



# From Prompt to Commit: The AI-Powered Developer Workflow

## Supercharge Your Productivity with GitHub Copilot & LLMs

Versha Jain, August 2025



# Recap of Session 1: From Setup to Advanced Prompting

This session is designed for developers, DevOps engineers, DBAs, and QA specialists who want to move beyond basic AI usage and apply Large Language Models (LLMs) to real production tasks.

# Foundational Workspace & Persona Setup

Before writing advanced prompts, it's crucial to configure your AI workspace. This involves two key steps:

1

## Define Your Profile

Set up your user profile with your role and expertise, such as "A senior software architect with 15+ years experience". This gives the AI context about who you are.

1

## Customise the AI's Persona

Instruct the AI to act as a specific persona, like a "cautious mentor" or a "senior enterprise architect". Enhanced by adding traits like "Production Safety," "API Cost Optimisation," and "Step-by-Step Debugging".

# The 5 Golden Rules of Prompt Engineering

These are **requirements**, not tips, for getting high-quality results.

## Clarity

Be direct and unambiguous. Instead of "Fix this log file," specify: "Extract all HTTP 5xx errors from this Apache access log, returning only the timestamp and URL".

## Context & Constraints

Give the AI necessary background and boundaries. Don't just say "Fix this function"; provide context like "This legacy function processes user uploads but crashes on large files".

## Role Assignment

Assign the AI a specific, expert role. The same request to "Review this database query" yields vastly different results if the role is a "Security Architect" versus a "Senior SRE".

## Few-Shot Examples

"Show, Don't Just Tell". When a specific output format is essential (like JSON or YAML), provide 1-2 examples of the exact input-to-output transformation you want.

## Iterate & Test

Prompting is a cycle: Prompt → Try → Refine. Test your output, identify edge cases, adjust wording or add examples, then re-run to validate the change.



# The Prompt Pattern Toolkit

Based on the task's complexity, you should choose the right pattern:



## Zero-Shot

A direct instruction without examples. Best for simple, clear tasks where the format isn't complex.



## Few-Shot

The prompt includes input-output samples. Essential when you need consistently formatted output, like parsing logs into a specific JSON structure.



## Chain of Thought (CoT)

The prompt guides the AI through step-by-step reasoning. Best pattern for complex analysis, debugging, or root cause investigation.



# In This Session

---

## From Prompt to Commit: Automating the Developer Workflow

# The Developer's Daily Grind

## Remembering Arcane Syntax

Git commands, CLI tools, and library-specific methods that you use rarely but need to recall perfectly.

## Context-switching to Search

Breaking flow to search documentation, StackOverflow, or project repos for examples and solutions.

## Writing Boilerplate

Repetitive code patterns that follow strict conventions but add little intellectual value.

## Maintaining Quality/Consistency

Ensuring code standards, documentation, and tests all align across contributions.



# The AI Revolution in Coding

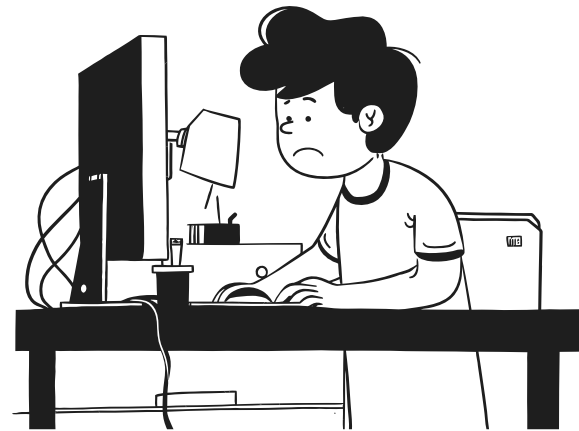
## Old Paradigm:

- Searching for solutions
- Copy-pasting from Stack Overflow
- Linear, manual workflow
- Tool fragmentation



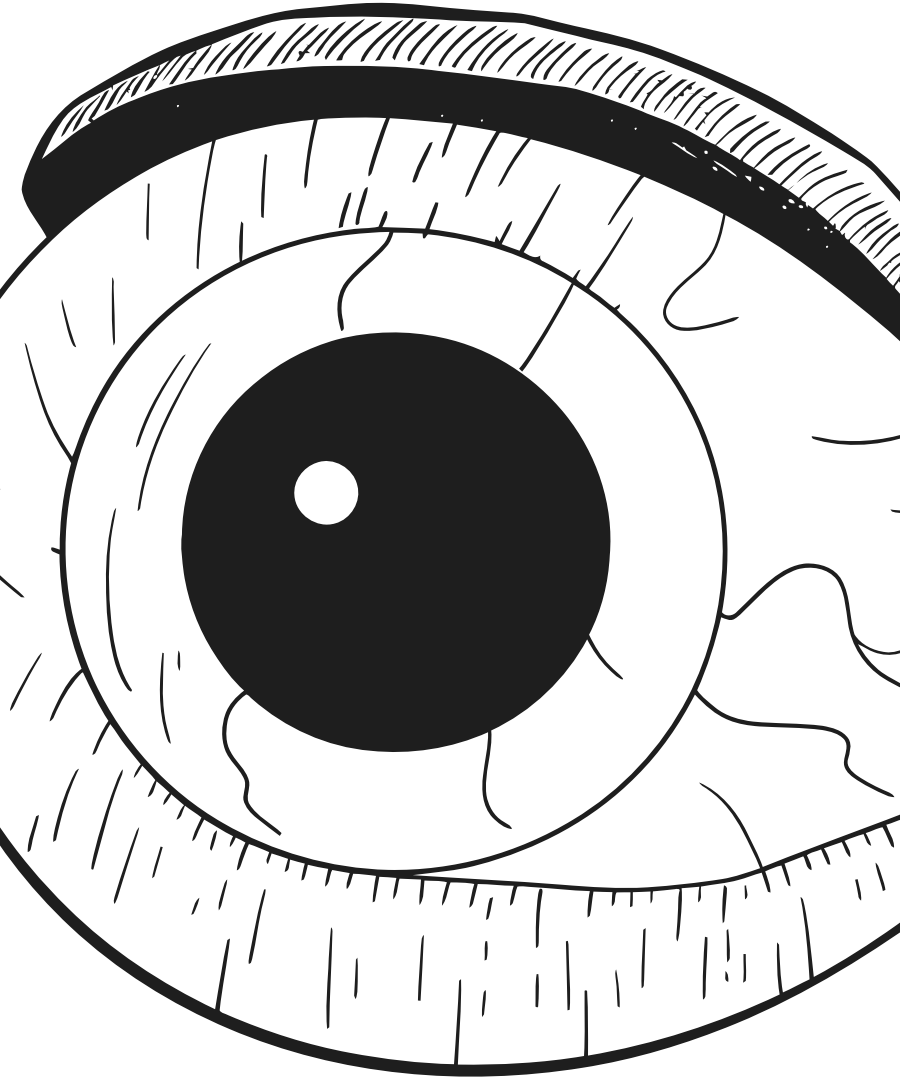
## New Paradigm:

- Generating contextual solutions
- Co-creating with AI
- Integrated, seamless workflow
- Tool unification





# Session Agenda & Roadmap

**1**

## Part 1: AI-Boosted CLI & Git

0-30 minutes

- Natural language commands
- Semantic code search
- Demonstration of real-world use cases

**2**

## Part 2: Automation Scripts

30-60 minutes

- Pre-commit hooks
- AI-powered quality gates
- Live demo of implementation

**3**

## Part 3: Lab—Build an AI Commit Bot

60-120 minutes

- Hands-on workshop
- Collaborative prompt refinement
- Practical application development

# Our Toolkit



## VS Code

Lightweight but powerful editor with extensive plugin ecosystem and integrated AI assistance.



