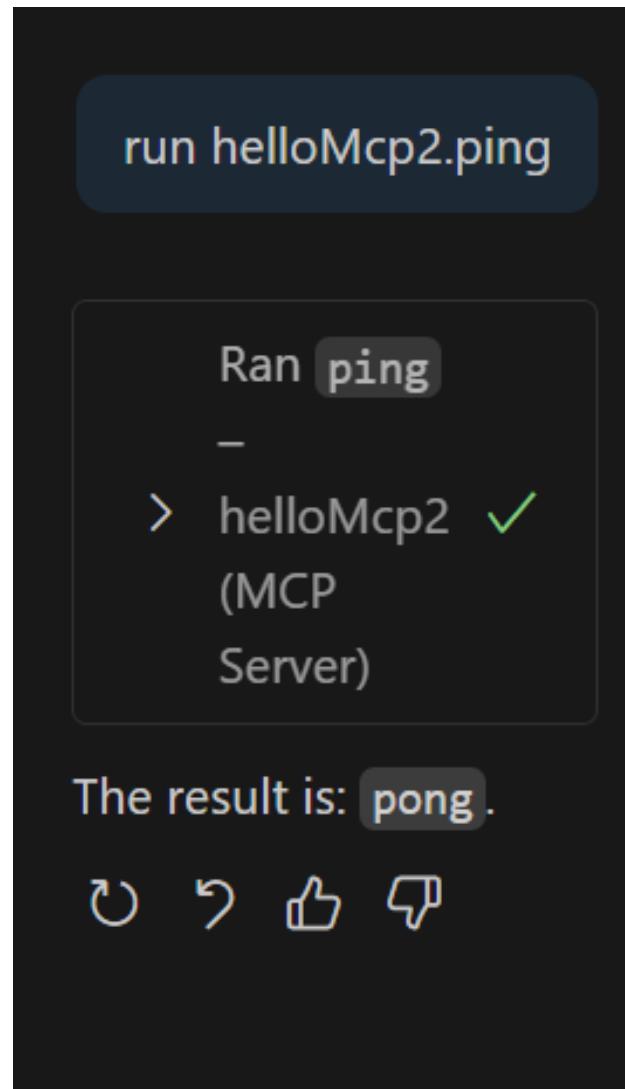
To execute a MCP command, refer to the *mcp.command* in the prompt.

For example:

```
run helloMcp.ping
```

You should see a “tool call” step and the result (for ping, it should be “pong”).

## MCP COMMAND - 2



If the command doesn't run:

- Command Palette can reset the environment: MCP: Reset Cached Tools or Developer:Reload Window.
- Make sure you're in Agent mode.

# MCP SERVERS

Repository for example MCP Servers.

<https://shorturl.at/gqXrr>

## 🔗 Reference Servers

These servers aim to demonstrate MCP features and the official SDKs.

- [Everything](#) - Reference / test server with prompts, resources, and tools.
- [Fetch](#) - Web content fetching and conversion for efficient LLM usage.
- [Filesystem](#) - Secure file operations with configurable access controls.
- [Git](#) - Tools to read, search, and manipulate Git repositories.
- [Memory](#) - Knowledge graph-based persistent memory system.
- [Sequential Thinking](#) - Dynamic and reflective problem-solving through thought sequences.
- [Time](#) - Time and timezone conversion capabilities.

## 🔗 Archived

The following reference servers are now archived and can be found at [servers-archived](#).

- [AWS KB Retrieval](#) - Retrieval from AWS Knowledge Base using Bedrock Agent Runtime.
- [Brave Search](#) - Web and local search using Brave's Search API. Has been replaced by the [official server](#).
- [EverArt](#) - AI image generation using various models.
- [GitHub](#) - Repository management, file operations, and GitHub API integration.
- [GitLab](#) - GitLab API, enabling project management.
- [Google Drive](#) - File access and search capabilities for Google Drive.
- [Google Maps](#) - Location services, directions, and place details.
- [PostgreSQL](#) - Read-only database access with schema inspection.
- [Puppeteer](#) - Browser automation and web scraping.
- [Redis](#) - Interact with Redis key-value stores.
- [Sentry](#) - Retrieving and analyzing issues from Sentry.io.
- [Slack](#) - Channel management and messaging capabilities. Now maintained by [Zencoder](#)
- [SQLite](#) - Database interaction and business intelligence capabilities.

# RETROSPECTIVE



This module highlighted how GitHub Copilot Chat transforms the way developers work by combining natural language interaction with flexible customization options. We explored the distinctions between chat messages and premium requests, the importance of indexing (local, workspace, and remote) for providing context, and how usage and budgeting controls help manage premium resources effectively.

# LAB 7 – PROMPT FILE



This lab has you create a tiny, messy factorial.py, run it once, then use a Copilot Prompt File (saved as .github/prompts/myprompt.md) to refactor the code for readability.

You'll add front-matter (title, description, mode: edit, model), include a short "rewrite for readability" instruction, and execute it from Copilot Chat as a slash command to produce cleaner names, comments, and formatting.

