

# Agenda

- 01 Pair Programming and AI Pair Programming – Introduction
- 02 Comparing Traditional and AI Pair Programming
- 03 Pair Programming and Copilot – Benefits
- 04 Workflow Transformation – Before and after Copilot
- 05 Best Practices – When to use Copilot and when to be cautious

# Pair Programming – Introduction



# AI Pair Programming



**GitHub  
Copilot**



Real-time code suggestions based on context



Learns from your coding style and patterns



Available 24/7, scales across teams



No ego, no judgment—just suggestions



---

Traditional  
Pair Programming

vs



---

AI Pair  
Programming



## Traditional Pair Programming

Knowledge exchange



Real-time feedback



Error catching





# AI Pair Programming

Pattern recognition



Instant suggestion



Documentation generation



# Pair Programming – Benefits

**01**



Human creativity  
and intuition

**02**



Complex  
architectural  
decisions

**03**



Knowledge transfer  
between  
teammates

**04**



Social learning and  
team building

# Copilot – Benefits

01



Instant access to  
common patterns

02



Rapid prototyping and  
boilerplate generation

03



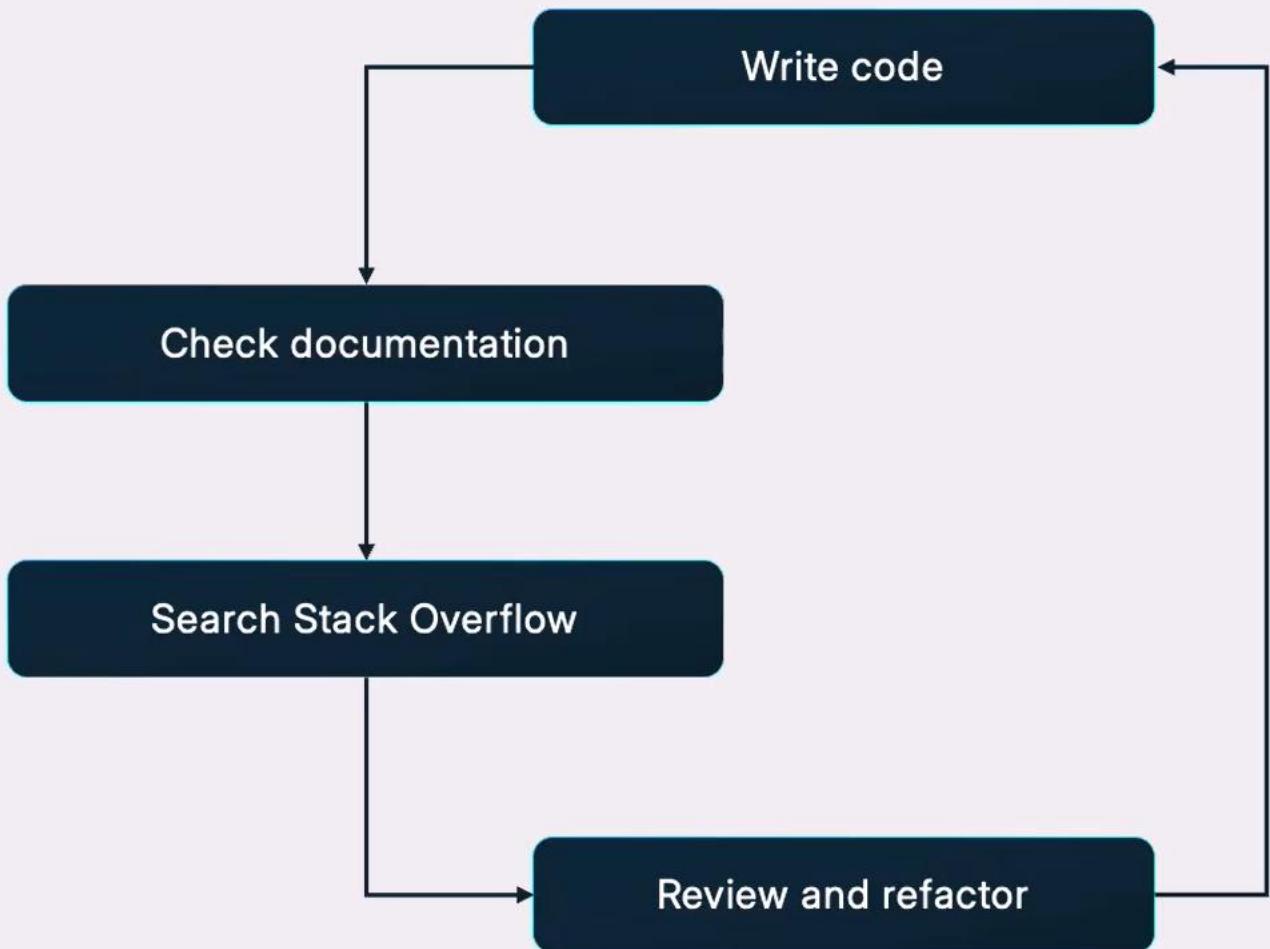
Consistent code style  
suggestions

04

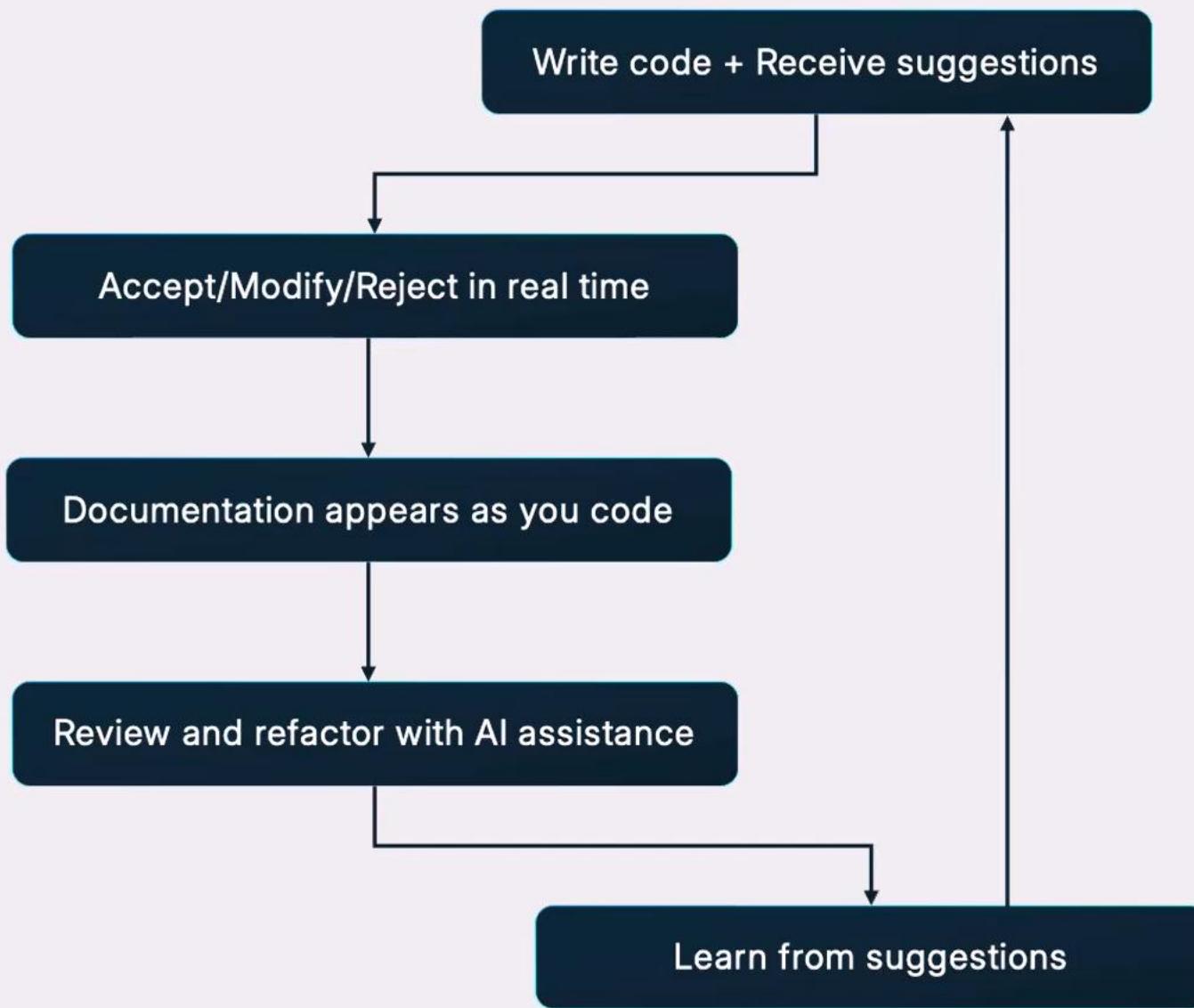


Documentation  
assistance

## Workflow: **Before Copilot**



# Workflow: After Copilot



# When to Use Copilot

01



Repetitive tasks and  
boilerplate code

02



API integration  
patterns

03



Test case  
generation

04



Documentation  
writing

05



Converting  
comments to code

# When to Be Cautious



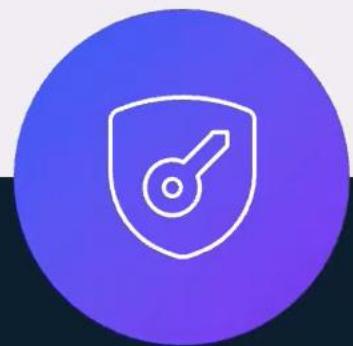
Security-critical  
code



Complex  
business logic



Performance-  
critical sections



Licensed/  
Proprietary  
algorithms

## Conclusion

- 01 AI pair programming is a tool, not a replacement
- 02 Start small, expand gradually
- 03 Learn from the suggestions
- 04 Share your experiences
- 05 Keep experimenting and evolving