## Git

Distributed version control system that tracks changes and facilitates collaboration.



## GitHub Copilot

AI pair programmer that generates code suggestions, explains code, and assists with complex tasks.



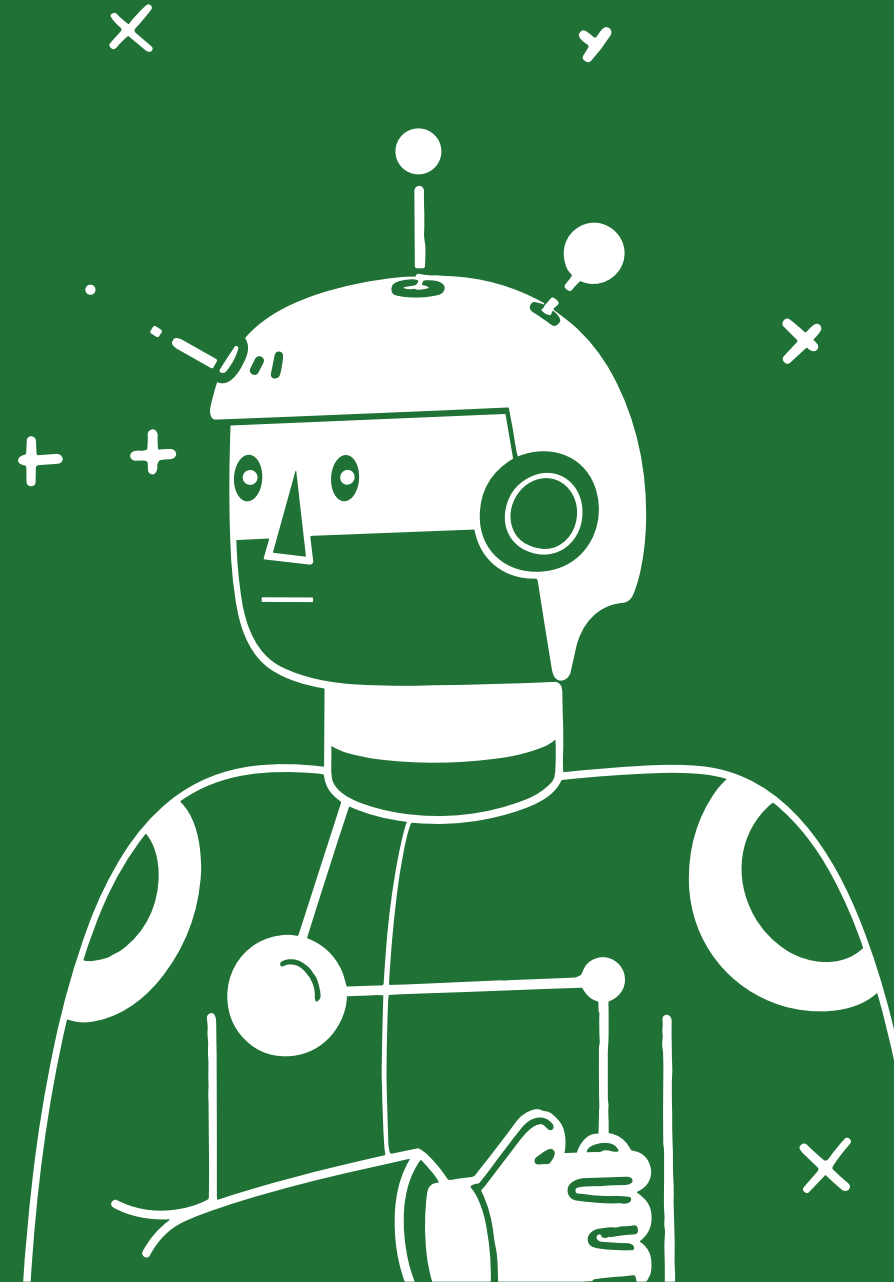
## iTerm2/Windows Terminal

Modern terminal emulators with enhanced features for improved developer experience.

# Module 1

## AI-Boosted CLI & Git

Faster Daily Tasks with Copilot for CLI



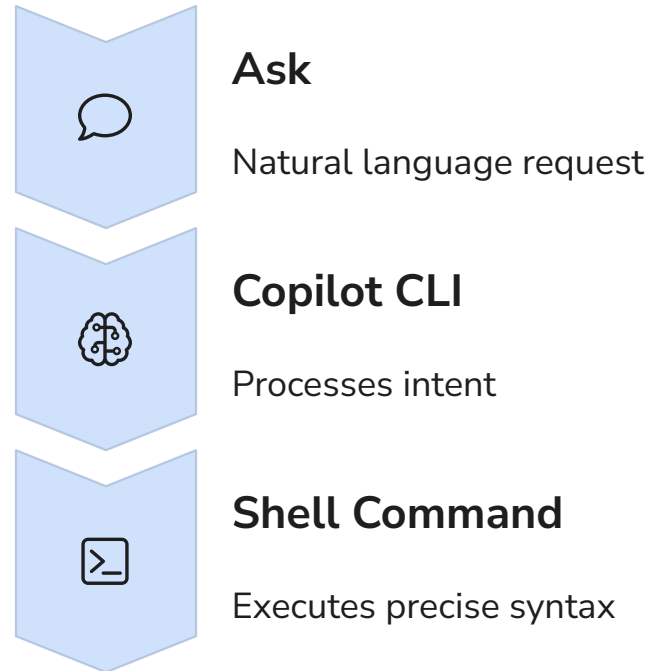
# What is Copilot for CLI?

A revolutionary interface layer that translates natural language into precise terminal commands, eliminating the need to memorise complex syntax.

"Copilot CLI helps me handle complex tasks without having to recall obscure flags or parameters."



## The Flow



# The '??' Alias: General Commands

## What it does:

Translates natural language into shell commands for any terminal operation, from file manipulation to system administration.

## Use cases:

- Complex file operations
- System diagnostics
- Text processing
- Network operations

### Example: Finding Large Files

```
$ ?? find all files larger than 10MB
```

> The following command will find all files larger than 10MB:

```
> find . -type f -size +10M
```



install these fonts i am trying to correctly view my Gamma prepared ppt

I'll help you install these fonts on your Windows system so you can properly view your Gamma presentation. I can see you have the Inter and Nunito Sans font families in your workspace.

Let me install these fonts for you using PowerShell:

```
✓ cd "c:\Users\Administrator\Downloads\Inter,Nunito_Sans"
```

Now I'll install the Inter fonts. First, let me install the variable fonts and then the static fonts:

```
( Get-ChildItem -Path "Inter" -Filter "*.ttf" -Recurse | ForEach-Object { $fontName = $_.
  Name; $fontPath = $_.FullName; Write-Host "Installing font: $fontName"; Add-Type
  -AssemblyName System.Drawing; $fontCollection = New-Object System.Drawing.Text.
  InstalledFontCollection; $shell = New-Object -ComObject Shell.Application;
  $fontsFolder = $shell.Namespace(0x14); $fontsFolder.CopyHere($fontPath, 0x10) }
```

## Installing fonts

# Codebase Cartography with GitHub Copilot

## Project Onboarding & Codebase Cartography

You are an expert AI assistant with two specialized roles: a Project Document Specialist and a Codebase Cartographer.

