
GitHub Copilot Training
for Developers
Andrew Scoppa

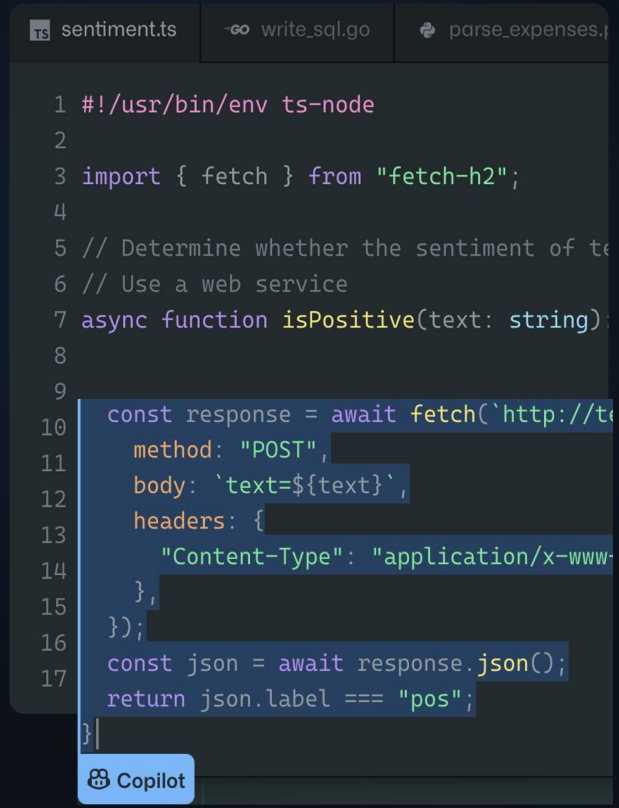
AGENDA

- GitHub Copilot - Introduction
- Best practices & prompt engineering
- In-class coding demos using copilot and copilot chat
- Secure coding
- Wrap-up, Q&A

GitHub Copilot Introduction

Let’s start with a high-level overview of GitHub Copilot

- GitHub Copilot is there to enhance daily work
  - It’s an intelligent “pair-programmer”
  - Like a smart assistant or mentor by your side
- Draws context from text & code in open tabs
- Powered by OpenAI
  - Copilot uses a transformative model
  - Think of something like Google Translate
- Trained on large datasets to ensure accuracy
  - It even can help with HTML and markdown!
- Copilot generates new code in a probabilistic way, and the probability that it produces the same code as a snippet that occurred in training is low.



Copilot vs Copilot Chat

Copilot

Direct Code Writing

Your “coding assistant”

Solo Development

Copilot Chat

In-Depth Interactive Assistance

Your “research assistant”

Collaborative Scenarios

GitHub Copilot + Chat

Helps developers stay in the flow throughout the entire SDLC

Using Different LLM Models with Copilot

- Several models to choose from including

GPT 5 - Next-generation successor to GPT-4o, geared toward deeper multi-step reasoning, tool orchestration, and extended trusted context windows.

Claude 3.5 Sonnet - Balanced model delivering strong reasoning, long-context handling, and fast iterative responses at moderate cost.

o3-mini (OpenAI) - Lightweight reasoning-optimized model variant focused on fast iterative problem solving (code, logic puzzles, structured planning) with lower latency and cost versus larger flagship models.

- Be aware of costs

Costs mainly come from tokens (input/output/context), model tier (mini / base / pro), and embedding storage/lookups.

Cut them by shortening prompts, limiting context, caching, and choosing right-sized models.

GitHub Models

VS Code Chat Participants

Chat participants are AI domain experts that can perform tasks or answer questions in a specific domain. Chat partcipants include:

- **@workspace**: Has context about the code in your workspace. Use @workspace when you want Copilot to consider the structure of your project, how different parts of your code interact, or design patterns in your project.
- **@vscode**: Has context about VS Code commands and features. Use @vscode when you want help with VS Code.
- **@terminal**: Has context about the VS Code terminal shell and its contents. Use @terminal when you want help creating or debugging terminal commands.
- **@azure**: Has context about Azure services and how to use, deploy and manage them. Use @azure when you want help with Azure. Requires Azure connection
- **@github**: Allows you to use GitHub-specific Copilot skills.