# Core Strengths

O1



Increased productivity

O2



Blazing-fast boilerplate generation

O3



Smart context understanding

**GitHub  
Copilot**

# Core Strengths



**GitHub  
Copilot**

04



Multi-language support (200+ languages)

05



Documentation generation

06



Test case suggestions

# Productivity Boosters



Real-time code completion



API pattern recognition



Repetitive task automation



Consistent coding style



Integrated development workflow

# Learning and Collaboration



## Collaboration

Shared knowledge



Code reviews



Documentation



# Benefits

01



Increased productivity

02



Reduced cognitive load

03



Faster prototyping

04



Built-in documentation help

05



Learning tool for new patterns

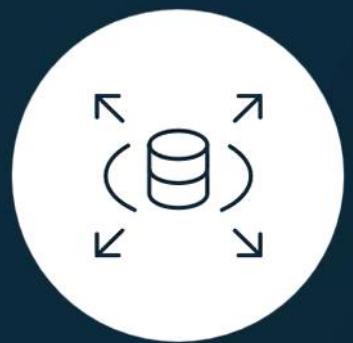
# Limitations – Performance



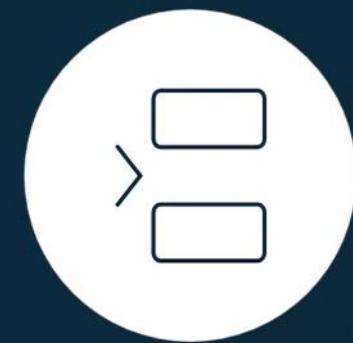
Network dependency required



Can lag with large codebases



Resource-intensive operation



Occasional context misalignment



API rate limits

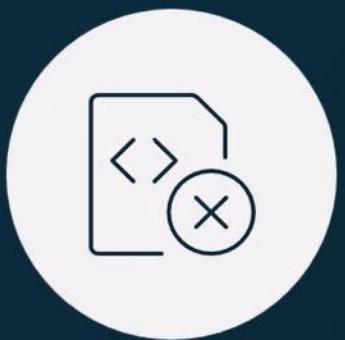
# Limitations – Code Quality



May suggest  
outdated  
patterns



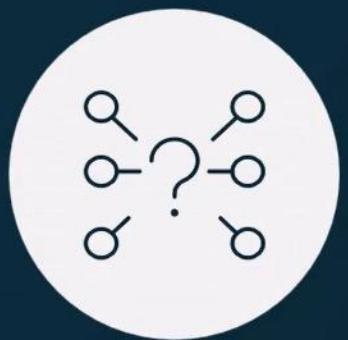
Security  
vulnerabilities  
possible



Inconsistent  
code quality

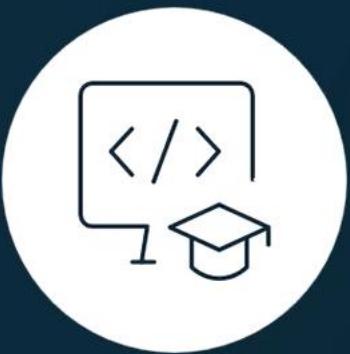


Generic solutions



Limited project-  
specific  
knowledge

# Limitations



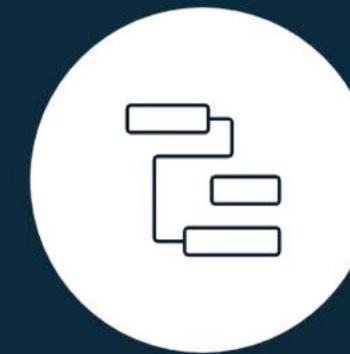
Not a replacement for understanding



May suggest outdated patterns



Requires careful review



Struggles with complex algorithms

# Maximizing Copilot's Value

01



Write clear  
comments to  
guide suggestions

02



Use descriptive  
variable names

03



Break complex  
tasks into smaller  
chunks

04



Review  
suggestions  
carefully

05



Learn from  
Copilot's  
patterns

# Common Pitfalls



Over-reliance on  
generated code



Accepting suggestions  
without review



Using Copilot for  
complex algorithms



Ignoring security  
implications

## Key Takeaways

01 Powerful tool with clear boundaries

02 Balance automation with oversight

03 Focus on appropriate use cases

04 Maintain security awareness

05 Continuous evaluation needed

# Understanding the SDLC

It provides a structured approach to build software.



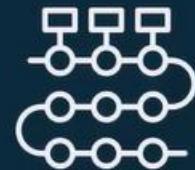
# SDLC Benefits

01



Organized Approach

02



Clear Milestones

03



Quality Control

# SDLC Challenges

01



Time Consuming

02



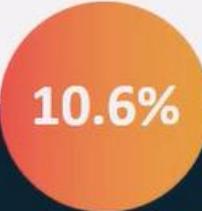
Knowledge Gaps

03



Technical Debt

# Impact of GitHub Copilot



10.6%

Pull Requests

Increase in pull requests



3.5

Cycle Time

Hour's reduction in cycle time



90%

Job Fulfillment

Developers report  
increased job fulfillment

# Requirements and Planning

Copilot connects natural language with technical documentation.



Natural Language



GitHub Copilot



Technical  
Documentation

# Requirements and Planning



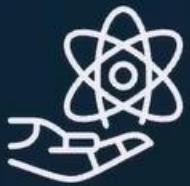
Design Specifications

**It helps outline:**

- Architectural decisions
- Component interaction
- System constraints

Ensures thorough planning and reduces developers' documentation time

# The Core Principles



**Six Fundamental  
Principles**

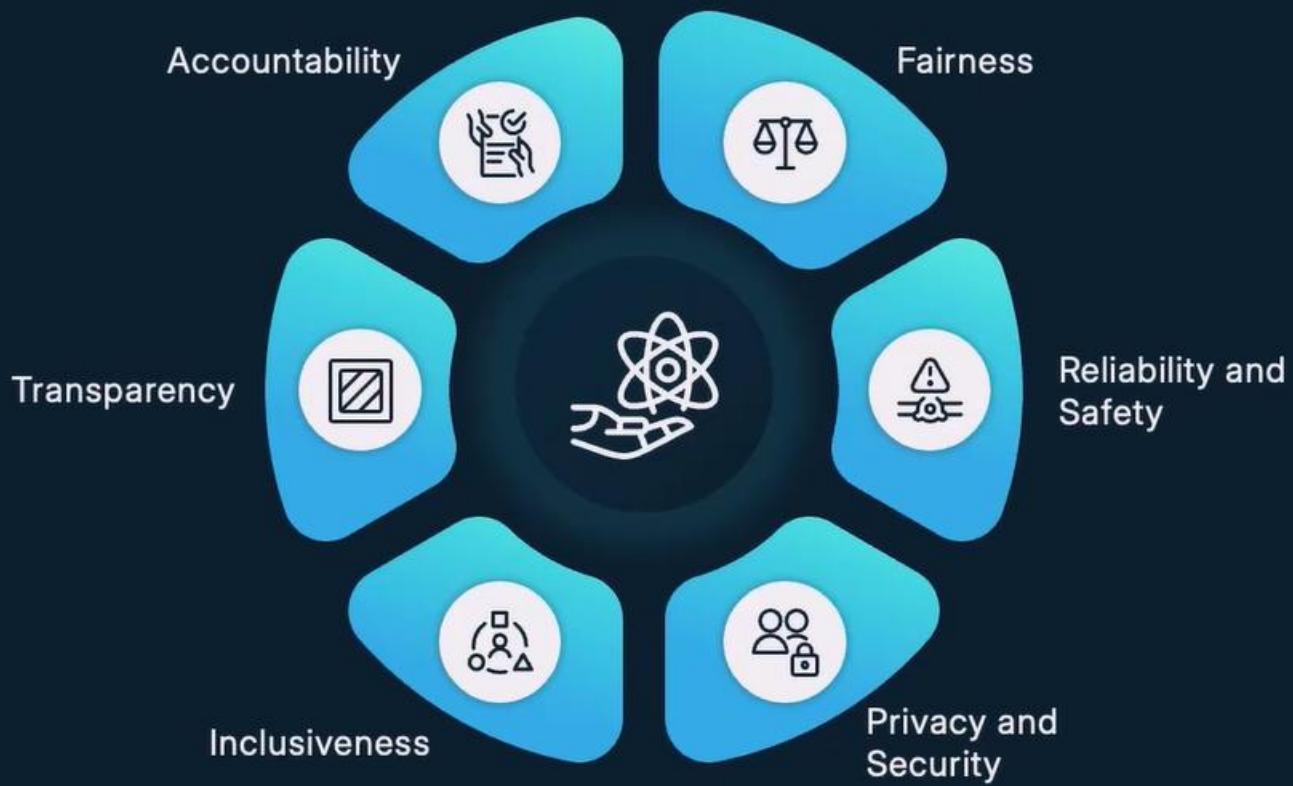


**Ethical and  
Responsible  
Implementation**



**Holistic  
Approach**

# The Six Fundamental Principles



# Fairness



AI systems must treat all people fairly

# Fairness



**GitHub Copilot**

- Microsoft reviews training data for bias
- Tests across diverse demographics
- Uses techniques like adversarial debiasing
- Monitors performance and even lets you override unfair outputs

# Fairness



## Core Requirements

AI systems must treat all people fairly, ensuring consistent recommendations across similar cases and avoiding differential impacts on similar groups