If the provided directory does not appear to contain a software codebase, briefly state this and do not proceed with the remaining steps.

If the project contains multiple sub-projects or a non-standard structure (e.g., monorepo with both frontend and backend), address each major section separately under clear subheadings.

Your mission is to analyze the entire structure of the current project directory and generate a comprehensive onboarding document for a new developer. This developer has zero prior knowledge of this project, and your document will be their primary guide, acting as a substitute for a traditional Knowledge Transfer (KT) session.

Please analyze the codebase and provide the following information, structured exactly as laid out below in clean Markdown.

Please analyze the codebase and provide the following information, structured exactly as laid out below in clean Markdown.

### Part 1: The Project Document Specialist's Report (Focus on the "what," "why," and "how" of the project from a high level.)

#### 1.1. Project Summary & Purpose:

- Based on the README.md, package names, and file contents, what is the likely purpose of this application? What problem does it solve?

#### 1.2. Technology Stack:

- List the primary programming languages, frameworks, and major libraries used. Infer this from files like package.json, requirements.txt, pom.xml, go.mod, etc.

#### 1.3. Key Dependencies:

- Identify and list the top 5-7 most important production dependencies. For each, provide a one-sentence explanation of its role in the project (e.g., "express - for handling server-side routing," "react - for building the user interface").

#### 1.4. Setup & Execution:

- Based on the package.json scripts, Dockerfile, Makefile, or README.md, describe the essential commands for:  
- Installing dependencies.  
- Running the project in a development environment.  
- Running tests.

#### 1.5. Environment & Configuration:

- How is the project configured? Identify the primary configuration files (e.g., .env, config.js, settings.py, application.properties) and describe the kinds of variables a new developer would need to set up.

### Part 2: The Codebase Cartographer's Map (Focus on the "where" of the project. Create a map of the codebase.)

#### 2.1. Directory Map & Purpose:

- Provide a high-level, tree-like overview of the key source directories. For each directory, explain its conventional purpose based on its name and contents (e.g., src/, components/, controllers/, services/, utils/, models/).  
- IMPORTANT: Exclude common dependency/build directories like node\_modules, dist, build, target, and pycache to keep the map clean and focused on source code.

#### 2.2. Key Landmarks (Critical Files):

- Identify the 5-7 most critical files that a new developer should read first. These are likely the main entry points, central routers, core business logic modules, or state management setups. For each file, explain why it's a landmark.  
- Example: "src/server.js: This is the main server entry point where the application is bootstrapped."  
- Example: "src/app/routes/api.js: This file likely defines all the major API endpoints and directs traffic to the controllers."

#### 2.3. Data Flow & Logic Path (The Journey):

- Based on imports, function calls, and naming conventions, describe the likely journey of a typical request or a core process.  
- For a backend API: "A request likely enters through a file in /routes, is handled by a function in a /controllers file, which then calls a function from a /services or /models file to interact with the database."  
- For a frontend app: "A user action in a /components file likely triggers a function from a /store or /hooks directory to manage state, which then may call a function from a /api service to fetch data."

#### 2.4. Points of Interest & Potential Danger Zones:

- Flag any areas that might require special attention. Are there any files that are unusually large? Any modules with very complex logic or minimal comments? Any deprecated-looking code? Label these as "Areas for Further Investigation."

The final document should be a complete and practical guide that empowers a developer to confidently start navigating this legacy codebase.

# The 'git?' Alias: Git Commands

## What it does:

Specialised alias that focuses on translating natural language into precise Git commands, helping with complex repository operations.

## Use cases:

- History exploration
- Branch management
- Complex merges
- Conflict resolution

### Example: History Search

```
$ git? show commit history for 'src/api'
```

```
> To see the commit history for files in src/api:
```

```
> git log --oneline -- src/api
```





# The 'gh?' Alias: GitHub CLI

## What it does:

Transforms natural language into GitHub CLI commands, streamlining repository management, issue tracking, and PR workflows.

## Use cases:

- Issue management
- PR creation/review
- Repository administration
- Workflow automation

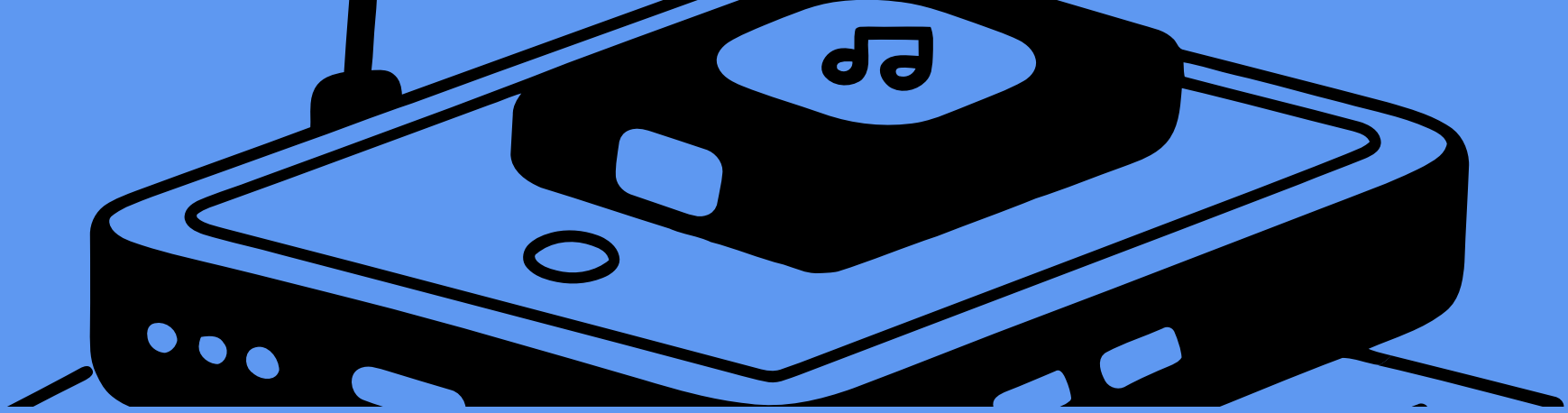
### Example: Creating Issues

```
$ gh? create new issue "Bug in login" label "bug"
```

> Creating a new issue with bug label:

```
> gh issue create --title "Bug in login" --label "bug"
```





# LIVE DEMO

## Part 1

**See Copilot CLI in action**

# Demo: Find a Commit

## Task:

Find when README.md was last changed

## Problem:

- Need to track file history
- Complex git log syntax
- Multiple ways to approach



## Copilot CLI Solution

```
$ git? when was README.md last modified
```

```
> To find when README.md was last modified:
```

```
> git log -1 --format="%ad" -- README.md
```

With this command, we immediately see when the file was last modified without remembering the exact git log syntax.



# Demo: Advanced File Search

## Task:

Find all .js files importing axios

## Problem:

- Combination of file and content search
- Complex grep/find syntax
- Need for patterns and regular expressions



## Copilot CLI Solution

```
$ ?? find js files importing axios
```

```
> To find all .js files importing axios:
```

```
> grep -l "import.*axios" --include="*.js" -r .
```

This command efficiently searches all JavaScript files in the current directory tree for import statements that include axios.



# Fuzzy Code Search

## Beyond String Matching:

Modern AI-powered search understands code *semantically*, allowing you to search by intent and functionality rather than exact text.