# DESCRIPTION



Prompt files in GitHub Copilot Chat are like reusable macros written in Markdown (.prompt.md). They hold custom instructions you can run on demand, with optional metadata like title, description, model, and tools. Instead of retyping the same guidance, you save it once and trigger it whenever needed, ensuring consistent and repeatable responses.

You can store prompt files in a workspace folder (e.g., .github/prompts/) or your personal config..

# SETUP

1. Create a new directory for the lab.
2. In the directory, create a file named factorial.py
3. Add comments to the beginning of the file:

```
#create factorial function that unreadable
```

```
#create a main that calculates 5! using the factorial function
```

```
#display the results
```

4. Run the program

# CREATE PROMPT.MD FILE

In VS Code, open the Copilot Chat side panel.

- You'll see it as a chat bubble icon in the left sidebar, or open with Ctrl+Shift+I (Windows/Linux) or Cmd+Shift+I (Mac).
- At the top-right of the chat panel, click the gear icon.
- Select Prompt Files → New Prompt File.
- Create the file at the project directory:  
`.github/prompts/myprompt.md`

# INITIALIZE FILE

Add this front-matter metadata at the beginning of the prompt.md file:

---

title: "Refactor for Readability"

description: "Clean up messy code and improve readability"

mode: edit

model: gpt-4

---

After the front matter, add the prompt below:

Please rewrite the selected code to improve readability.

Use meaningful variable names, add comments where necessary, and follow standard formatting practices.

# EXECUTE PROMPT.MD FILE

- Run Your Prompt as a slash command (/)
  - Select the factorial code
  - Open an inline chat prompt
  - In Copilot Chat, type /refactor-for-readability

The factorial function should now be readable.

# Lab completed



"Develop a passion for learning."

# AGENT MODE

SETUP, REFERENCE, AND MORE



# INTRODUCTION



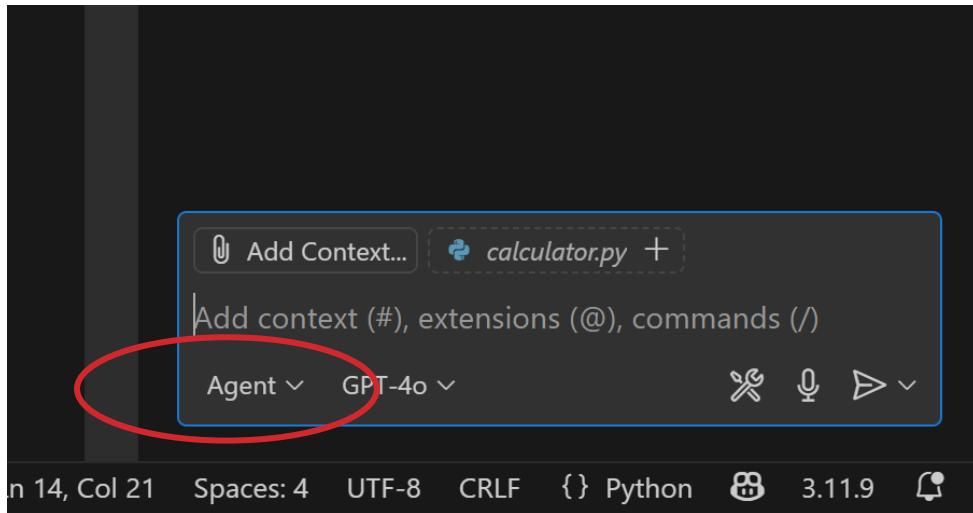
The slides show how GitHub Copilot Chat moves beyond simple completions into a smarter assistant. They highlight Agent Mode's strengths in iterative tasks, multipass refinement, and external tool use, with examples like generating unit tests using unittest, running them, and fixing failures. This illustrates how Copilot can manage workflows that go beyond single edits.

# AGENT MODE

Agent mode in GitHub Copilot goes beyond simple inline completions by acting like an intelligent assistant across your workspace. Instead of only filling in code, it can scaffold projects, refactor multiple files, or set up tests based on natural language instructions. With deeper awareness of your codebase, it handles broader workflows that span more than a single line of code, making it especially valuable for multi-step or multi-file tasks.

What makes agent mode distinct is its ability to chain actions and interpret complex prompts. You can ask it to generate a new feature, add tests, and update documentation in one request. Unlike standard chat or inline suggestions, which are limited to local context, agent mode works with the whole project. It functions more like a collaborator, automating repetitive work and letting you focus on higher-level design and problem-solving.

# SET AGENT MODE



Agent Mode in GitHub Copilot can be enabled from the Copilot Chat side panel in VS Code, where you switch between modes like Edit or Agent. It can also be set through the gear menu in Copilot settings or the command palette by searching for Copilot commands. These options give flexibility for toggling Agent Mode whether you prefer the side panel, menus, or keyboard shortcuts.

## AGENT - ITERATIVE



In Agent Mode, Copilot doesn't just generate a single answer and stop. It can revisit its own output, re-check, and improve it in multiple passes. This allows for more polished results and is particularly useful for complex tasks like generating multi-file scaffolds, debugging flows, or evolving designs step by step.

# AGENT – EXTERNAL INTEGRATION



Unlike standard Copilot chat, Agent Mode can trigger or coordinate with tools outside the editor (depending on configuration). For example, it might run linters, test frameworks, or project commands to validate changes.