# Fairness

## Key Implementation Methods



Reviewing  
training data  
for bias



Testing with balanced  
demographic  
samples



Using adversarial  
debiasing  
techniques



Monitoring  
performance across  
user segments

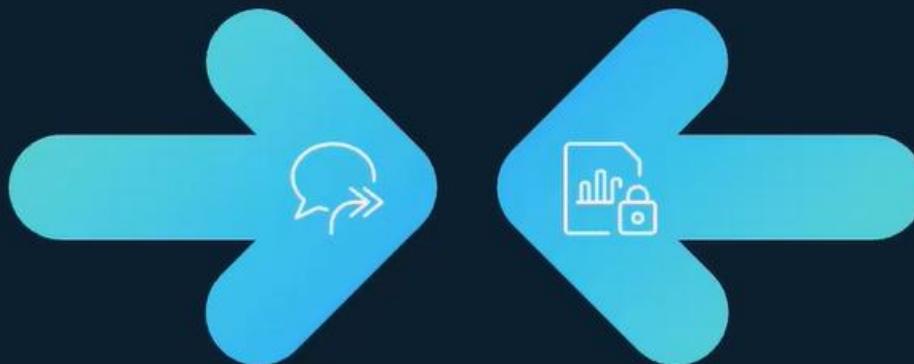


Implementing  
override controls for  
unfair model scores

# Reliability and Safety

## Copilot's Reliability

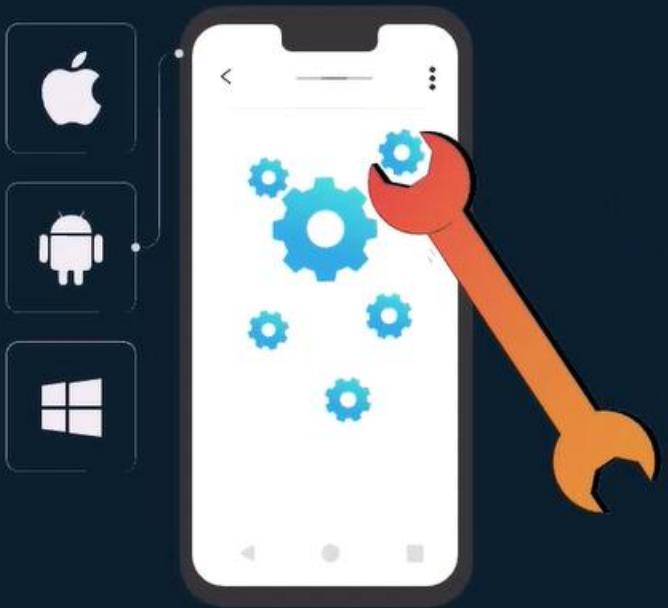
Deliver consistent  
suggestions



Handle weird  
inputs safely

Copilot is resistant to manipulation and minimizes harm, whether that's buggy code or security risks.

# Reliability and Safety



## Consistent and Safe Operation

Systems must operate consistently and safely, functioning as designed and providing safe responses to unexpected conditions.

# Fairness

Emphasis

Predictable  
behavior



Robust testing and  
development

Crucial

Predictable behavior under normal conditions and robust testing and development processes are crucial.

# Privacy and Security



Encryption

Hardware Security Modules

Azure's Secure Storage

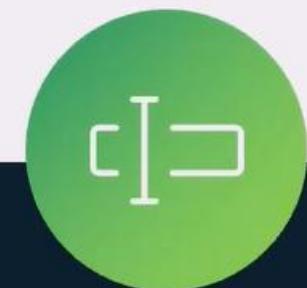
# Privacy and Security



Data  
Collection



Security  
Measures



Anonymization  
Techniques



Data  
Collection



Security  
Measures



Anonymization  
Techniques

# Privacy and Security

Obtaining explicit user permission

Collecting only essential data

Regular monitoring of data inputs



Data  
Collection



Security  
Measures



Anonymization  
Techniques

# Privacy and Security

Strong encryption during transfer and storage

Hardware Security Modules (HSMs)

Secure key storage in systems like Microsoft Azure

Implementation of envelope encryption

Regular security audits and access controls

# Inclusiveness



**GitHub Copilot**

## Supports Diverse Users

Screen reader compatibility and Offline functionality



## Global Input

Understands regional coding styles and dialects

# Inclusiveness

Systems must work well for diverse users and be accessible for all individuals with physical and mental abilities

1



**Empowering All Users**

**Global Availability**



AI should be available in developing regions and support limited connectivity

Development should include diverse community input and ensure equal benefits distribution

3



**Diverse Input**

**Implementation Examples**



Cross-demographic facial recognition, screen reader support, regional dialect support, alternative interaction modes, offline functionality

4

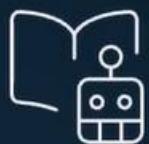
# Transparency



Microsoft provides explanatory interfaces and offers debugging tools

# Transparency

01



Understandable  
Systems

02



Honest  
Communication

03



Comprehensive  
Auditability

04



Implementation  
Methods

# Accountability

01 |  Responsible Operation

02 |  Continuous Monitoring

03 |  Regular Audits

04 |  Company Responsibility

05 |  Proactive Management

## Summary

- 01 **Fairness** keeps Copilot equitable
- 02 **Reliability and Safety** make Copilot dependable
- 03 **Privacy and Security** protect your data
- 04 **Inclusiveness** opens it to all
- 05 **Transparency** explains its moves
- 06 **Accountability** ensures oversight

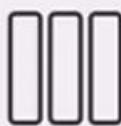
# Prompt Engineering – Introduction

<b>Definition</b>	Process of crafting clear instructions to guide AI systems
<b>Purpose</b>	Generate context-appropriate, syntactically correct code
<b>Focus</b>	Project-specific needs and requirements



# The Four S's of Prompt Engineering

**01**



Single

**02**



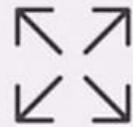
Specific

**03**



Short

**04**



Surround

# The Four S's of Prompt Engineering

**01**



**Single**

Prompt should focus on one well-defined task or question at a time

**02**



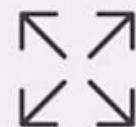
**Specific**

**03**



**Short**

**04**



**Surround**

# The Four S's of Prompt Engineering

**01**



Single

**02**



Specific

Prompts should be  
more explicit and  
detailed

**03**



Short

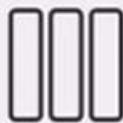
**04**



Surround

# The Four S's of Prompt Engineering

**01**



Single

**02**



Specific

**03**



Short

Concise prompts  
are easier for the AI  
to process  
completely

**04**



Surround

# The Four S's of Prompt Engineering

**01**



Single

**02**



Specific

**03**



Short

**04**



Surround

Keep relevant files  
open to provide the  
AI with ambient  
context

# Clarity and Context

01



Build on  
single/specific  
principles

02



Provide explicit  
instructions

03



Include relevant  
contextual details

04



Use comments to  
give additional  
context

# Examples and Iteration

01



Use examples to clarify requirements

02



Treat responses as part of a dialog

03



Iterate and refine prompts if needed

04



Enrich initial comments with additional details

# Prompting Approaches

## 1 Zero-Shot Learning

Generates code  
without specific  
examples



Relies on  
foundational  
training



Uses natural  
language  
descriptions



# Prompting Approaches

2

## One-Shot Learning

Uses a single example for context



Generates similar code based on example



Provides more context-aware responses



# Prompting Approaches

3

## Few-Shot Learning

Uses  
multiple  
examples



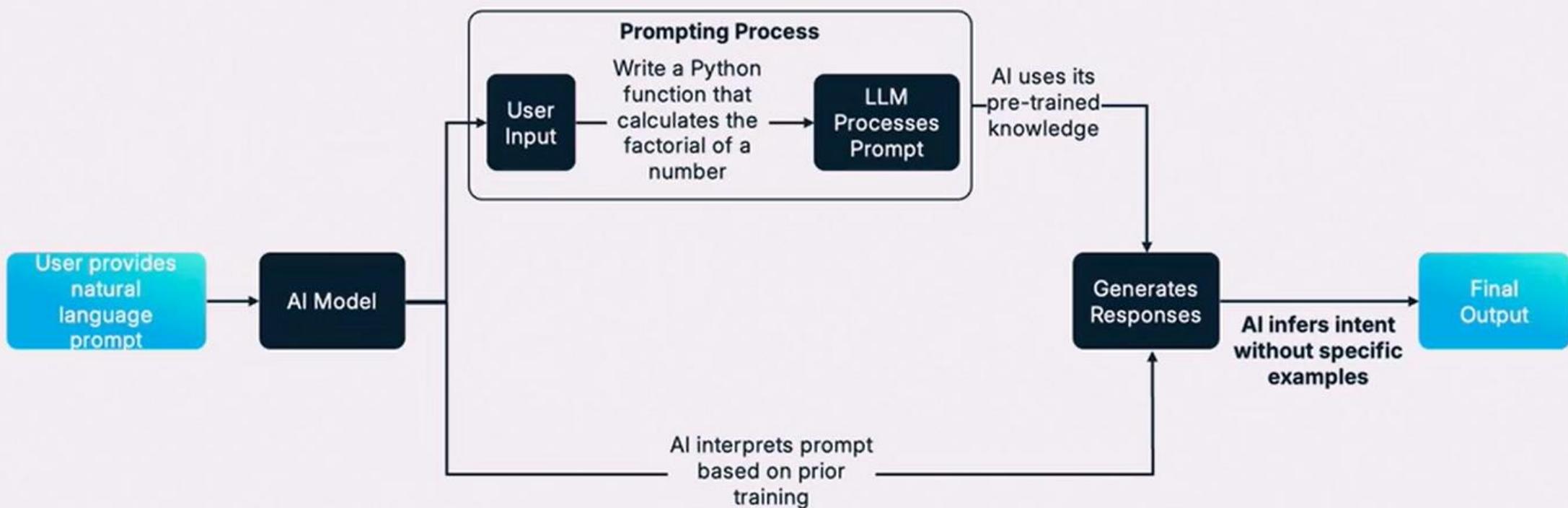
Balances between  
unpredictability  
and precision