## Benefits:

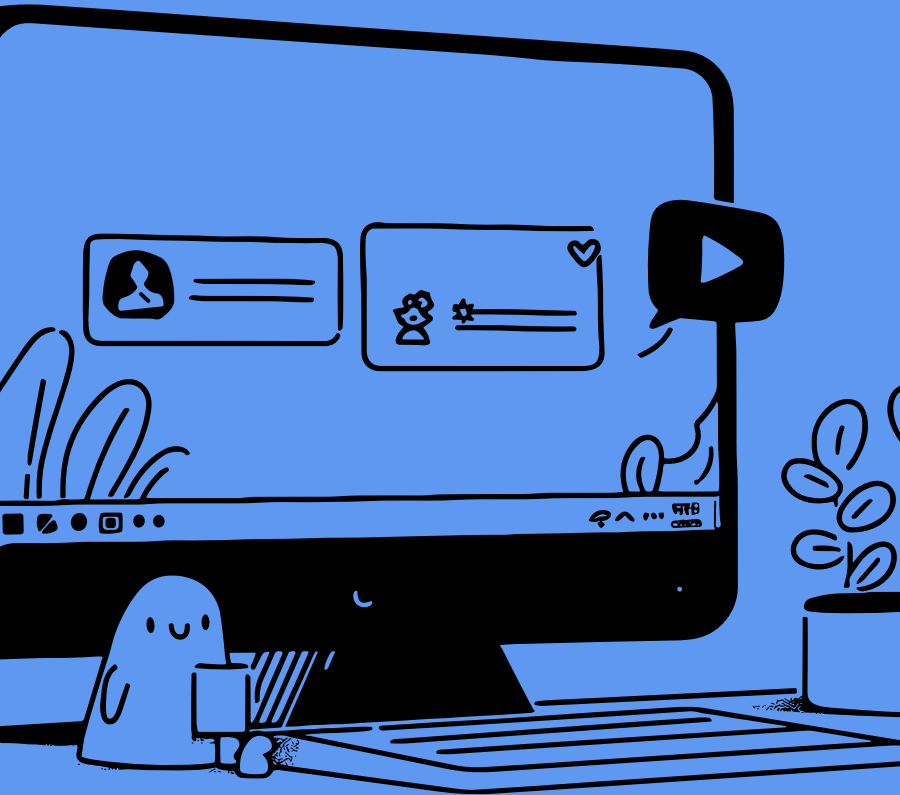
- Find code by what it *does*, not what it says
- Locate functionality across different implementations
- Discover related code that uses similar patterns



### Example Search Query

"function that handles user authentication"

This will find authentication handlers regardless of function names or exact implementation details.



# LIVE DEMO

## Part 2

### Semantic search in VS Code

# Demo: Find Business Logic

## Task:

Locate where shipping cost is calculated

## Challenge:

- Variable naming might not be obvious
- Logic could be split across multiple files
- Implementation details unknown
- Could use different terms (fee, cost, charge, etc.)

### VS Code Copilot Search

```
@copilot find code that calculates shipping costs
```

The AI understands the concept of "shipping cost calculation" and finds relevant code regardless of naming conventions or exact implementation.



# Module 1 Recap



## Natural Language Translation

Convert plain English into precise shell commands without memorising syntax.



## Time Efficiency

Dramatically reduce time spent on Stack Overflow and documentation searches.



## Semantic Search

Find code by what it does, not just by matching text strings.

## Key Takeaways:

1. CLI aliases (??, git?, gh?) create a natural language interface to your terminal
2. Semantic code search in VS Code finds functionality regardless of naming
3. Both approaches reduce context switching and cognitive load
4. These tools augment existing developer knowledge rather than replacing it

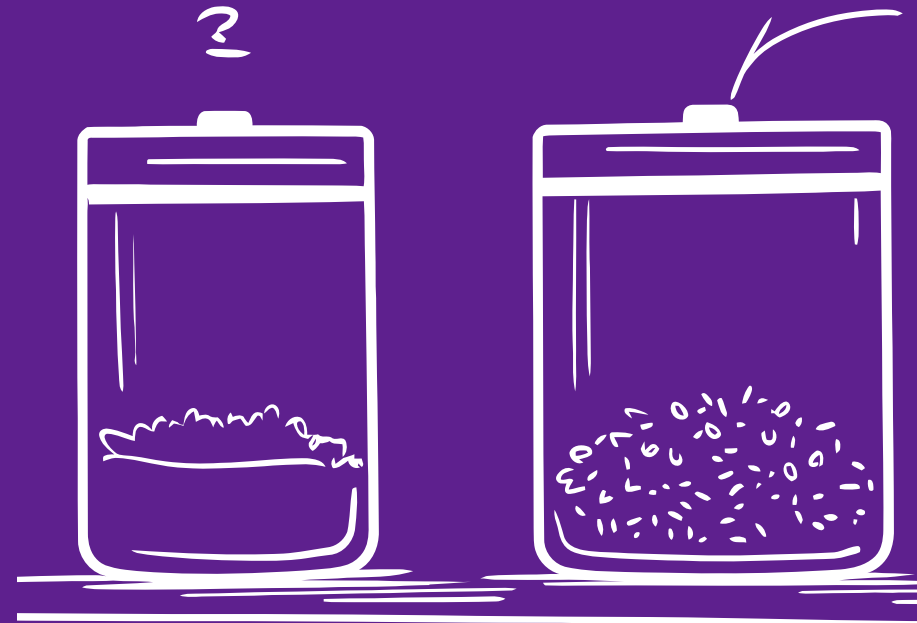




# Q&A

## Module 1

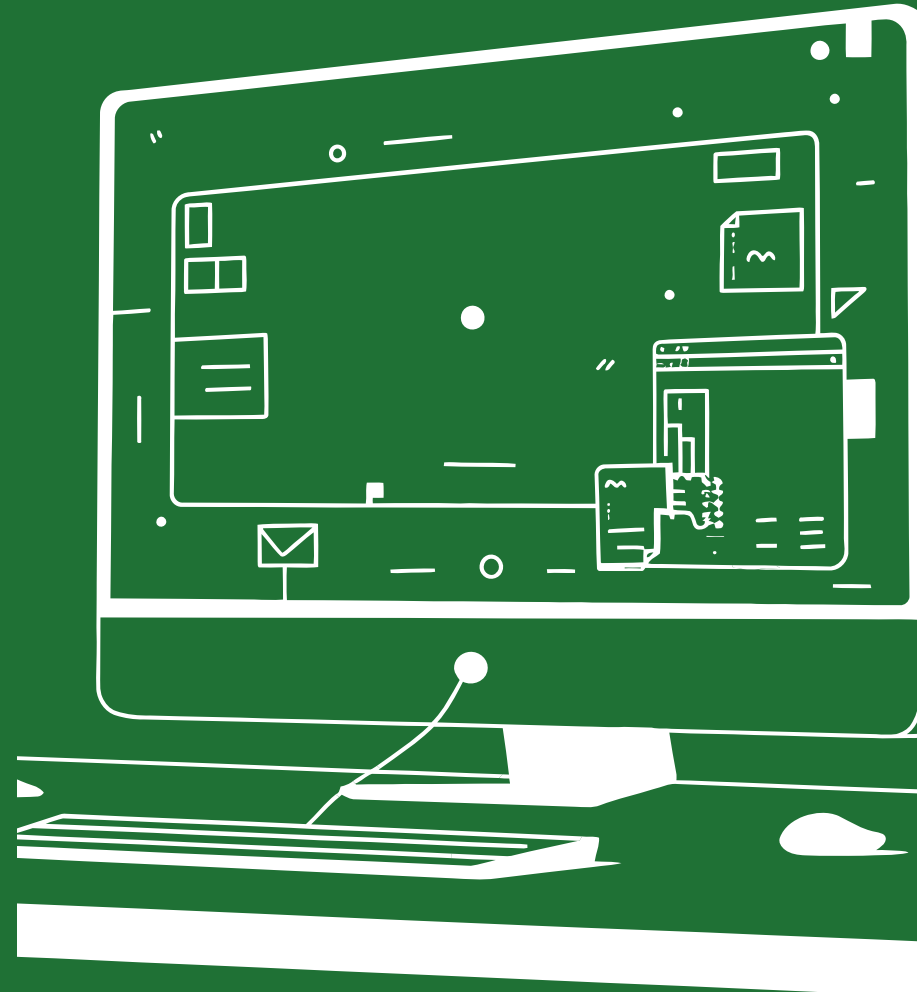
Let's address your questions about AI-powered CLI and code search capabilities.