## AGENT MODE – UNIQUE FEATURES

Multi-file awareness – It can make coordinated changes across several files, not just the one you're editing.

Task chaining – Able to execute multi-step instructions (e.g., scaffold a feature, add tests, and update docs in a single request).

Workspace-level context – Uses the project's structure and relationships, not just the current file, to generate better results.

Automation of workflows – Goes beyond code completion to handle setup, refactoring, or boilerplate creation at scale.

## AGENT MODE – UNIQUE FEATURES - 2

- Multi-file awareness – It can make coordinated changes across several files, not just the one you're editing.
- Task chaining – Able to execute multi-step instructions (e.g., scaffold a feature, add tests, and update docs in a single request).
- Workspace-level context – Uses the project's structure and relationships, not just the current file, to generate better results.
- Automation of workflows – Goes beyond code completion to handle setup, refactoring, or boilerplate creation at scale.

## AGENT MODE – UNIQUE FEATURES - 3

- Natural language commands – Accepts broader, high-level prompts (e.g., "Add user authentication with tests") rather than just line-by-line guidance.
- Consistency across files – Ensures new code, tests, and documentation align with the existing project style and architecture.
- Broader scope than edit mode – Instead of limited local edits, it works like a collaborator who can manage larger development tasks.

# COMPARISONS

Mode	Scope	Context	Iteration	Output
Agent Mode	Multi-file, workspace-wide	Full project	Multipass	Coordinated changes across files
Edit Mode	Single file or selection	Local only	One-pass	Single edit suggestion
/new Command	Creates new project files	Minimal	One-pass	Project scaffold with starter files

# WALKTHROUGH – AGENT



Distinguishing Agent mode with a classic prompt.

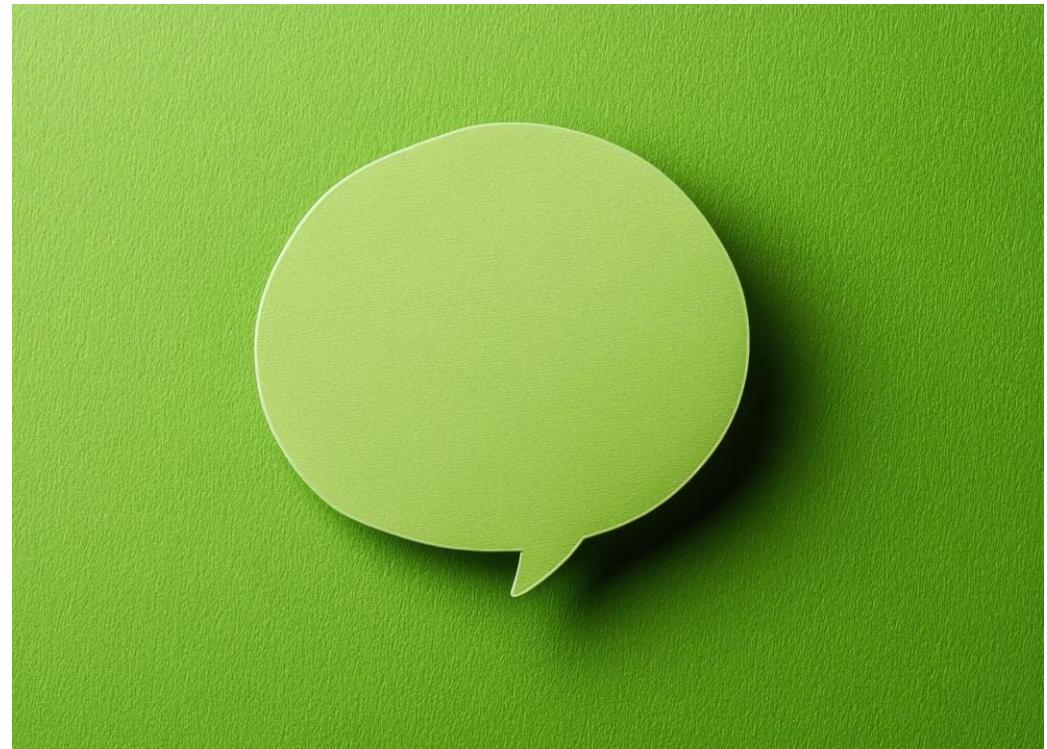
# CALCULATOR.PY WITH ERROR

```
class Calculator:  
    def add(self, a, b):  
        return a - b  
  
    def subtract(self, a, b):  
        return a - b
```

```
def multiply(self, a, b):  
    return a * b  
  
def divide(self, a, b):  
    if b == 0:  
        raise ValueError(  
            "Division by zero is not ...")  
    return a / b
```

# MULTIFACETED PROMPT

"Generate unit tests for the Calculator class using Python's unittest framework, run the tests, and fix any failures."



# WHY AGENT MODE?

Why it works in Agent Mode:

- Creates new test files with unittest.
- Executes the tests as an external tool.
- Iteratively fixes errors until tests pass (multipass + tool integration).

Why it fails in Edit Mode:

- Edit Mode can only edit the selected code. It cannot run unit tests with unittest, or iterate on failures.