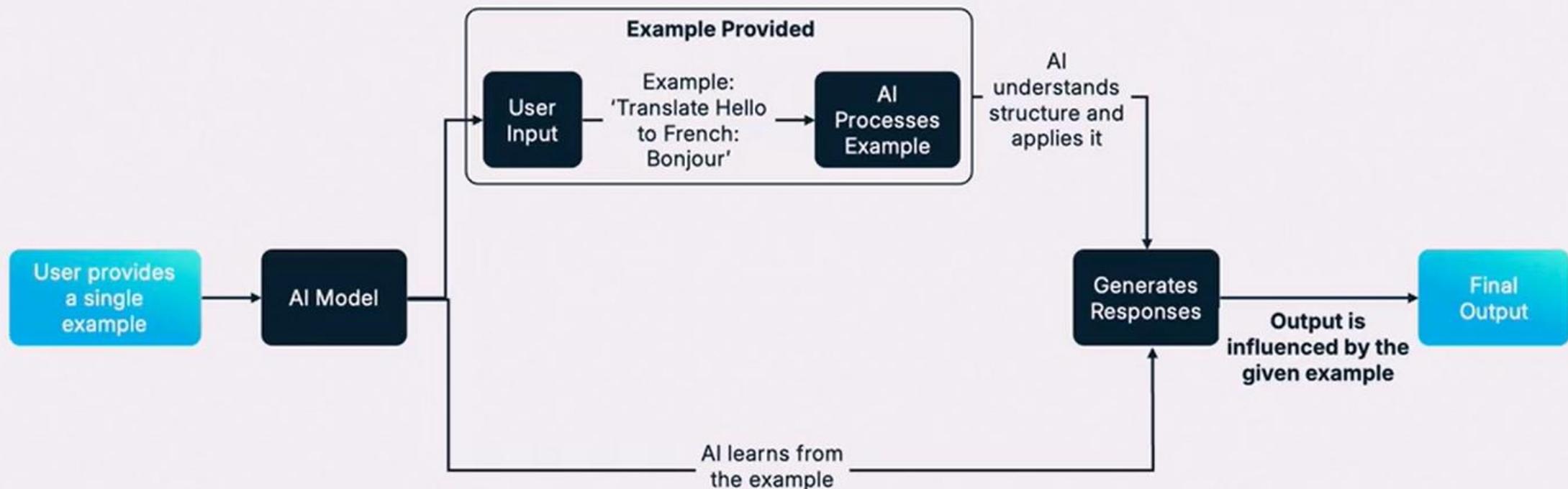
Provides  
more refined  
outputs



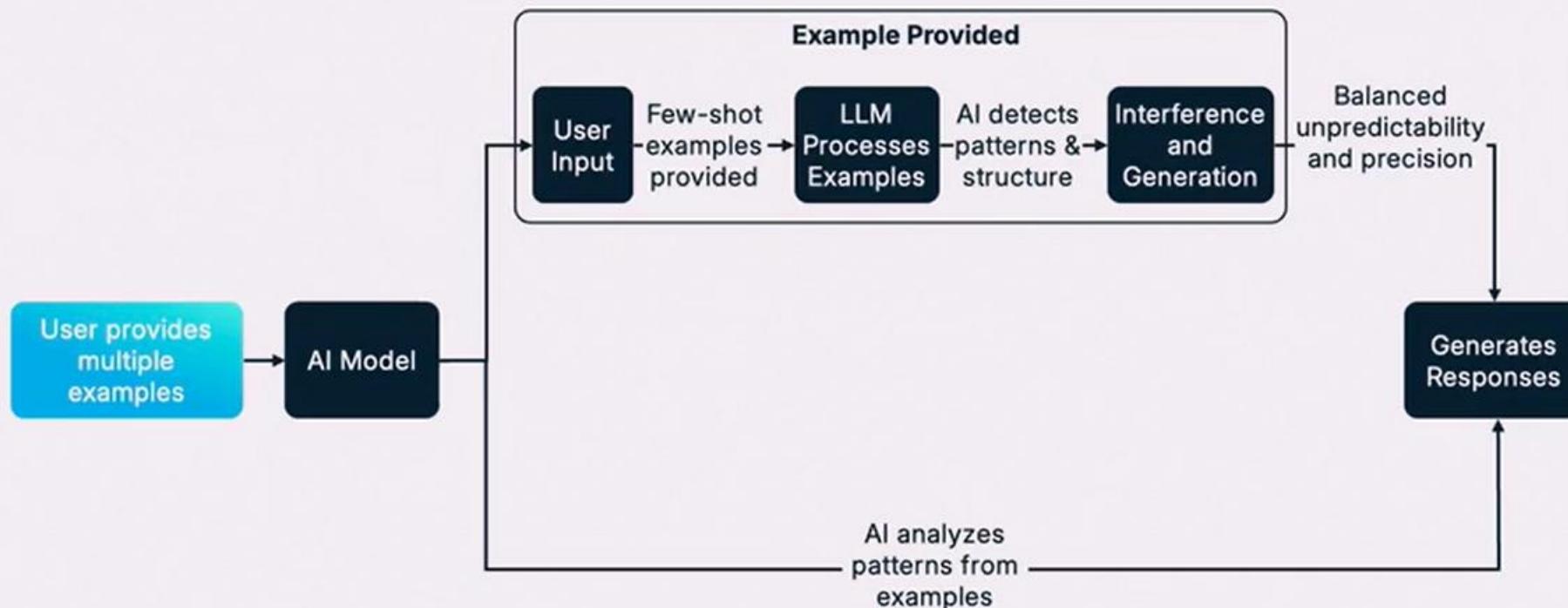
# Zero-Shot Learning



# One-Shot Learning



# Few-Shot Learning



# Prompt Engineering – Best Practices

## Examples and Iteration

01



Use examples to clarify requirements

02



Treat responses as part of a dialog

03



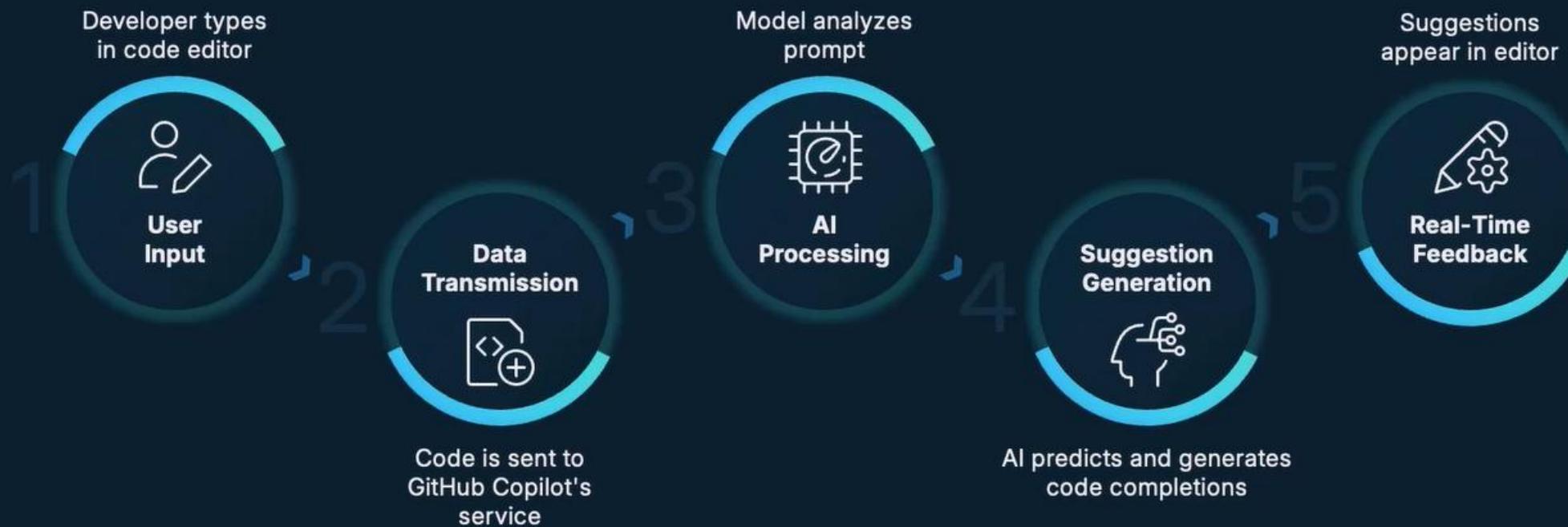
Iterate and refine prompts if needed

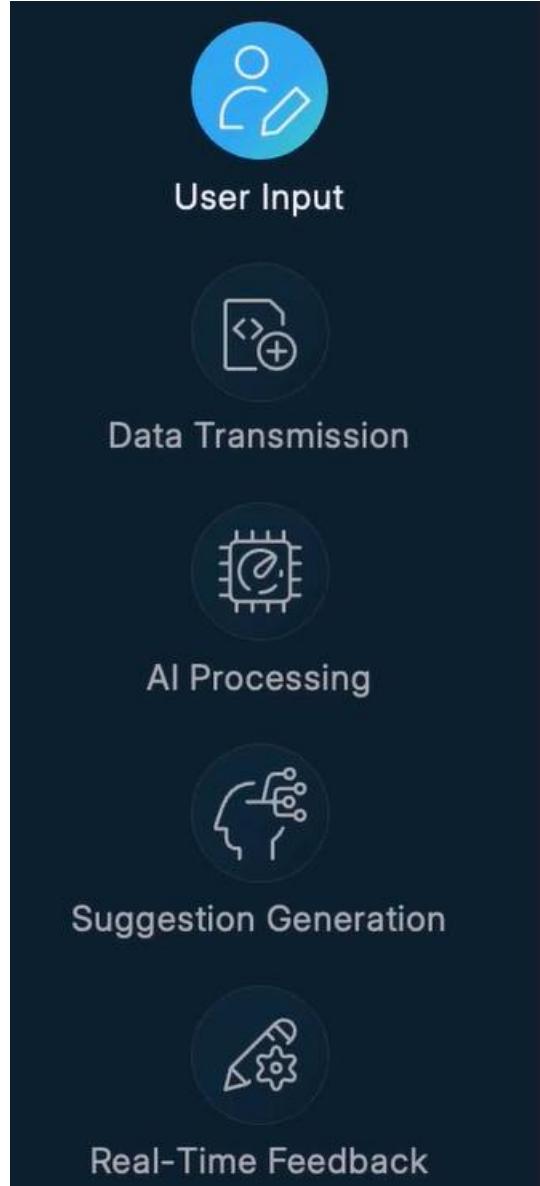
04



Enrich initial comments with additional details

# The GitHub Copilot Workflow





# The GitHub Copilot Workflow – Step 1

**01**

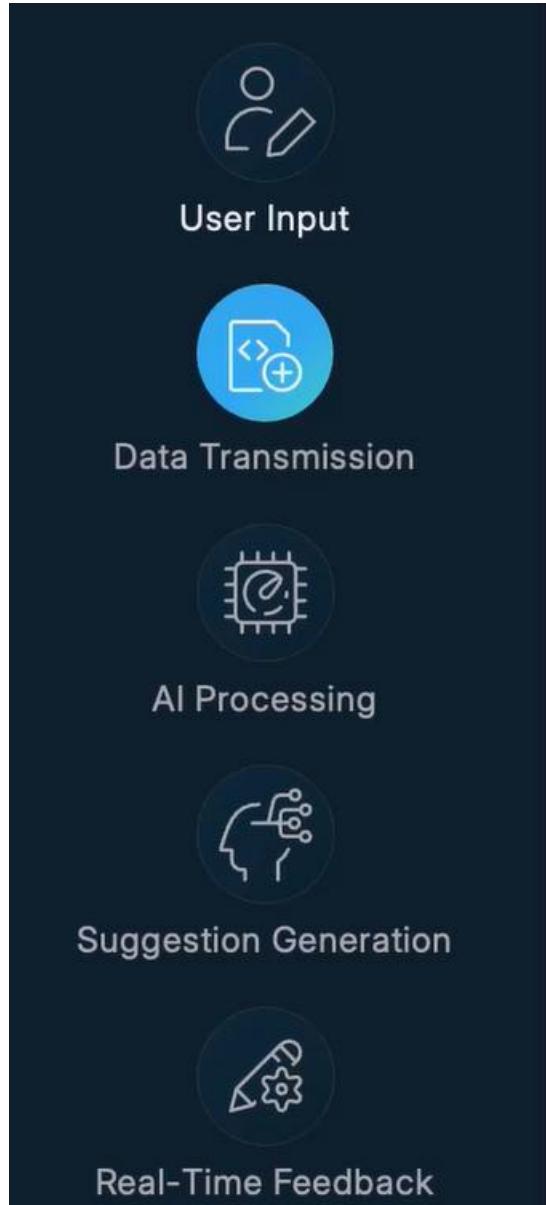
You type in your  
code editor  
(VS Code,  
JetBrains, etc.)

**02**

As you write, the  
GitHub Copilot  
extension tracks  
what you've  
typed

**03**

Extension  
analyzes your  
current context in  
real time



## The GitHub Copilot Workflow – Step 2

**01**

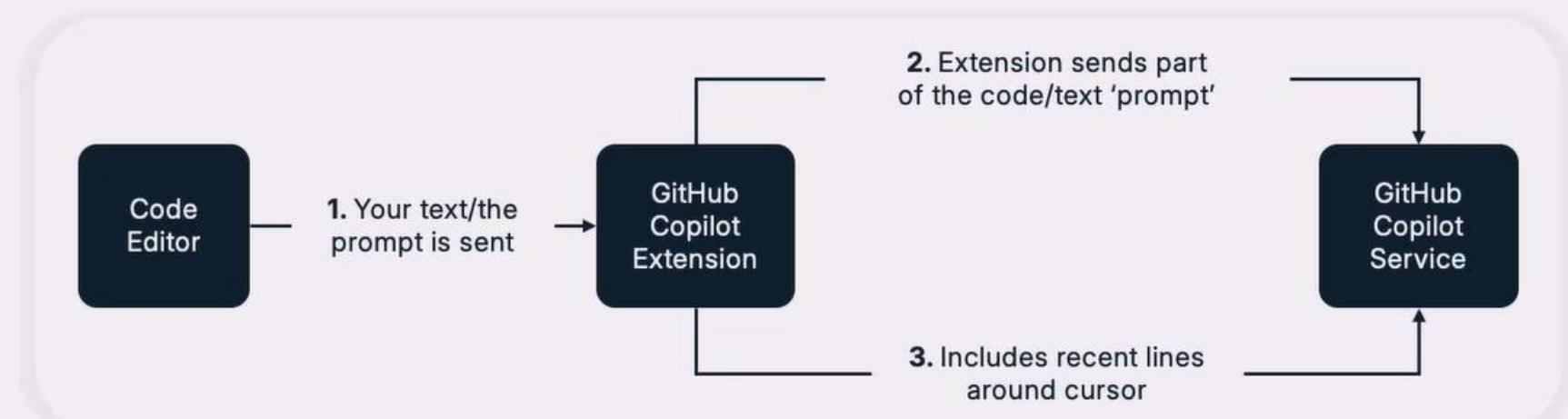
Your text is sent  
to the GitHub  
Copilot service

**02**

The extension  
sends part of  
your code/text  
(the "prompt")

**03**

Includes recent  
lines around your  
cursor for  
context





User Input



Data Transmission



AI Processing



Suggestion Generation



Real-Time Feedback

# The GitHub Copilot Workflow – Step 3

**01**

On the server, an AI model analyzes the prompt

**02**

Model is powered by OpenAI's Codex or GPT technology

**03**

Pattern recognition based on training data (open-source code repositories)

This analysis happens at multiple levels and enables Copilot to generate contextually relevant and syntactically correct code suggestions in the next step