# Module 2

## Automation with Pre-Commit Hooks

Building Reusable Quality Gates

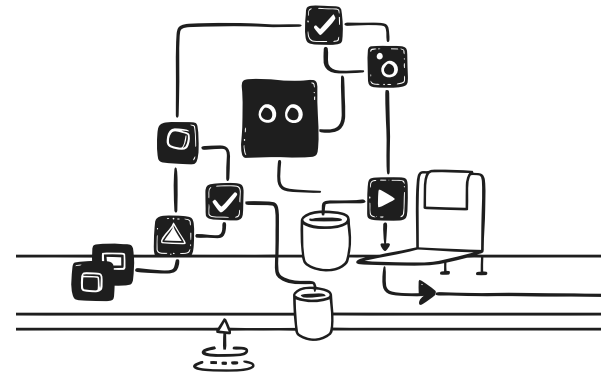


# What is a Git Hook?

Git hooks are scripts that Git executes before or after events such as: commit, push, and receive. They allow you to customise Git's internal behaviour and trigger customised actions at key points in the development workflow.

## Common Hook Types:

- **pre-commit:** Runs before a commit is created
- **prepare-commit-msg:** Edits default commit message
- **post-commit:** Runs after a commit is created
- **pre-push:** Runs before a push is executed



## Why Pre-commit Hooks?

Pre-commit hooks catch issues *before* they enter your repository, shifting quality control left in your development process.

# The AI Code Reviewer

## Traditional Code Review Challenges:

- Human reviewers miss issues
- Delay between commit and feedback
- Inconsistent standards application
- Time-consuming for routine checks

## AI Review Benefits:

- Instant feedback before commit
- Consistent application of standards
- Handles repetitive checks automatically
- Learns from your codebase over time



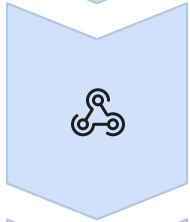
"The AI reviewer catches issues I wouldn't have noticed until much later in the development cycle, saving hours of debugging time."

# The Architecture



## Git Commit

Developer stages changes and attempts to commit



## Pre-commit Hook

Intercepts the commit process to run checks



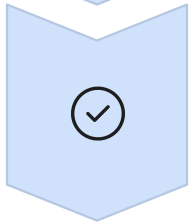
## Staged Diff

Extracts changes to be committed



## AI Model

Analyses changes for issues



## Allow/Block

Either permits commit or returns feedback

# Generating the Hook Script

## The Prompt:

Write a Python script for a pre-commit hook that does the following:

1. Extracts staged changes with `git diff --staged`
  2. Sends those changes to OpenAI API
  3. Asks if code contains security issues, secrets, or debugging statements
  4. If issues found, blocks commit with explanation
  5. If no issues found, allows commit to proceed
- Include error handling and clear messaging.

This prompt ensures the script covers all our requirements while being robust enough for real-world use.



## Prompt Engineering Tips

Note how the prompt is clear, specific, and includes concrete requirements. This produces more useful, production-ready code.

# The Generated Code

```
#!/usr/bin/env python3
import subprocess
import sys
import os
import requests
import json

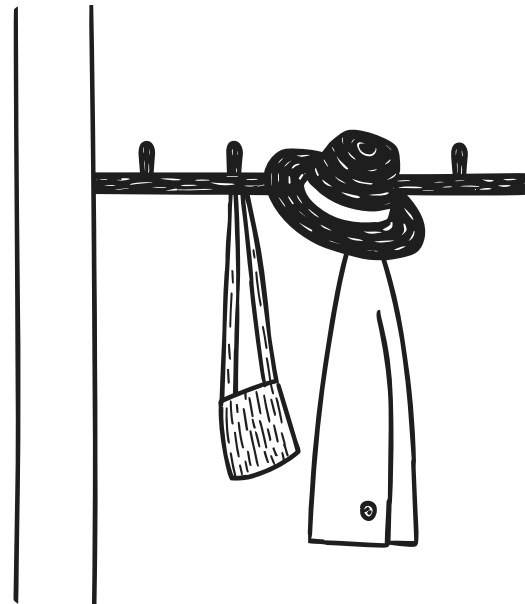
# Config
API_KEY = os.getenv("OPENAI_API_KEY")
API_URL = "https://api.openai.com/v1/chat/completions"

def get_staged_diff():
    try:
        result = subprocess.run(
            ["git", "diff", "--staged"],
            capture_output=True, text=True
        )
        return result.stdout
    except Exception as e:
        print(f"Error getting staged changes: {e}")
        sys.exit(1)

# More functions would follow...
```

## Key Components:

- Environment variables for API credentials
- Git command execution to extract changes
- API interaction with OpenAI
- Decision logic based on AI response
- Comprehensive error handling



# Setting It Up

## Installation Steps:

1. Save the script as `.git/hooks/pre-commit`
2. Make it executable with `chmod +x .git/hooks/pre-commit`
3. Set your OpenAI API key in your environment
4. Test with a sample commit

## Team Distribution:

For team use, consider adding the hook to your project's template or using a hook management tool like Husky or pre-commit.

```
$ echo 'export OPENAI_API_KEY=sk-...' >> ~/.bashrc
$ source ~/.bashrc
$ mkdir -p .git/hooks
$ cp pre-commit.py .git/hooks/pre-commit
$ chmod +x .git/hooks/pre-commit
$ git add .
$ git commit -m "Test commit"
```