**JUST DO IT.**

# RETROSPECTIVE



This module demonstrated how GitHub Copilot Chat can support developers in ways that go well beyond inline code completion. We examined the unique capabilities of Agent Mode, such as task chaining, multipass refinement, and integration with external tools, and contrasted these with the more limited scope of Edit Mode.

Through examples like generating unit tests, the module showed where Agent Mode is best applied and how it can streamline workflows.

# LAB 8 – AGENT / EDIT



Review the following list of prompts. If the prompt requires the Agent mode, circle Yes (Y) – either literally or figuratively.

# PROMPTS

1. Optimize this sorting function for speed, then benchmark it on sample data and refine it again if needed ( Y / N )
2. Generate unit tests for this function using unittest, run them, and fix any failing cases ( Y / N )
3. Write a regex to validate email addresses, test it against sample inputs, and refine it until all pass ( Y / N )
4. Draft a SQL query for this dataset, execute it, and rewrite it to improve performance if it runs slowly ( Y / N )

## PROMPTS - 2

5. Summarize this function with a docstring, then rewrite the docstring to match Google style guidelines ( Y / N )
6. Implement error handling for this function, run sample inputs through it, and adjust the handling as needed ( Y / N )
7. Write a simple CLI argument parser, test it with example commands, and refine it for invalid inputs ( Y / N )
8. Check this script with flake8 for style violations, then automatically fix and re-check until clean ( Y / N )

## PROMPTS - 3

9. Generate a JSON schema for this data, validate it with test data, and refine the schema with constraints ( Y / N )
10. Produce a logging solution for this script, run it on sample executions, and improve the log messages ( Y / N )

# ANSWERS

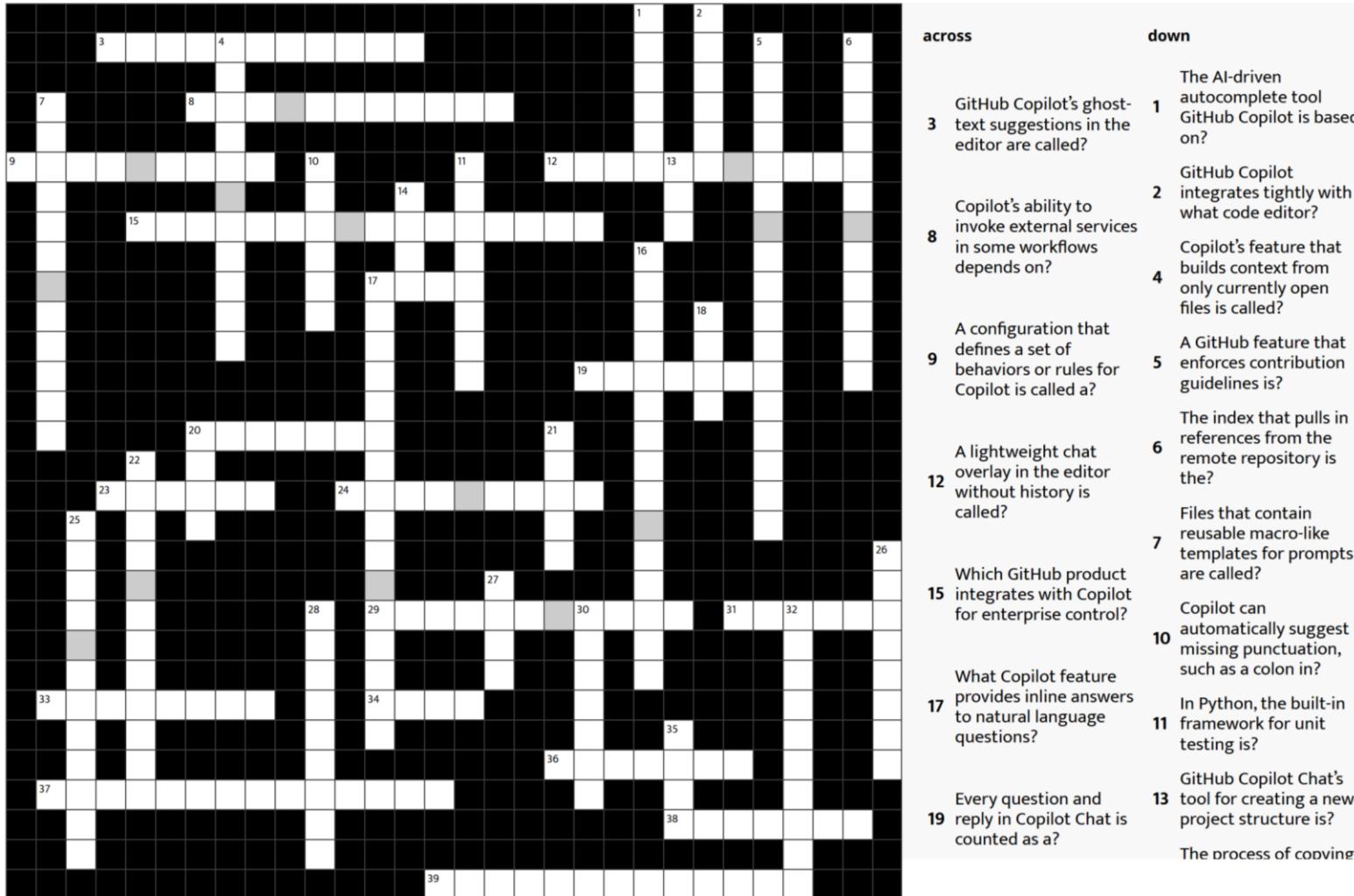
Find the answers in the lab directory in a markdown file: answers.md.