User Input



Data Transmission



AI Processing



Suggestion Generation



Real-Time Feedback

# The GitHub Copilot Workflow – Step 4

**01**

Based on the previous analysis, the model predicts what comes next

**02**

Multiple completion options may be generated

**03**

Predictions are ranked by confidence

**04**

Results are sent back to your editor



User Input



Data Transmission



AI Processing



Suggestion Generation



Real-Time Feedback

# The GitHub Copilot Workflow – Step 5

**01**

Suggestions appear in your editor as "ghost text"

**02**

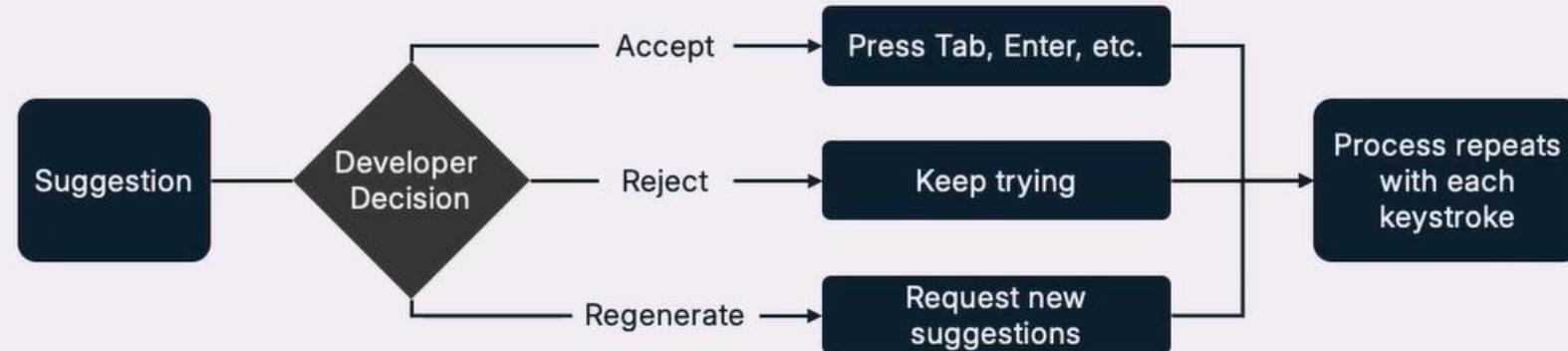
Multiple suggestions may be available

**03**

Appears nearly instantaneously as you type

# The GitHub Copilot Workflow – Step 6

Developer Decision



**Accept**

Press Tab,  
Enter, etc.



**Reject**

Keep typing



**Cycle**

Use keyboard  
shortcuts



**Repeat**

Process repeats  
with each keystroke

## Conclusion: The Complete Workflow

- 01 User Input
- 02 Data Transmission
- 03 AI Processing

# Writing Effective Prompts

1 Be specific about desired functionality

2 Provide context in comments before your code

3 Include edge cases or constraints in your prompts

4 Consider the file type and programming language

```
name: str
description: Optional[str] = None
price: float
quantity: int

# In-memory data store
items = []

# route route to accept pings and return a success message
@app.get("/")
def read_root():
    return {"message": "Welcome to the FastAPI Root"}
```

 Edit with Copilot

# Start your editing session by defining a set of files that you want to work with. Then ask Copilot for the changes you want to make.