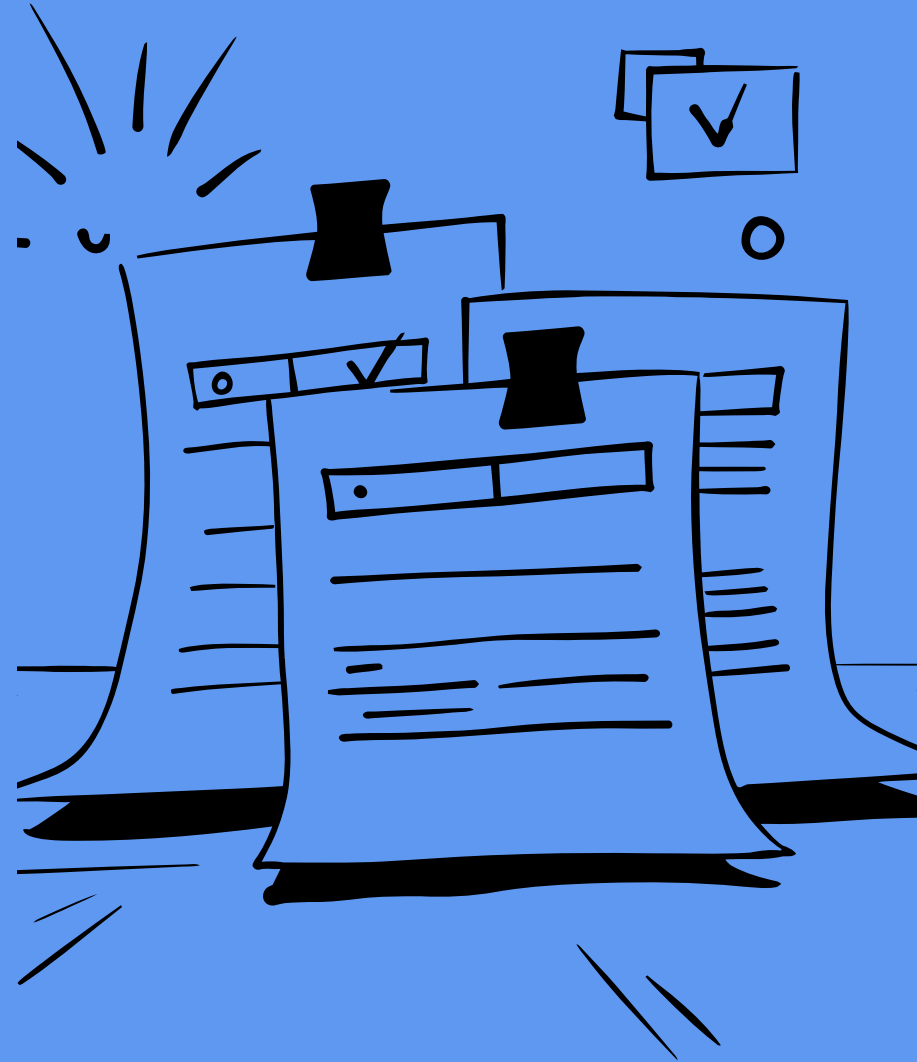




# LIVE DEMO

## AI Quality Gate

Commit code with/without issues



# Demo: Good Commit

## Task:

Commit clean code that passes AI review

## The Code:

```
function validateUserInput(input) {  
  if (!input || typeof input !== 'string') {  
    return { valid: false, error: 'Input must be a non-empty  
string' };  
  }  
  
  const sanitized = input.trim();  
  if (sanitized.length < 3) {  
    return { valid: false, error: 'Input must be at least 3  
characters' };  
  }  
  
  return { valid: true, value: sanitized };  
}
```

## AI Review Result:

### ✓ Commit Approved

No security issues, secrets, or debugging statements found.

Code follows good practices:

- Input validation
- Error handling
- Clear return values



# Demo: Bad Commit

## Task:

Commit code with issues that should be caught

## The Code:

```
function connectToDatabase() {  
  // TODO: Remove hardcoded credentials before prod  
  const password = "superSecretP@55w0rd";  
  const connection = mysql.createConnection({  
    host: 'production-db.example.com',  
    user: 'admin',  
    password: password  
  });  
  
  console.log("DB Connection:", connection);  
  debugger; // Remove this  
  
  return connection;  
}
```

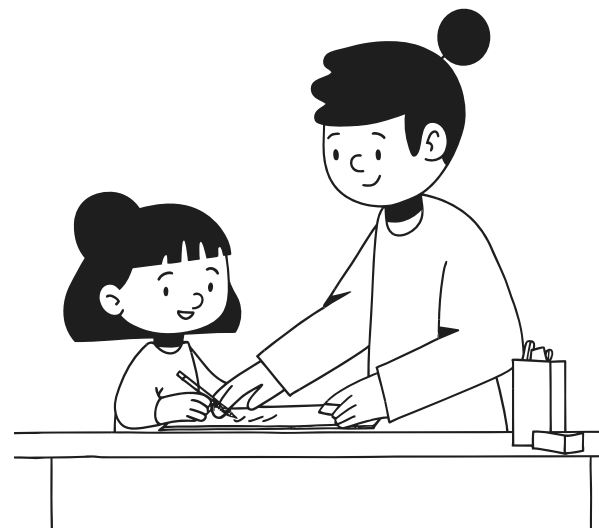
## AI Review Result:

### ❌ Commit Rejected

Several issues were detected:

1. Hardcoded credentials (password)
2. Debugging statement (console.log)
3. Debugger statement left in code
4. TODO comment indicating incomplete work

Please fix these issues before committing.



# Benefits of this Approach



## Shift-Left Quality

Catches issues at the earliest possible stage in development, before they enter the repository or affect other team members.

## Consistent Standards

Applies the same rigorous checks to every commit, regardless of who made it or when it was made, ensuring uniform code quality.

## Reduced Review Burden

Human reviewers can focus on architecture and logic instead of spending time catching basic issues like secrets or debugging code.

## Continuous Learning

The AI model improves over time as it's exposed to more of your codebase, becoming increasingly tailored to your team's patterns.

# Module 2 Recap



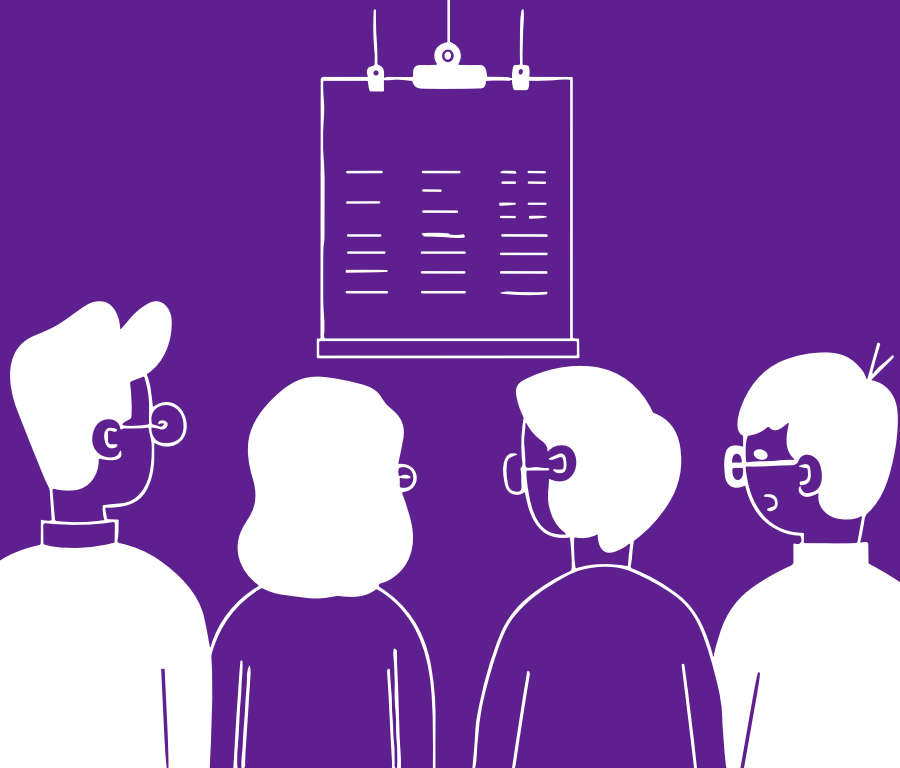
## Key Takeaways:

1. Git hooks provide automation at critical workflow points
2. Pre-commit hooks catch issues before they enter the codebase
3. AI-powered reviews offer consistency and depth
4. The approach shifts quality control left in the development timeline
5. Implementation is straightforward with AI-generated scripts



## Next Steps

Consider expanding this approach to other Git hooks for complete workflow automation.