# LAB 8B – PUZZLE



# CROSSWORD PUZZLE - EXERCISE

<https://shorturl.at/SiVxo>



# Lab completed



# AGILE METHODOLOGIES

SCRUM, USE CASES, SPRINT BACKLOG



# INTRODUCTION



This module shows how to turn a simple user story into precise, testable behavior. You'll start with a backlog item, write acceptance criteria, and translate them into Gherkin (Given-When-Then) scenarios so developers, testers, and stakeholders share a clear, executable understanding.

Then you'll use Copilot Chat to accelerate a tight TDD loop: generate test scaffolds from those scenarios, run failing tests, and add just enough code to pass—repeating until the feature is done.

# AGILE METHODOLOGY

Agile methodology is a flexible approach to project management and software development that delivers value in small, usable increments. Instead of long cycles, Agile teams release features frequently, gather feedback, and adapt quickly to change. It emphasizes collaboration, customer involvement, and continuous improvement, making it ideal for fast-moving environments.

Agile roles support teamwork and accountability. The Product Owner prioritizes the backlog and defines value, while the Scrum Master guides Agile practices and removes obstacles. The Development Team is cross-functional and self-organizing, delivering working increments each cycle. Stakeholders provide feedback to keep the product aligned with user and business needs.

# AGILE METHODOLOGY

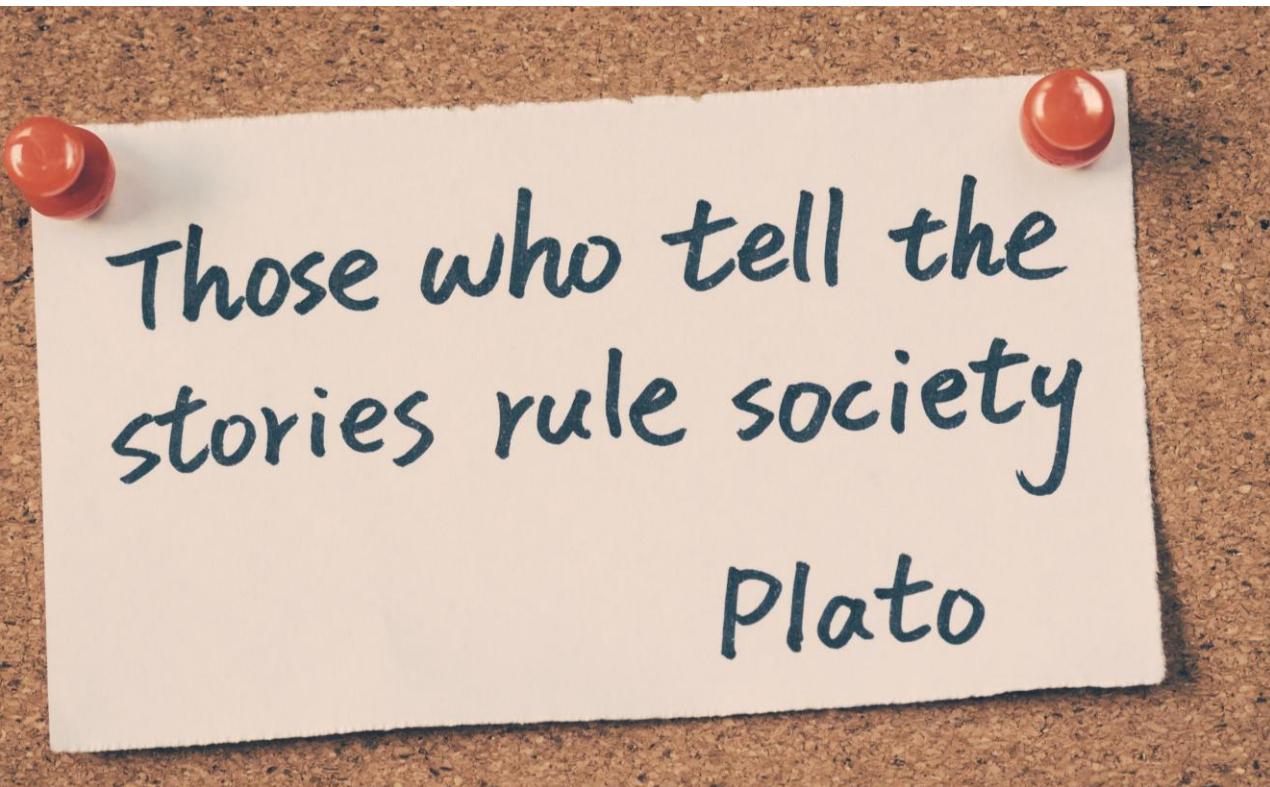
Agile methodology is a flexible approach to project management and software development that delivers value in small, usable increments. Instead of long cycles, Agile teams release features frequently, gather feedback, and adapt quickly to change. It emphasizes collaboration, customer involvement, and continuous improvement, making it ideal for fast-moving environments.

