# AI-04636 VSCode Github CoPilot

*What is a Github Copilot CLI in VSCode? What can I use it for?*

**GitHub Copilot CLI** brings the capabilities of GitHub's Copilot coding agent directly into your command-line terminal.

When used inside **VS Code** (via the integrated terminal or connected external terminal), it bridges the gap between your editor and command line. It runs as a standalone process, allowing you to trigger heavy, agentic coding tasks without freezing or slowing down VS Code's editor UI.

## Key Uses for GitHub Copilot CLI in VS Code

- **Multi-File & Repo-Wide Refactoring:** Execute complex, multi-file edits, codebase migrations, or architectural updates autonomously without managing GUI diff approvals line-by-line.
- **Command Generation & Debugging:** Translate natural language into shell commands (e.g., complex `git`, `docker`, or `find` commands) or diagnose terminal errors and build failures directly in place.
- **Parallel Execution ( `/fleet` Mode):** Run multi-threaded agent sessions to update multiple sub-projects or packages simultaneously in the background while you continue working.
- **Terminal-Native Git Workflows:** Generate commit messages, review pull requests, create branches, and handle Git conflicts via simple prompts.
- **Context Sharing with VS Code:** Right-click files in VS Code to attach them to your active Copilot CLI prompt, or use `@` mentions directly inside the terminal.
- **Custom Tool Extensions (MCP):** Connect Model Context Protocol (MCP) servers to let your terminal agent interact with external databases, APIs, or custom enterprise pipelines.

## VS Code Agent vs. Copilot CLI

| Feature | VS Code Copilot Chat / Inline | GitHub Copilot CLI |
|---|---|---|
| **Primary Focus** | Code completions, single-file edits, inline explanations | Heavy multi-file tasks, terminal commands, autonomous agent loops |
| **Execution** | Shared within the VS Code UI process | Decoupled background CLI process |
| **User Interaction** | Visual GUI diffs, click-to-accept edits | Text-driven, terminal-native output, reviewed via Git diff |

*Show me step-by-step how to install GitHub Copilot CLI and connect it with VS Code.*

## Prerequisites

- Active **GitHub Copilot** subscription.
- **Node.js** (v18 or higher) installed on your machine.
- **VS Code** installed with an active workspace open.

## Step 1: Install the GitHub Copilot CLI

Open your terminal (macOS/Linux) or PowerShell/Command Prompt (Windows) and run:

Bash

```
npm install -g @github/copilot
```

*(Alternative package managers: Homebrew* `brew install copilot-cli` *or WinGet* `winget install GitHub.Copilot` *)*

## Step 2: Authenticate with GitHub

1. Launch the CLI by running:

   Bash

   ```
   copilot
   ```

2. If this is your first run, trigger the login command:

   Plaintext

   ```
   /login
   ```

3. Copy the one-time device code displayed in your terminal, open the provided browser URL (e.g., `[https://github.com/login/device](https://github.com/login/device)` ), paste the code, and click **Authorize**.

## Step 3: Connect Copilot CLI to VS Code

**Method A: Automatic Connection (Inside VS Code)**

1. Open VS Code and open your project folder ( `File > Open Folder...` ).
2. Open VS Code's integrated terminal ( `Ctrl+\`` or Cmd+`` ).
3. Run `copilot` . Copilot CLI will automatically detect the open VS Code workspace and link to it.

**Method B: Manual Connection via `/ide` Command**

If you launched Copilot CLI in a standalone external terminal outside of VS Code:

1. Make sure your target project is open in VS Code in **Trusted Mode**.
2. Inside your active Copilot CLI terminal session, type:

   Plaintext

   ```
   /ide
   ```

3. Select your open VS Code workspace from the interactive menu to attach the CLI session directly to your editor instance.

## Step 4: Verify the Setup

Once connected, test context-sharing between the CLI and VS Code:

- Run a prompt like:

  Plaintext

  ```
  Explain the structure of this project and show me active git changes.
  ```

- Use the `/diff` command inside the CLI session to view and review code modifications applied directly to your open VS Code files.

*List top 22 Github Copilot Prompts*

These top 22 GitHub Copilot prompts are organized across the core phases of the software development lifecycle:

**Code Generation & Architecture**

1. **Planning before coding:** "Propose a step-by-step implementation plan for adding [FEATURE] to this project using [TECH STACK]. Do not write code yet—ask me to confirm first."

2. **Boilerplate/Scaffolding:** "Create a production-ready [LANGUAGE/FRAMEWORK] module that implements [FEATURE]. Include strict type safety, input validation, and standard project structure."

3. **Database Schema / Migration:** "Write a [DATABASE, e.g., PostgreSQL] migration script to add [TABLE/COLUMNS] with foreign keys, indexes for performance, and proper constraint checks."

4. **Cross-Language Translation:** "Convert this [SOURCE LANGUAGE] function to [TARGET LANGUAGE]. Maintain functional parity, idiomatic style, and highlight any language-specific nuances."