# Q&A

## Module 2

Let's address your questions about pre-commit hooks and AI-powered code reviews.



# Module 3

## LAB - AI Commit Message Bot

Build an AI commit message generator

# The Goal

## Current Problem:

### ⊗ Typical Commit Messages

- "Fixed stuff"
- "Updates"
- "WIP"
- "Changes requested by John"

These vague messages make code archaeology nearly impossible and complicate release notes, code reviews, and debugging.

## Target Solution:

### ✓ AI-Generated Messages

- "feat(auth): implement password reset endpoint"
- "fix(db): prevent connection leak in user service"
- "refactor(ui): optimize rendering in product list"
- "docs(api): update authentication examples"

Consistent, descriptive messages that follow conventional commits format and provide clear context.



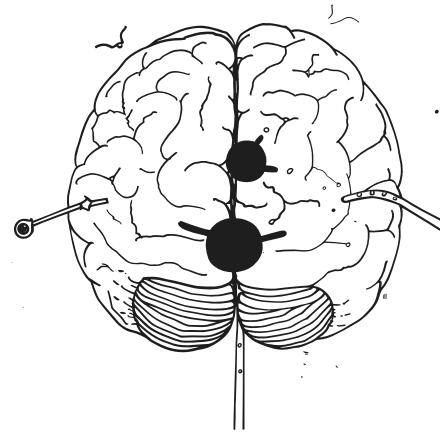


# The Core Concept

## Perfect Context:

The staged changes (`git diff --staged`) provide the ideal context for generating a commit message:

- Shows exactly what's changing
- Includes file paths for scope determination
- Contains actual code with comments
- Reveals intent through code structure



### Extract Diff

Get staged changes



### AI Analysis

Process code context



### Generate Message

Create conventional commit

# The Magic Prompt - Part 1

## Setting the AI Role:

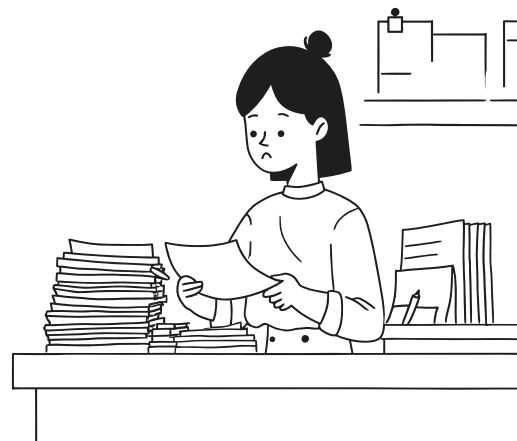
You are CommitGPT, an expert developer with deep knowledge of Conventional Commits format. Your specialty is analyzing code changes and writing clear, descriptive commit messages that follow best practices.

You'll analyze a git diff and produce a concise but informative commit message that accurately describes the changes while following the Conventional Commits specification.

I'll provide the output of ``git diff --staged``. Your task is to generate a commit message based on the changes.

## Key Elements:

- **Persona:** Expert developer with specific knowledge
- **Task Definition:** Clear explanation of what we want
- **Context:** Explaining what input will be provided
- **Standards:** Reference to Conventional Commits



# The Magic Prompt - Part 2

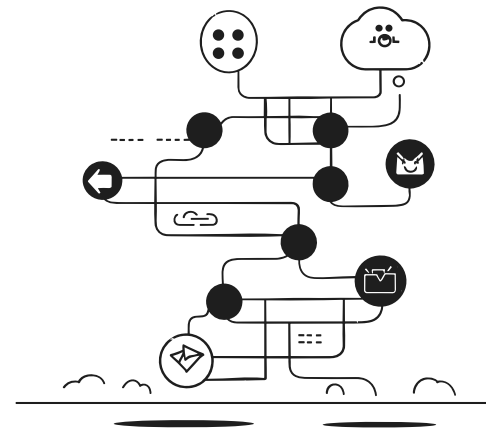
## Setting Rules & Format:

Rules for the commit message:

1. Follow the format: [(scope)]:
2. Use one of these types: feat, fix, docs, style, refactor, perf, test, build, ci, chore
3. The scope is optional but encouraged - derive it from the files or components being changed
4. The description should be concise (under 50 chars) and written in imperative mood ("add" not "adds")
5. Focus on the "why" not just the "what"
6. Don't include ticket numbers or references unless they appear in the diff
7. Return ONLY the commit message, nothing else

## Key Elements:

- **Specific Format:** Clear structure to follow
- **Limited Options:** Constrained type values
- **Quality Guidelines:** Rules for good descriptions
- **Output Constraints:** Clear instruction on what to return
- **Character Limits:** Enforcing conciseness



# LAB TIME: Let's Build!

## What We'll Create:

- A script that extracts staged changes
- An API call to generate commit messages
- A Git alias for easy access
- A custom prompt template

## Tools Needed:

- Python 3.6+
- Git
- OpenAI API key
- Text editor



**Clone starter repo:**

[git.io/ai-commit-lab](https://git.io/ai-commit-lab)



### Repository Contents

- Starter script with comments
- Example prompts
- Sample test data
- Setup instructions

# Lab Instructions



## Clone Repository

Clone the starter repository using: `git clone https://git.io/ai-commit-lab`



## Install Dependencies

Run `pip install -r requirements.txt` to install required packages



## Set API Key

Add your OpenAI API key to the `.env` file or set it as an environment variable



## Review Script

Open `commit-message.py` in your editor and review the code, paying special attention to the prompt template



## Set Up Alias

Add an alias to your Git config: `git config --global alias.ai-commit '!python /path/to/commit-message.py'`

# Pair-Prompt Refinement

## Original Prompt:

You are CommitGPT, an expert developer with deep knowledge of Conventional Commits format. Your specialty is analyzing code changes and writing clear, descriptive commit messages that follow best practices.

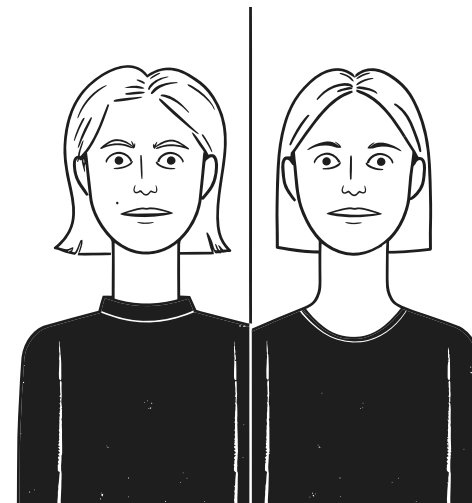
## Potential Improvements:

- Add team-specific conventions
- Include JIRA ticket extraction
- Modify tone (professional vs. casual)
- Adjust format (Conventional vs. custom)

## Enhanced Prompt:

You are CommitGPT, an expert developer on our e-commerce platform team. You have deep knowledge of Conventional Commits format and our team's patterns. You'll extract JIRA ticket IDs (FORMAT: ECM-123) from comments or branch names and include them in the commit message.