Agile roles support teamwork and accountability. The Product Owner prioritizes the backlog and defines value, while the Scrum Master guides Agile practices and removes obstacles. The Development Team is cross-functional and self-organizing, delivering working increments each cycle. Stakeholders provide feedback to keep the product aligned with user and business needs.

# AGILE PRACTICES

- Scrum – Organizes work into short sprints (2–4 weeks) with defined roles and ceremonies like stand-ups, sprint planning, and retrospectives.
- Kanban – Uses a visual board to track tasks, limit work in progress, and improve flow for continuous delivery.
- Extreme Programming (XP) – Focuses on technical practices like TDD, pair programming, and continuous integration to improve code quality.
- SAFe (Scaled Agile Framework) – Extends Agile principles across large organizations, coordinating multiple teams through alignment, collaboration, and program-level planning.

# USER STORIES



A use case describes how a user interacts with a system to achieve a goal, written from the user's perspective. For example: "As a shopper, I want loyalty discounts applied at checkout so that I pay the correct price."

GWT (Given–When–Then) turns a use case into an actionable task. For the discount example: Given a shopper has a loyalty card, When they check out, Then the discount is applied. GWT makes expected behavior clear and testable for developers, testers, and stakeholders.

# USE STORY



A user story describes how a user interacts with a system to achieve a goal, written from the user's perspective. For example: "As a shopper, I want loyalty discounts applied at checkout so that I pay the correct price."

GWT (Given–When–Then) turns a use case into testable acceptance criteria. For the discount example: Given a shopper has a loyalty card, When they check out, Then the discount is applied. GWT makes expected behavior clear and testable for developers, testers, and stakeholders.

# GHERKIN (GWT)

Gherkin is a plain-text language used in Behavior-Driven Development (BDD) to describe how software should behave in a way that both business stakeholders and developers can understand. It uses a simple structure of Given, When, Then to turn requirements into clear, testable scenarios.

# SCRUM WORKFLOW

SCRUM  
AGILE



PRODUCT OWNER



PRODUCT BACKLOG



SPRINT BACKLOG

RETROSPECT

PLANNING

REVIEW

SPRINT

DEPLOYMENT

IMPLEMENTATION

DAILY SCRUM



# STORY

The story used within this module is about a shopper checking out with a system that applies loyalty discounts. It illustrates how Agile teams take a customer need—ensuring the correct price is charged—and refine it step by step: starting from a backlog item, writing acceptance criteria, translating them into Gherkin scenarios, generating test scaffolds, and finally creating minimal passing code.

# COPILOT AND AGILE

GitHub Copilot helps this workflow by turning backlog items and user stories into acceptance criteria and Gherkin scenarios. During Sprint Planning, once a team commits to a story (like applying loyalty discounts), they refine it into testable conditions. Copilot Chat can draft clear acceptance criteria and map them into Given–When–Then scenarios stored in .feature files. This makes backlog items executable and shows students how Agile artifacts evolve into specifications Copilot can use.

Copilot also generates test scaffolds and minimal code from those criteria. With slash commands like /tests or /explain, students can turn criteria into unit test skeletons, check edge cases, and then add just enough code to pass. Edit mode supports guided edits, while Agent mode handles multi-file updates. This demonstrates test-driven development—moving from stories to tests to working code—with Copilot assisting at each step.