Asking GitHub Copilot questions in your IDE

VS Code Slash Commands

Use slash commands in chat mode to avoid writing complex prompts for common scenarios. Slash commands include:

/doc: generate code documentation (Inline Chat)

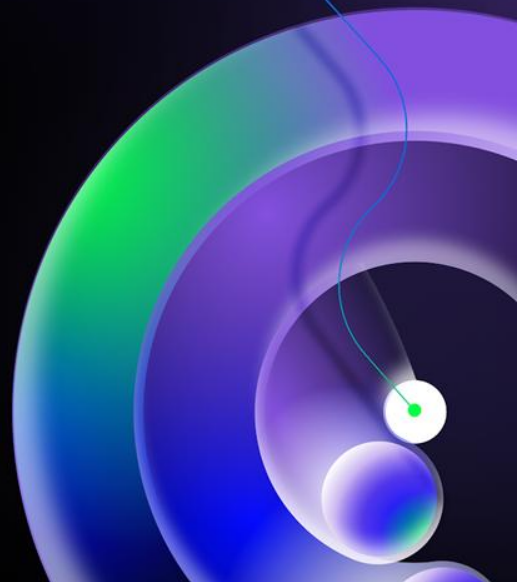
/explain: explain how the selected code works

/fix: propose a fix for the problems in the selected code

/test: generate unit tests for the selected code

/new: scaffold code for a new workspace or new file

/newNotebook: create a new Jupyter Notebook



VS Code Chat Variables

Use chat variables to include specific context in your prompt. To use a chat variable, type # in the chat prompt box, followed by a chat variable. Chat variables include:

- #selection - The current selection in the active editor
- #codebase - Searches through the codebase and pulls out relevant information for the query.
- #editor - The visible source code in the active editor
- #terminalLastCommand - The active terminal's last run command
- #terminalSelection - The active terminal's selection
- #file - Choose a file in the workspace

Prompt Engineering

What is a Prompt?

In the context of Copilot, a prompt is a piece of code or natural language description that is used to generate code suggestions. It is the input that Copilot uses to generate its output.

- The Copilot Team

What is Prompt Engineering?

Prompt engineering is the process of designing and creating high-quality prompts that can be used to generate accurate and useful code suggestions with Copilot.

- The Copilot Team

IMO: It is basically tips and tricks to help make your prompts more reliable. Most of the time when you have traditional prompts or one off prompts that you use and throw away, you don't really need prompt engineering. But once you introduce this into production, you want this to be much more reliable. You want to be able to predict how that response will be formatted and look.

Prompting Best Practices

Write Clear Instructions

Models can’t read minds
Long outputs? Ask for brief
Too simple outputs? Ask for expert-level writing
Did not like the format? Exemplify the output

Tactics:

- Include details on prompt
- Ask the model to be a persona (“act like a senior Java engineer)
- Use delimiters to separate parts of prompts
- Specify steps required to complete the task
- Provide examples
- Say how long the output should be

Provide References

References reduce chances of a confident wrong answer when asked about citations or URLs

Tactics:

- Instruct to answer using reference text
- Instruct to answer with citations from a reference text

Prompting Best Practices

Split Complex into Simpler Tasks

Just like Software Engineering, solving a bigger problem is disproportionately harder than combining the solution of smaller, easier problems.

The bigger the tasks, the higher the error rates

Tactics:

- Summarize or filter previous dialogue
- Construct full summaries of the code base

Give the model time to think

Prefer to ask for a thought process instead of straight answer

This gives the model time to “reflect” on the response and converge to a better output

Tactics:

- Instruct to work out a solution instead of rushing to conclusion
- Construct the context with snippets of information from the whole context to start guiding the model

Prompting Best Practices

Use External Tools

Copilot and other Large Language Models are not the best tools for all jobs

If a task can be done more reliably by another tool, use it and feed the results into Copilot

Tactics:

Sometimes a calculator is faster at math than a LLM

Give the model references to access contextual information (although Copilot does not yet have capabilities of browsing the web, the model it was trained on might have references to the same link or website)

Test Changes Systematically

Just like code, the output of LLMs should be tested constantly

The earlier and the smaller the test scope, the easier to spot and fix the problems

Tactics:

Deal with the output like if it was a function that needs unit testing to sparkle confidence

Copilot Across the SDLC

“I am an experienced software engineer and would like an overview how GitHub Copilot can be integrated into the software development life cycle.