Your tone should be professional but clear. For UI changes, include the affected page or component in the scope. For API changes, include the endpoint or service name.



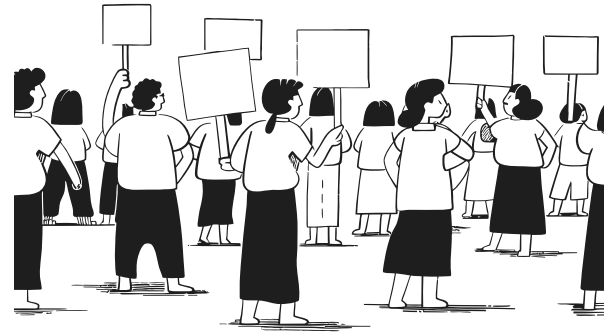
# Lab Showcase & Discussion

## Share Your Results:

- What modifications did you make to the prompt?
- What challenges did you encounter?
- How effective were the generated messages?
- What edge cases did you discover?

### Discussion Point

How could we improve the process for different code styles or languages?



## Example Results:

```
feat(auth): implement password reset flow
fix(cart): resolve price calculation bug in checkout
refactor(api): optimize product search queries
docs(readme): update installation instructions
style(ui): adjust button spacing in mobile view
test(user): add unit tests for profile validation
```

# Beyond Commit Messages



## Documentation

Generate JSDoc comments, README updates, or API documentation based on code changes.



## Release Notes

Aggregate commit messages into comprehensive, user-friendly release notes.



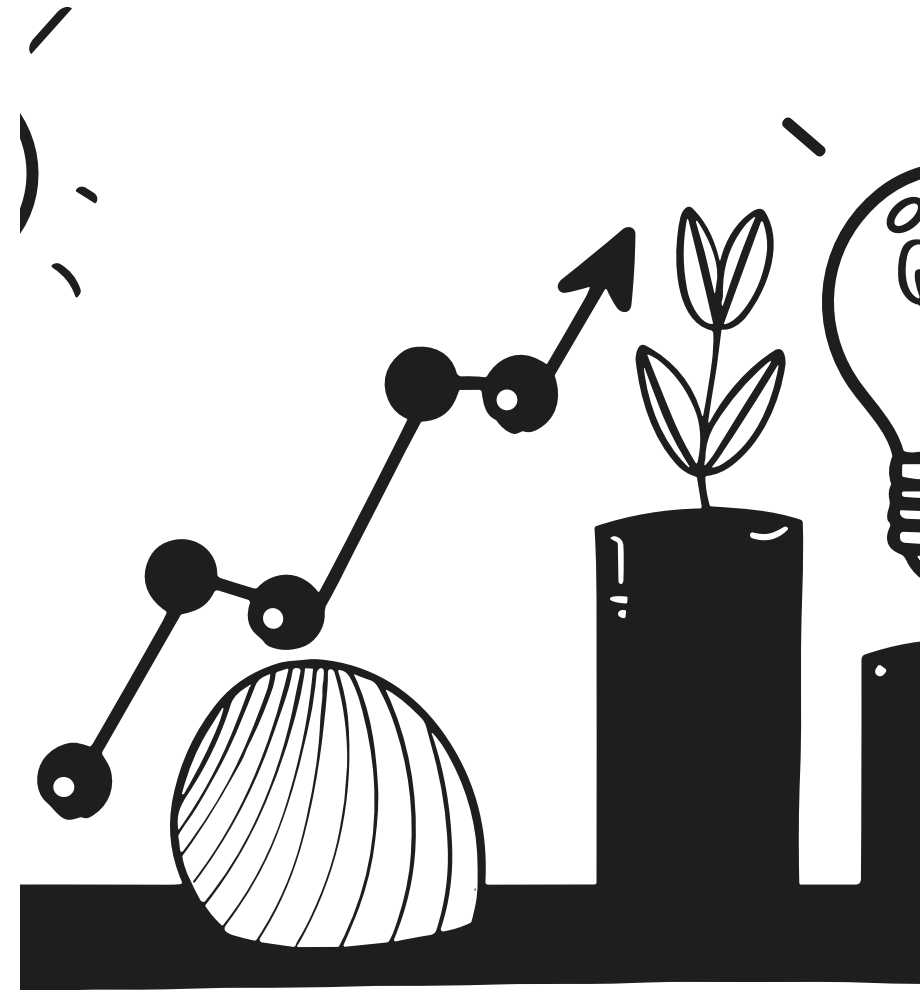
## PR Descriptions

Create detailed pull request descriptions with summaries of changes and testing instructions.



## Refactoring

Suggest refactoring opportunities based on patterns detected in code changes.





# Module 3 Recap



## Diff = Perfect Context

Git diff provides the ideal context for AI to understand code changes.



## Prompt Engineering Unlocks AI

Carefully crafted prompts ensure high-quality, consistent outputs.

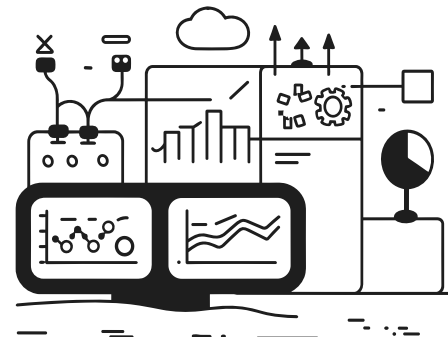


## Automate Creative Tasks

AI excels at summarization and creative tasks that developers often rush.

## Key Takeaways:

1. Code changes themselves provide rich context
2. Conventional Commits improve repository organization
3. Custom prompting creates tailored, team-specific outputs
4. The approach can extend to many development tasks
5. This workflow saves time while improving quality





# Conclusion: Your New Superpowers



## Natural Language CLI

Command your terminal using plain English, eliminating the need to memorise complex syntax or constantly search for commands.

## Automated Quality Gates

Let AI review your code before it enters the repository, catching potential issues and maintaining consistent standards.

## Custom AI Tools

Build specialized AI assistants for your specific workflow needs, from commit messages to documentation generation.

# The Future is Collaborative

## Human + AI > Human or AI

The most powerful development workflows combine human creativity and judgment with AI's pattern recognition and recall abilities.

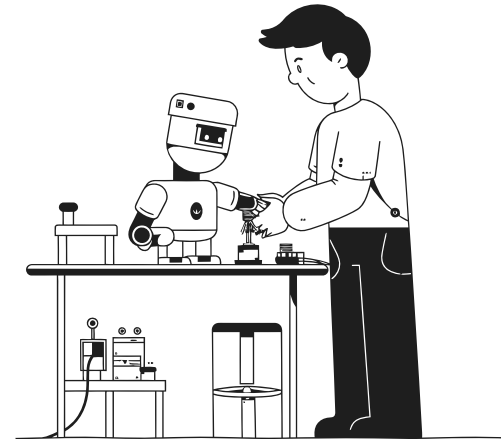
### The Human Element:

- Strategic thinking
- Creative problem-solving
- Ethical considerations
- Context awareness

### The AI Element:

- Pattern recognition
- Knowledge retrieval
- Consistency enforcement
- Repetitive task handling

"AI doesn't replace developers. It amplifies their capabilities by handling the mechanical aspects of coding, allowing humans to focus on the creative and strategic elements."



# Thank You & Q&A

Thank you!

Versha Jain

[b](#)