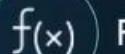
```
# Copilot is powered by AI, so mistakes are possible. Review output carefully before use.
@app.get("/items", response_model=Item)
async def get_items():
    return items
```

or type # to attach context

```
# Get a specific item by index
@app.get("/items/{item_id}", response_model=Item)
async def get_item(item_id: int):
    if item_id < 0 or item_id >= len(items):
        return {"error": "Item not found"}
    return items[item_id]
```

```
# Create a new item
@app.post("/items", response_model=Item)
```

# Common Prompt Patterns

 Function description prompts

 Bug fix requests framed as comments

 Code refactoring prompts

 Test generation prompts

```
how can I refactor this to ensure it meets Pep8 standards?  
    trace: float  
    quantity: int  
  
GitHub Copilot  
  
Here's how to refactor your code to meet PEP8 standards:  
1. Install autopep8: If you don't have it already, install autopep8 using pip: pip install autopep8 and return a success message.  
2. Run autopep8: Run autopep8 on your file: autopep8 --in-place --aggressive --aggressive  
# c:/Users/jerem/Projects/APIDemo/app.py  
# Get all items  
@app.get("/")  
async def get_items():  
    return items  
  
# Get a specific item by index  
@app.get("/items/{item_id}", response_model=Item)  
async def get_item(item_id: int):  
    if item_id < 0 or item_id >= len(items):  
        return {"error": "Item not found"}  
    return items[item_id]  
  
# Create a new item
```

# Prompt Security – Best Practices

**01**



Understand what to  
avoid in prompts

**02**



Handle sensitive  
information API keys,  
passwords, or  
personal data

**03**



Be aware of  
corporate policy  
considerations

**04**



Maintain data privacy  
awareness  
throughout the  
development process

# Copilot's Learning Mechanisms



Global improvement



Project-specific learning



Language and framework adaptation



Pattern recognition over time

## Key Takeaways

- 01 Copilot processes prompts through a secure, multi-stage flow
- 02 Multiple safeguards ensure quality, security, and ethics
- 03 Feedback drives continuous improvement
- 04 Understanding the flow helps developers write better prompts

# Learning Objectives

- 01 The inbound flow of prompt processing
- 02 The outbound flow of code generation and delivery
- 03 Security and filtering mechanisms in place
- 04 How feedback improves Copilot over time

# The Dual Flow Process

## 1. Inbound Flow

01



User prompts

02



Context gathering

03



Processing

04



Code generation

# The Dual Flow Process

## 2. Outbound Flow

01



Generated code

02



Post-processing

03



Validation

04

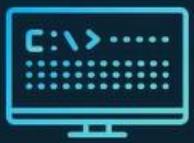


Delivery

# The Dual Flow Process



# Why Understanding the Flow Matters?



Better prompts is equal to  
better code suggestions



Security considerations when  
using AI in development



Maximizing productivity  
with Copilot

# Step 1: Secure Prompt Transmission

## HTTPS Communication



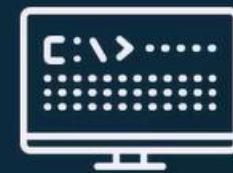
All communications occur over HTTPS

## Secure Transmission



User prompts are securely transmitted to GitHub Copilot's servers

## Prompt Types



Types of prompts: Chat interactions, natural language comments

# Step 2: Context Gathering

## Code Context

Code before and after cursor position

## File Details

Filename and type details

## Project Information

- Adjacent open tabs information
- Project structure and file paths

## Technical Environment

Programming languages and frameworks in use

## FIM Technique

Fill-in-the-Middle (FIM) technique explained

## Step 3: Proxy Filtering



**GitHub-Owned Microsoft  
Azure Tenant**

Hosts proxy server



**Traffic Filtering**

Prevents prompt hacking  
attempts



**System Integrity Protection**

Security guardrails for prompt  
processing

## Step 4: Toxicity Filtering

Content filtering mechanism

Personal data protection



Prevention of hate speech and inappropriate content

Ethical and responsible code

# Step 5: Code Generation With LLM

## LLM Processing

How Large Language Models process the filtered prompt

## Context Integration

Context integration for relevance

## Code Suggestion

Generation of appropriate code suggestions

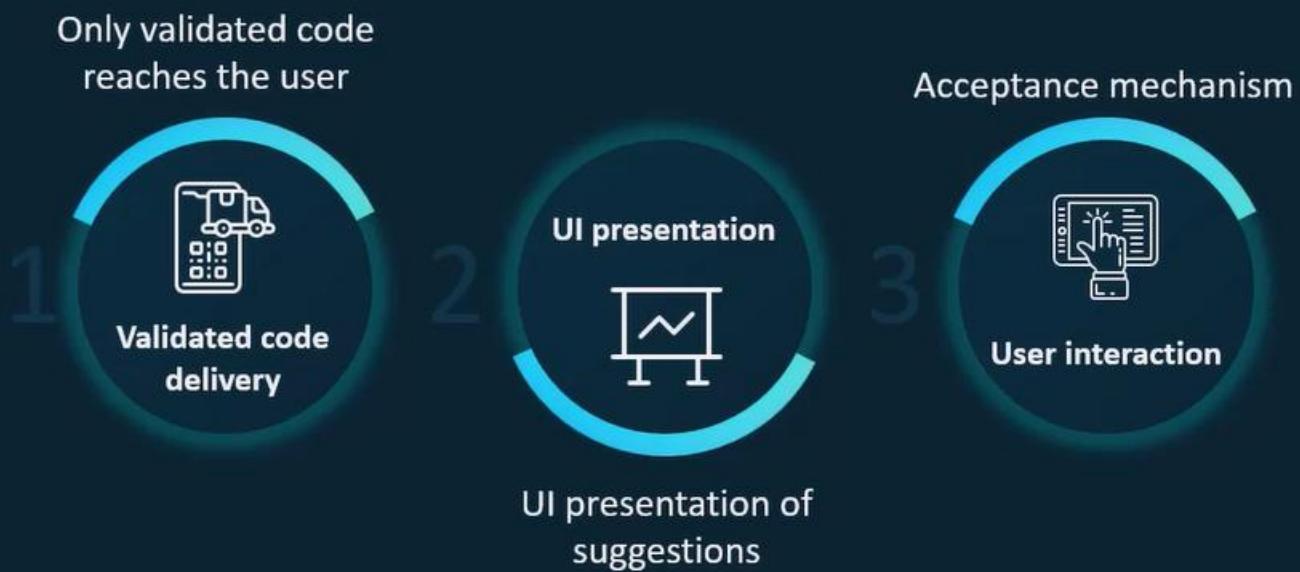
## Project Alignment

Project-specific requirement alignment

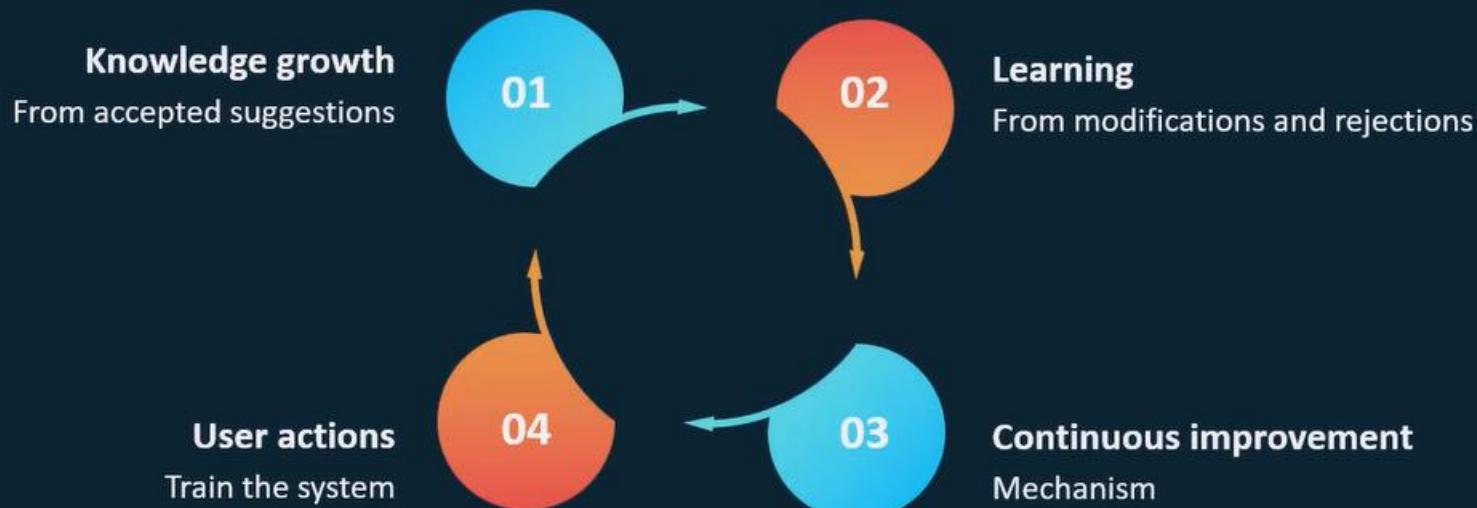
# Step 1: Post-Processing and Response Validation



## Step 2: Suggestion Delivery

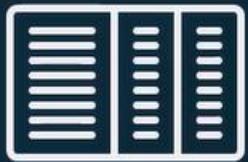


## Step 3: Feedback Loop Initiation



## Step 4: Repeat Process

01



Cumulative feedback  
application

02



Improvement in  
understanding user intent

03



Refinement of code  
generation over time

04



Enhanced context  
utilization

# Writing Effective Prompts



Be specific about desired functionality



Provide context in comments



Include edge cases or constraints



Consider the file type and programming language

# Common Prompt Patterns

01



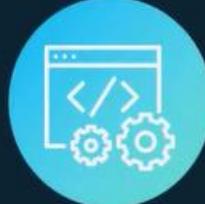
Function description  
prompts

02



Bug fix requests

03



Code refactoring  
prompts

04



Test generation  
prompts

# Prompt Security – Best Practices



What to avoid in  
prompts



Handling sensitive  
information



Corporate policy  
considerations



Data privacy  
awareness

# Copilot's Learning Mechanisms



Global improvement



Project-specific learning



Language and framework adaptation



Pattern recognition over time

# Administrator Controls



Optional filters



Enterprise policy  
management



Security controls and  
configurations



Compliance  
considerations

## Key Takeaways

- 01 Copilot processes prompts through a secure, multi-stage flow
- 02 Multiple safeguards ensure quality, security, and ethics
- 03 Feedback drives continuous improvement
- 04 Understanding the flow helps developers write better prompts