**Debugging, Testing & Error Handling**

5. **Diagnose Runtime Error:** "I am seeing this error in my terminal: [PASTE ERROR]. Explain the root cause and provide the minimal diff required to fix it." 6. **Regression Test Generation:** "Write a unit test using [TEST FRAMEWORK, e.g., Jest/PyTest] for a bug where [EDGE CASE] causes unexpected failure. Ensure the test fails now and passes after the fix." 7. **Edge-Case Coverage:** "Analyze @filename and generate unit tests covering zero/null inputs, boundary conditions, and invalid data types." 8. **Mocking External APIs:** "Write unit tests for @filename by mocking all external network calls to [API NAME] using [MOCK LIBRARY]."

**Refactoring & Code Quality**

9. **Zero-Side-Effect Refactor:** "Goal: Refactor @functionName for readability and testability. Constraints: Do not change the function signature, output contracts, or public API."

10. **Hot Path Optimization:** "Analyze @filename for algorithmic inefficiencies. Suggest optimizations to reduce time complexity without introducing new dependencies."

11. **Design Pattern Application:** "Refactor this block of conditional logic to use the [DESIGN PATTERN, e.g., Strategy / Factory / State] pattern to improve extensibility."

12. **DRY Abstraction:** "Identify duplicated logic across these selected files and extract it into a reusable utility module."

**Security & Review**

13. **Security Audit:** "Review @filename for common security vulnerabilities (OWASP Top 10, SQL injection, XSS, unhandled secrets). Suggest safer alternatives."

14. **PR / Commit Summary:** "Generate a clear, conventional commit message and pull request summary based on my active git diff."

15. **Code Standard Enforcement:** "Review this file against [LANGUAGE/FRAMEWORK] idiomatic best practices and suggest improvements for consistency and maintainability."

**Terminal, CLI & DevOps**

16. **Complex Shell Command:** "Give me a single terminal command for [OS, e.g., Linux/macOS] to recursively find [FILE TYPE] modified in the last 7 days and [ACTION]."

17. **Docker / Containerization:** "Draft a multi-stage Dockerfile for this project optimized for minimal image size, non-root execution, and cached layer builds."

18. **CI/CD Pipeline:** "Create a GitHub Actions workflow that runs linting, unit tests, and builds on push to main or pull requests."

**Documentation & Knowledge Transfer**

19. **Docstring Generation:** "Write comprehensive inline documentation/docstrings for @filename following [STANDARD, e.g., JSDoc / Google Python style]."

20. **Legacy Code Breakdown:** "Explain what @filename does step-by-step as if I am onboarding to this codebase for the first time. Highlight key dependencies and risks."

21. **API Endpoint Documentation:** "Generate OpenAPI 3.0 (Swagger) spec documentation based on the route handlers in @filename ."

22. **README Generation:** "Draft a clean README.md for this repository including prerequisites, quickstart instructions, environment variables, and project structure."

*What are the core prompt engineering strategies to get the best responses from GitHub Copilot?*

To consistently get accurate, high-quality responses from GitHub Copilot, follow these seven core prompt engineering strategies:

## 1. Leverage "Neighboring Tabs" & Open Context

Copilot reads active files open in your editor to infer intent.

- Keep related schema definitions, types, or utility modules open in adjacent tabs.
- In Copilot Chat, explicitly control context by using `@workspace` to search the repo or `#file` to attach specific files directly.

## 2. Follow the 3 S's: Single, Specific, Short

Vague prompts lead to generic solutions. Break down intent into precise steps:

- **Poor:** "*Build a user authentication system.*"
- **Strong:** "*Write an Express.js middleware function that validates JWT tokens in the* `Authorization` *header, handles expired tokens with a 401 response, and attaches the decoded payload to* `req.user`."

## 3. Give Examples (Few-Shot Prompting)

Copilot mirrors existing code patterns. Show it what you want before asking for new output:

TypeScript

```
// Input: "john.doe@example.com" -> Output: "j***e@example.com"
// Input: "alex@test.org" -> Output: "a**x@test.org"
// Write a function `maskEmail(email: string): string` following this pattern:
```

## 4. Provide Top-Level & Function-Level Intent

State the overall goal at the top of the file before generating code.

Python

```
# Module: Weather Data Processing pipeline
# Goal: Ingest JSON payload, strip null fields, and calculate 7-day rolling average.

def process_daily_metrics(data: list[dict]) -> list[dict]:
    # Filter out entries where temperature or humidity is None
```

## 5. Define Constraints and Architectural Rules

Explicitly state what Copilot **must** or **must not** do:

- Mention specific library versions ("*Use React 18 functional components with hooks*").
- Define boundary limits ("*Do not import external dependencies. Handle edge cases where input array is empty.*").
- Specify performance bounds ("*Implement using iterative loops rather than recursion to avoid stack overflow.*").

## 6. Use Custom Instruction Files ( `.github/copilot-instructions.md` )

Instead of repeating standard preferences in every prompt, create a `.github/copilot-instructions.md` file in your repository. Copilot automatically reads this for project-wide rules:

- Preferred testing framework (e.g., PyTest vs. Unittest).
- Coding conventions (e.g., tabs vs. spaces, strict typing enforcement, error handling patterns).

## 7. Iterate & Refine via Chat

Treat Copilot's output as a draft. If the first response isn't quite right, guide it sequentially:

1. "*Plan the architecture first without writing code.*"
2. "*Now generate the core logic.*"
3. "*Add comprehensive docstrings and handle null-pointer edge cases.*"