- Provide a detailed explanation of how Copilot can assist in each phase of the SDLC, including planning, design, development, testing, deployment, and maintenance.

- Include specific examples of tasks or activities within each phase where Copilot can be particularly beneficial.”

Providing Editor Context to Copilot

To help Copilot generate accurate suggestions:

- Add a top-level comment block describing the purpose of the file
- Front load as many imports as needed (import / include / requires / etc.)
- Create a detailed comment block describing the purpose of an operation or UDT
- Use sample code as a starting point
- Have related content open in other tabs

Strategic Prompt Engineering

In your IDE you enter the following comment in a C++ source file:

```
/*
Create a factorial function
*/
```

Assume the following response is returned by Copilot:

```
int factorial(int n) {
    int result = 1;
    for(int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

Strategic Prompt Engineering

- While that is correct, it’s possibly not what you want!
- Instead:
- - You would like the function to be named named ‘fact’, have an
- unsigned int argument ‘n’, and return an unsigned long long
- - You also want the function to be recursive

Strategic Prompt Engineering

To assist Copilot in building a context to work with you:

- Add an overview comment at the top of the file.

- Include the necessary header files you may be requiring.

```
/*
This file contains a recursive implementation of a factorial function in C.

The factorial function is defined as:
- 0! = 1
- n! = n * (n - 1) * (n - 2) * ... * 1

The file also contains functions to calculate the number of permutations
of 'n' items taken 'r' at a time, and a memoized version of the permutation function.
*/
#include <stdio.h>
#include <assert.h>
```

Strategic Prompt Engineering

Rewrite the function comment providing as much detail as necessary:

```
Define a factorial function 'fact' that takes an unsigned integer 'n'
and returns an unsigned long long.

The function should use recusion.

Implementation:
- If n == 0 or n == 1, return 1
- return n * fact(n - 1)
```

Strategic Prompt Engineering

Now the resulting response is acceptable:

```
unsigned long long fact(unsigned int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    return n * fact(n - 1);
}
```

Strategic Prompt Engineering

- Use Copilot to build additional functions that use factorial:

```
Create a function 'perm' that calculates the number of
permutations of 'n' items taken 'r' at a time.
- 'n' and 'r' are non-negative integers.
- The return value should be an unsigned long long.

Use the 'fact' function to calculate the factorial of 'n' and 'n - r'.
Return the result of n! / (n - r)!
```

- The response will likely be the following code:

```
unsigned long long perm(unsigned int n, unsigned int r) {
    return fact(n) / fact(n - r);
}
```

Helpful Patterns

* Use descriptive variable names to make your intentions clear.

```
totalSampleCount = 1000
```

* Maintain consistent naming conventions for variables and functions.

```
i.e. using camelCase for variable names consistency
```

* Define method signatures with unambiguous parameter names and types.

```
double calculateAverageSampleSize(unsigned long samples[], size_t size)
```

* Use comments to explain the purpose of the code.