# (1) WORKFLOW

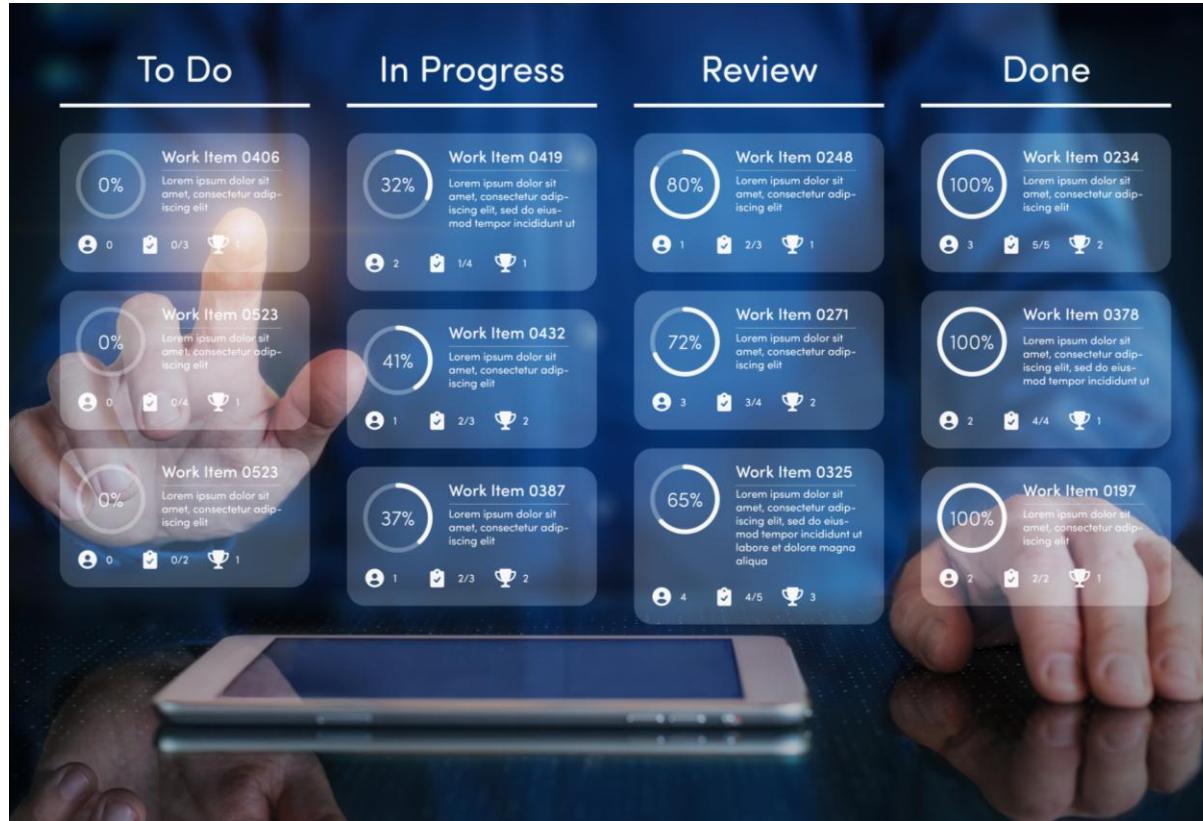
You want to [1] convert ACs and Gherkin (GWT syntax), [2] then build scaffolding as unit tests, and finally create [3] minimal code to make unit test pass. [4] The team can then begin to fully implementing each method iteratively.

Getting started: Paste the use cases into a `use_stories.md`. You will add the file to the context, using `#file`.

Have Copilot create the acceptance criteria with this prompt:

From `#file`, generate 1-3 acceptance criteria for each user story. Format your output as markdown (`user_stories_a.md`), with each story followed by its acceptance criteria list.

# ACCEPTANCE CRITERIA



Acceptance criteria define when a backlog item is “done.” This is important for completing a “done” increment at the end of a Sprint.

You will use GWT to encapsulate your user stories, with “T” being derived from the acceptance criteria.

## (2B) GWT

Convert user stories to GWT. Refer to the input file using #file. You can generate unit tests from GWT using Cucumber. However, Copilot can also be used.

Here is the prompt.

Convert each user story in #file:user\_stories\_a.md into gwt syntax and save to gwt.md.

# (3) FAILING UNIT TESTS???



This step introduces Test-Driven Development (TDD): first you create failing tests (red), then you add just enough code to pass them (green). For this step, you will create the unit tests from JWT,

## (3B) FAILING UNIT TESTS

Generate the unit test from the GWT file. Naturally, you will use Copilot instead of Cucumber. One reason is that Cucumber would require Agent mode. However, if using Copilot, Edit mode is sufficient.

/tests Read #file:gwt.md and create test\_shop.py. One test per GWT; include one boundary and one negative test per acceptance criteria.

# (4) MINIMAL IMPLEMENTATION



This step provides a minimal implementation of each unit, making each test pass. The status of each test will change to green. Making the test pass may require multiple passes (iteratively). For that reason, this should be done in Agent mode.

Importantly, this makes the code executable, which is required for a Sprint Increment.

## (4B) MINIMAL IMPLEMENTATION

Provide a minimal implementation of the test suite for all tests to pass. This must be done iteratively.

Here is the prompt:

In shop.py, minimally implement the code necessary to make the unit tests in #file:test\_shop.py pass. Do this iteratively if required.

# RETROSPECTIVE



This module ties Agile ideas to hands-on execution: you took a real user story—applying loyalty discounts at checkout—then turned it into precise acceptance criteria and Gherkin scenarios before driving a tight TDD loop from failing tests to minimal passing code. Along the way, you saw how Copilot amplifies each step—drafting criteria, mapping them to Given-When-Then, generating test scaffolds, and helping with focused edits—plus when to reach for Edit vs. Agent mode.



# LAB 9 – ACCOUNTS RECEIVABLE



You will work with Accounts Receivable (AR) user stories such as generating invoices, tracking payments, and sending overdue notices. These stories provide the starting point for turning business needs into testable requirements.

# DESCRIPTION



You will use GitHub Copilot to convert user stories into acceptance criteria, then into Given–When–Then (GWT) scenarios. From these, Copilot can generate test scaffold. Finally, you will ask Copilot for a minimal implementation to make tests pass. This end-to-end flow—story → criteria → GWT → tests → code—shows how Agile practices quickly turn AR requirements into working, validated features.

# USER STORIES TO GWT

## 1. Set up

Open invoice\_use\_case.md (user stories).

## 2. Use Copilot to generate Acceptance Criteria

Prompt Copilot to draft clear, testable ACs for each story. Save as invoice\_use\_case\_ac.md.