# Available Slash Commands

**/doc**

Inserts documentation comments related to the cursor position, such as generating comments for methods

**/exp**

Starts a new conversation thread with a fresh context, allowing you to switch between different threads

**/explain**

Provides an explanation for selected code; if code is selected in the editor, Copilot will explain that specific code

**/fix**

Proposes fixes for problems, typos, and errors in the selected code

# More Slash Commands

**/fixTestFailure**

Finds and fixes failing tests

**/generate**

Generates code snippets based on your question or prompt

**/help**

Provides quick reference and basics about using GitHub Copilot

**/optimize**

Analyzes the selected code and suggests optimizations for performance, efficiency, or code length

**/tests**

Creates unit tests for the selected code

**/clear**

Starts a new chat session

**/new**

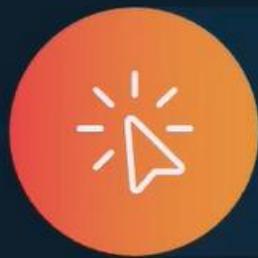
Creates a new project

# Accessing Slash Commands



## Type a slash

Type a forward slash (/) directly in the Copilot Chat message box



## Click the Slash button

Click the Slash button in the GitHub Copilot Chat window



## Use inline chat

Use the inline chat by pressing Alt+/  
or right-clicking and selecting "Ask  
Copilot" from the context menu

# Seamless Integration

01



**Main Chat Window:** Use slash commands in the dedicated Copilot chat panel

02



**Inline Chat Dialog:** Access commands without leaving your code context

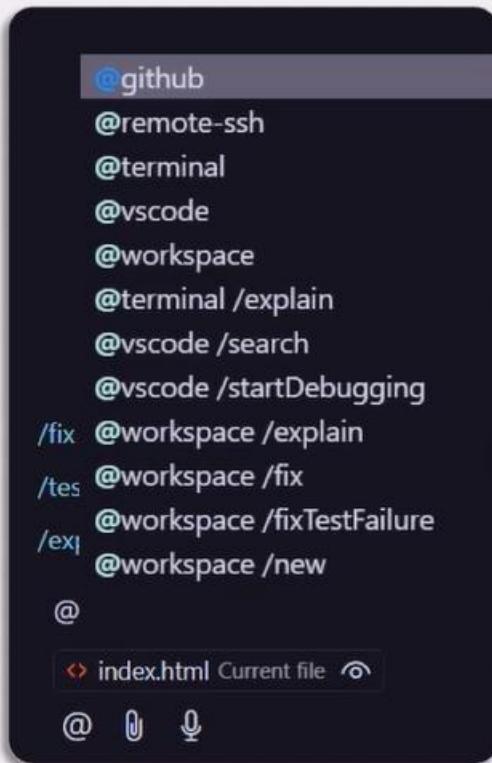
03



**Continuous Workflow:** Get assistance without disrupting your coding flow

# Functionality and Features

## How Copilot Agents Operate



Agents are triggered by "@" mentioned in the chat interface (e.g., @workspace, @azure, @github).

Each agent has specialized knowledge and access to specific contexts.

Agents analyze relevant information sources (workspace files, terminal output, etc.) to provide tailored assistance.

Agents use natural language processing to understand queries and generate appropriate responses.

# Usage

## Accessing and Invoking Agents

Type "@" in the chat prompt box to see available agents

Select a specific agent by typing "@" followed by the agent's name (e.g., @workspace)

Frame your question or request specifically for the chosen agent's expertise

# Common Agents and Their Purposes



**@workspace**

Understands project structure, dependencies, and code relationships



**@terminal**

Assists with terminal commands, shell operations, and command-line tools



**@vscode**

Helps with VS Code-specific features, commands, and workflows



**@azure**

Assists with Azure services, deployments, and configurations (currently in preview)



**@github**

Provides GitHub-specific Copilot skills and integrations

```
select_file)
self.file_button.pack(pady=5)

tk.Label(root, text="Number of top words:").pack()
self.n_words = tk.Entry(root)
self.n_words.insert(0, "20")
self.n_words.pack()

self.plot_button = tk.Button(root, text="Plot")
self.plot_button.pack(pady=5)

self.plot_frame = tk.Frame(root)
self.plot_frame.pack(fill=tk.BOTH, expand=True)

self.filename = None

select_file(self):
self.filename = filedialog.askopenfilename(
    filetypes=[("Text Files", "*.txt"), ("All"])

plot(self):
if self.filename:
    for widget in self.plot_frame.winfo_children():
        widget.destroy()
```

## Introduction – The Importance of Context

Context is key to unlocking Copilot's true power

Guides the AI to generate more accurate responses

Context manipulation = Better quality suggestions

## Context Management – Goals



Enhance the relevance  
of suggestions

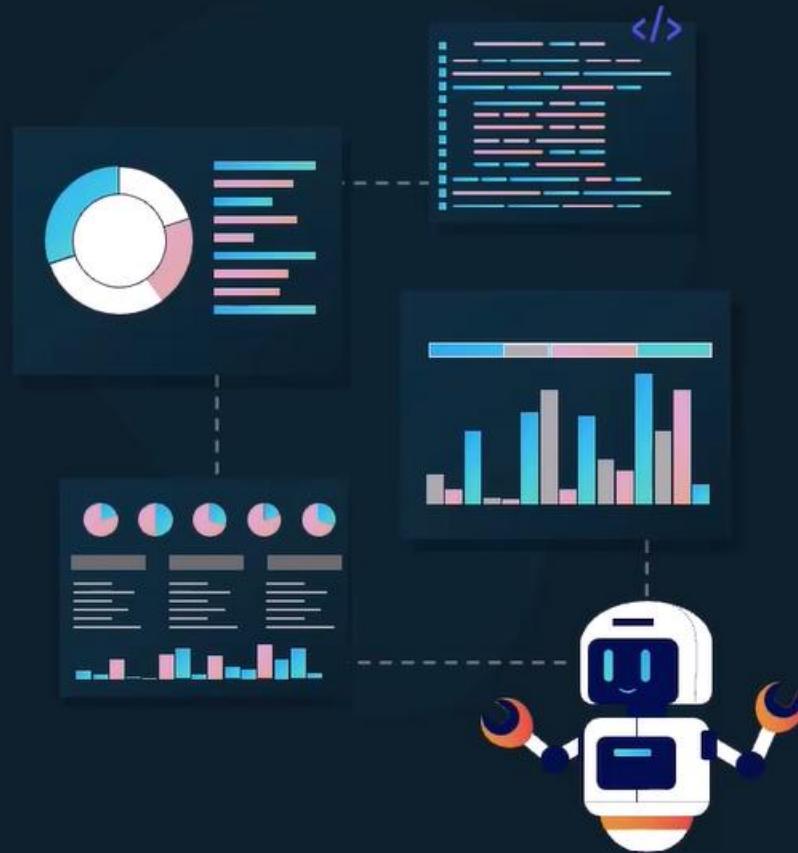


Reduce the need for  
repeated clarifications



Improve overall  
development efficiency

# Understanding Context – Definition



Context

Information provided to AI model

Guides understanding and improves output quality

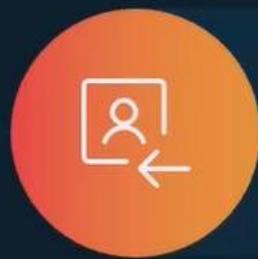
Takes many forms of data  
E.g., code files, selections, terminal output/other relevant information

# Types of Context



## Calculated Context

Automatically determined by extension



## Explicit Context

Directly specified by user to ensure Copilot considers specific information



## Implicit Context

Automatically added by Copilot, based on patterns it recognizes in the project

```
@workspace /explain  
↳ word_counter_gui.py:28-30
```

## Workspace

> Used 2 references

Let's break down this code which handles file selection in a GUI application:

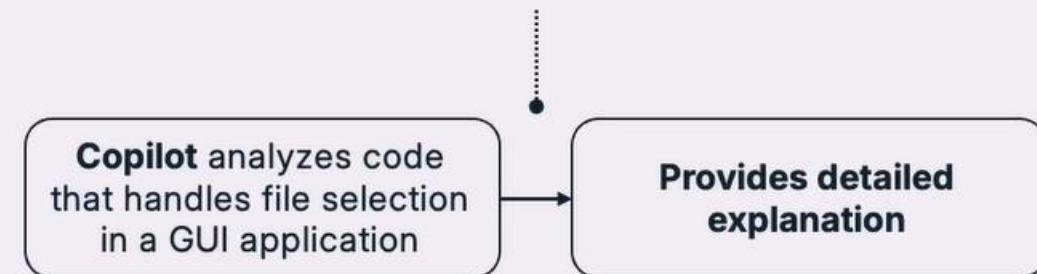
The `select_file` method is part of a class (likely a GUI window or frame) and provides a way for users to select a file through a graphical interface. Here's what's happening:

1. The method uses `filedialog.askopenfilename()`, which is a built-in Tkinter dialog that opens a standard operating system file picker window.
2. The `filetypes` parameter is set with a list of tuples that define which file types should appear in the file picker:
  - `("Text Files", "*.txt")` shows text files with .txt extension
  - `("All Files", "*.*")` shows all file types
3. When called, this dialog will:
  - Block the main window until the user makes a selection or cancels
  - Return the complete file path as a string if a file is selected
  - Return an empty string if the user cancels the operation
4. The selected file path is stored in `self.filename` for later use in the application.