## 3. Translate ACs into GWT (Given–When–Then)

Prompt Copilot to convert the ACs into concise scenarios. Save as GWT.md.

# CREATE UNIT TESTS

## 4. Generate unit tests from GWT

Prompt Copilot to create Python unittest tests based on GWT.md. Save as test\_accounts\_receivable.py.



# IMPLEMENT AND TEST

## 5. Implement minimal production code

Prompt Copilot to generate the smallest changes in your app module (e.g., accounts\_receivable.py) to make the tests pass, iterating as needed.

## 6. Run the test suite

```
python -m unittest discover -v
```

# Lab completed



# MANAGING GITHUB REPOSITORIES

## COMMIT AND PULL REQUESTS



# INTRODUCTION



The presentation explains how GitHub Copilot Chat goes beyond inline code suggestions to help with repository management. It emphasizes Copilot's role in automating and improving routine developer tasks such as writing commit messages, generating pull request descriptions, suggesting branch names, documenting repositories, and supporting code reviews

# REPOSITORY MANAGEMENT

1. Copilot is more than just inline code completion—it supports repo management.
2. Works in VS Code, JetBrains IDEs, Neovim, and directly on GitHub.com.
3. Uses context from code and project history to suggest:
  - Commit messages and branch names
  - Pull request descriptions and review comments
  - Repository documentation and GitHub Actions
4. Benefits: faster collaboration, fewer mistakes, consistent team practices.

# EXAMPLE

GitHub Copilot prompt:

"Summarize this commit for me."

Copilot Suggestion:

Refactor user authentication logic; extracted JWT validation into separate module for reusability.

Saves time writing context-heavy commit notes while keeping history meaningful.

# COMMITS

Writing clear, consistent commit messages is often neglected.

- Copilot automatically analyzes the diff and proposes meaningful summaries.
- Helps enforce team-wide commit conventions (e.g., Conventional Commits).
- Reduces “bad commits” like update file or fix stuff.
- Especially useful in fast-moving teams where speed but quality is important.

## EXAMPLE

- Changed Code: Added input validation to form handler.
- Copilot Suggestion:

Add input validation for form handler to prevent invalid submissions.

Concise, descriptive, and instantly reusable!

# BEST PRACTICES

- Always review Copilot's suggestion for accuracy.
- Adapt to your team's standards (Conventional Commits, Jira references, etc.).
- Add extra context if change affects multiple systems.
- Copilot = drafting assistant, not a replacement for human judgment.
- Encourages better commit hygiene across the project.

## BEST PRACTICES - EXAMPLE

- Prompt: "Write a commit message using conventional commits style."
- Copilot Output:  
fix(auth): correct JWT expiration handling causing login failures

# PR

1. Copilot generates PR descriptions summarizing key changes.
2. Can include:
  - High-level purpose of the PR
  - List of modified files or features
  - Risk/impact assessment
3. Improves review efficiency by giving reviewers the “big picture.”
4. Avoids poorly written PRs like “Added stuff”

# EXAMPLE

Prompt: "Generate a PR description for these changes."

Copilot Suggestion:

### ### Summary

- Added input validation to user form handler
- Refactored validation into utils module
- Updated unit tests to cover edge cases

### ### Impact

Prevents invalid user data from reaching backend

Well-structured, follows GitHub PR template style.

# IMPROVING CODE REVIEWS

- Copilot explains complex diffs in plain English.
- Can suggest inline comments for readability or maintainability.
- Proposes refactor opportunities before review is even requested.
- Helps newer team members understand senior-level code faster.
- Keeps reviews focused on correctness, not wording.

# EXAMPLE

Code Snippet:

```
if(user.role == 'admin' ||  
    user.permissions.includes('edit')) {  
    allowEdit();  
}
```

Copilot Suggestion for PR comment:

“Consider extracting permission logic into a utility function to improve readability and testability.”

# BRANCH NAME SUGGESTIONS

- Encourages consistent naming (e.g., feature/, bugfix/, chore/).
- Saves time thinking of names during development rush.
- Avoids ad-hoc branches like don-test1 or temp-stuff.
- Consistency improves searchability and automation hooks.

## EXAMPLE

Prompt: "Suggest a branch name for adding payment retries."

Copilot Suggestion:

feature/payment-retry-mechanism

Matches convention: easy for CI/CD pipelines to pick up.

# REPOSITORY DOCUMENTATION

1. Copilot can draft and update:
  - README.md (usage, setup, contribution)
  - Contributing.md and issue/PR templates
  - API usage guides and inline docs
2. Keeps documentation aligned with evolving codebase.
3. Reduces outdated docs, one of the top developer frustrations.

# EXAMPLE

Prompt: "Generate README instructions for running the app."

Copilot Output:

```
## Running the App
```

1. Clone the repository
2. Install dependencies with `npm install`
3. Start the server using `npm run start`
4. Access the app at <http://localhost:3000>

# RETROSPECTIVE



This module demonstrated how GitHub Copilot Chat enhances repository management by supporting commits, pull requests, reviews, and documentation. We saw how Copilot can generate meaningful commit messages, structured PR descriptions, and even suggest improvements during code review.

"Develop a passion for learning."



# Excellent

You have completed the course successfully!