# Calculated Context

Automatically determined based on relevance

Example: @workspace analyzes prompt and selects relevant files



```
filetypes=[('text files', '*.txt'), ('All files', '*')]

def plot(self):
    if self.filename:
        for word in words:
            lot_frame.winfo_children()[-1].append(word)

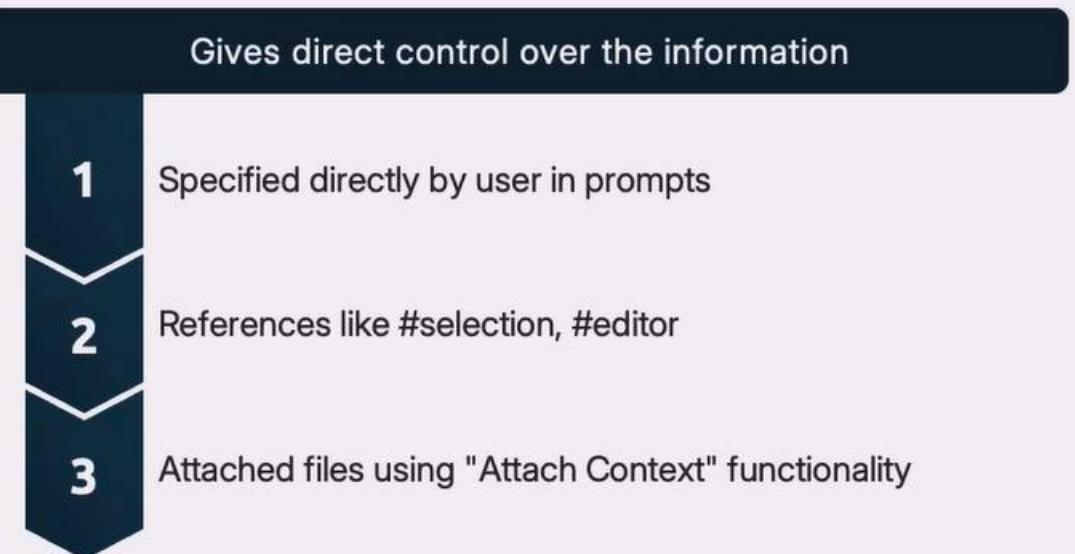
    Go to Definition F12
    Go to Declaration
    Go to Type Definition
    Go to Implementations Ctrl+F12
    Go to References Shift+F12
    Peek > e(self.filename)
    ext)
    words(words)
    (word_count, int(self.filename))

Find All References Shift+Alt+F12
Find All Implementations
Show Call Hierarchy Shift+Alt+H
Show Type Hierarchy

Copilot >
    Editor Inline Chat Ctrl+Shift+I
    Explain
    Fix
    Review and Comment
    Generate Docs
    Generate Tests
    Add Selection to Chat
    Add File to Chat
    Add Selection to Copilot Edits
    Add File to Copilot Edits
    Cut Ctrl+X
    Copy Ctrl+C
```

# Explicit Context

Gives direct control over the information



# Implicit Context

01



Automatically added  
by Copilot

02



Includes current file  
and selection

03



Can be disabled in  
configurations

# Context Sources

## Code files and project structure

Provide information about your codebase's organization and relationships

## Current editor selection

Shows you are actively working

## Terminal output

Provides error messages or command results

## Source control changes

Inform Copilot about recent modifications to your codebase

The screenshot shows a code editor window with a dark theme. The file being edited is named 'plot.py'. The code contains a function definition:40 self.plot\_word\_count(word\_count, int(self.n\_words.get()))
41
42 def plot\_word\_count(self, word\_count, n):
43 = sorted(word\_count.items(), key=lambda x: x[1], reverse=True)
44
45 fig, ax = plt.subplots(figsize=(10, 6))
46
47A tooltip is displayed over the line 'fig, ax = plt.subplots(figsize=(10, 6))'. The tooltip has a yellow background and contains the following text:

Enter to Apply, Ctrl+Enter to Preview

which

- o (
- o (

3. When

- o Bl
- ca
- o Re
- o Re

4. The se

the app

The tooltip also includes a small icon of a person with a gear and a lightning bolt.

# Context Referencing – Chat Participants (Agents)

**@workspace**

Understands project structure and code relationships

**@terminal**

Helps with terminal commands and shell operations

**@vscode**

Specializes in VS Code features and commands

**@github**

Provides GitHub-specific skills

**@azure**

Assists with Azure services and deployments

```
ter_gui.py 2 #plot
dCountApp
    ⚭ plot WordCountApp • main.py
ntApp:
    ⚭ plot WordCountApp • word_counter_gui.py
Filename
    ⚭ plot_button WordCountApp • main.py
    ⚭ plot_button WordCountApp • word_counter_gui.py
:_file(s
Filename
iletypes
')] )
    ⚭ plot_frame WordCountApp • main.py
    ⚭ plot_frame WordCountApp • word_counter_gui.py
    ⚭ plot_word_count main.py
    ⚭ plot_word_count WordCountApp • word_counter_gui.py

self):
    f.filename:
        or widget in self.plot_frame.winfo_children():
            widget.destroy()

text = read_text_file(self.filename)
ords = clean_text(text)
ord_count = count_words(words)
elf.plot_word_count(word_count, int(self.n_words.get()))
)

word_count(self, word_count, n):
ords = sorted(word_count.items(), key=lambda x: -x
reverse=True)[ :n]

Figure(figsize=(10, 6))
```

# Context Referencing – File and Symbol References

- 1 **#filename** Points Copilot to specific files in your project
- 2 **#symbol** Targets specific code elements like functions, classes, or variables
- 3 **#sym** Opens Quick Pick selector for symbols

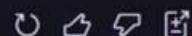
# Context Referencing – Selection References

## Help Copilot Focus on Specific Portions of Your Code

### Key improvements:

- Increased figure size for better readability
- Added value labels on top of each bar
- Added grid lines for easier reading
- Changed bar color to a more visually appealing blue
- Adjusted text rotation to 45° with right alignment
- Added `tight_layout()` to prevent label cutoff
- Changed the bar label placement for better visibility

These changes will make the visualization more informative and professional-looking.



What other visualizations can we add?

Ask Copilot

word\_counter\_gui.py:42-46 Current file ↗

@ 🎧

Claude 3

**#selection**

Directs Copilot to consider current editor selection

**#editor**

Includes visible contents of active editor

Promotes an inline chat to preserve context

# Context Referencing – Terminal and Source Control

## #codebase

Provides relevant workspace content, giving Copilot a broader view of the project

## #changes

Includes source control changes

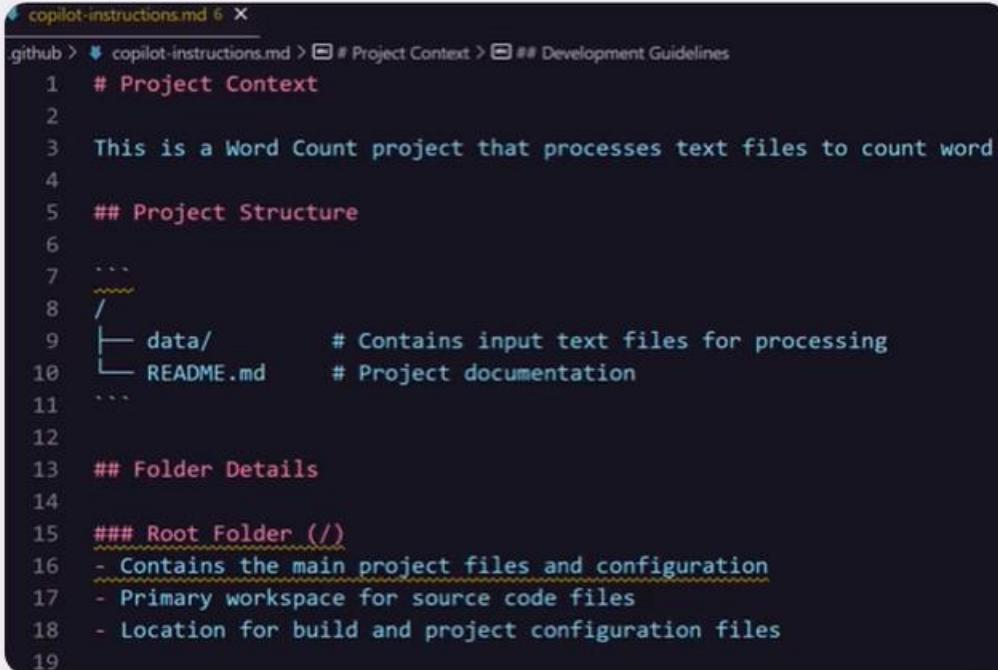
## #terminalLastCommand

Shows the last command run in the terminal

## #terminalSelection

Includes terminal selection

# Advanced Techniques – Custom Instructions



```
copilot-instructions.md 6 X
github > copilot-instructions.md > # Project Context > ## Development Guidelines
1 # Project Context
2
3 This is a Word Count project that processes text files to count word
4
5 ## Project Structure
6
7 ...
8 ...
9   data/      # Contains input text files for processing
10  README.md  # Project documentation
11 ...
12
13 ## Folder Details
14
15 ### Root Folder (/)
16 - Contains the main project files and configuration
17 - Primary workspace for source code files
18 - Location for build and project configuration files
19
```

Repository-level instructions can be defined in  
**File:** (.github/copilot-instructions.md)

Create purpose-specific instructions for different tasks  
**E.g.,** Code generation, documentation, or testing

Important to balance context volume

With too little context, Copilot lacks the necessary information,  
and too much context will overwhelm it with irrelevant details