```
This function 'palindrome' takes an array of strings
as input and returns  true if a palindrome is found.
```

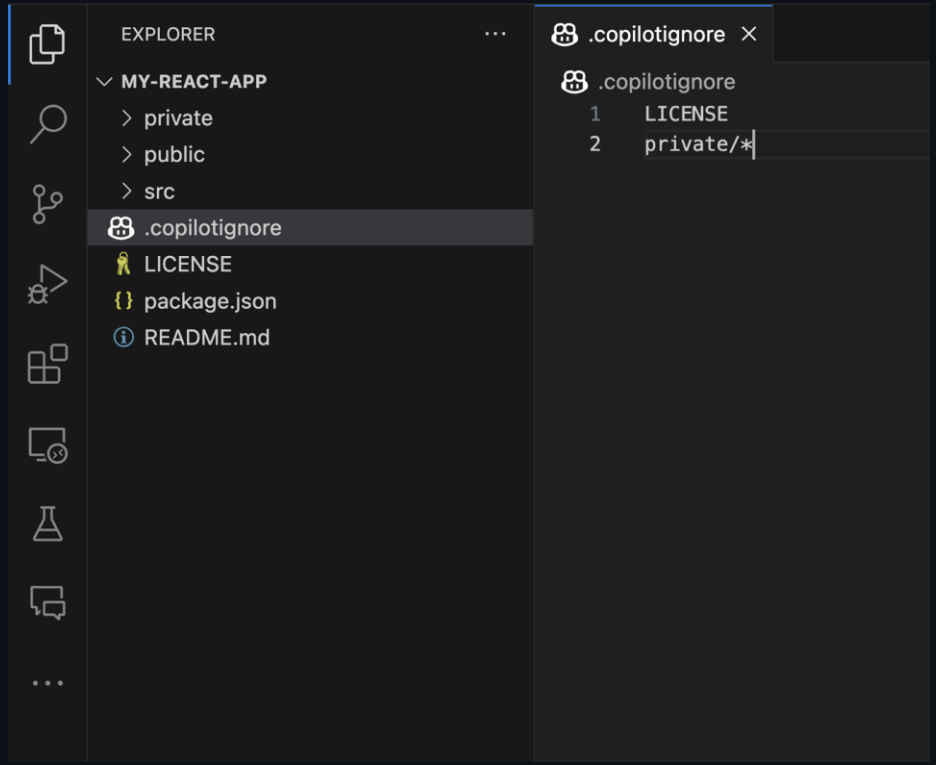
Helpful Patterns

```
* Specify error handling scenarios.

    Exit where the input is NULL and throw and
    exception if the input out of range [min, max] inclusive.

* Describe control flow structures.

    Write a while loop to print the first 'n' values in the fibonnaci sequence.

* Show examples of how to use the code.

    int samples[] = {1, 2, 3, 4, 5};
    double average = calculateAverageSampleSize(samples, 5);
    printf("Average: %f\n", average); // Average: 3.0

* Test your code.

    Write a test suite to verify the correctness of the code.
```

Secure coding

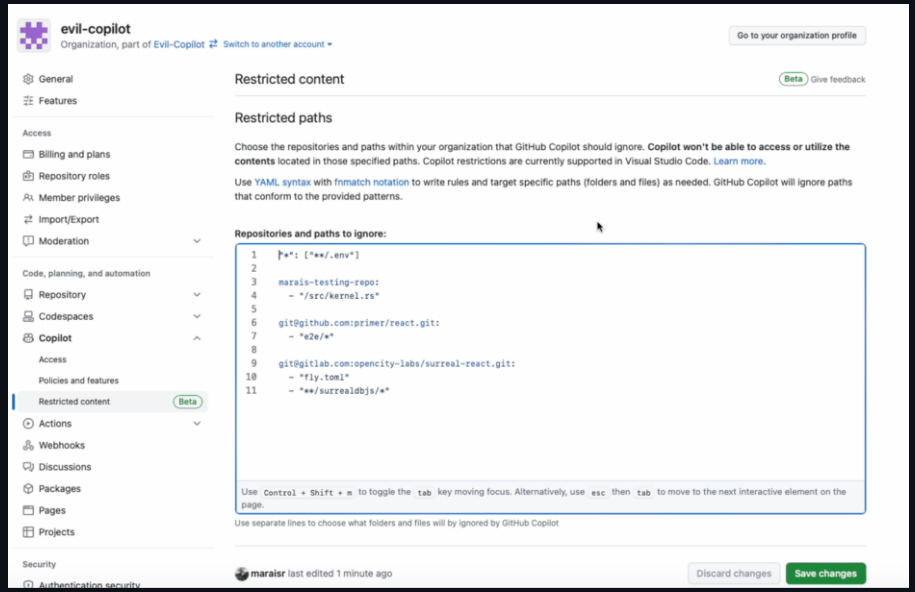
Block files from Copilot

Use .copilotignore to block files and folders from being used by Github Copilot



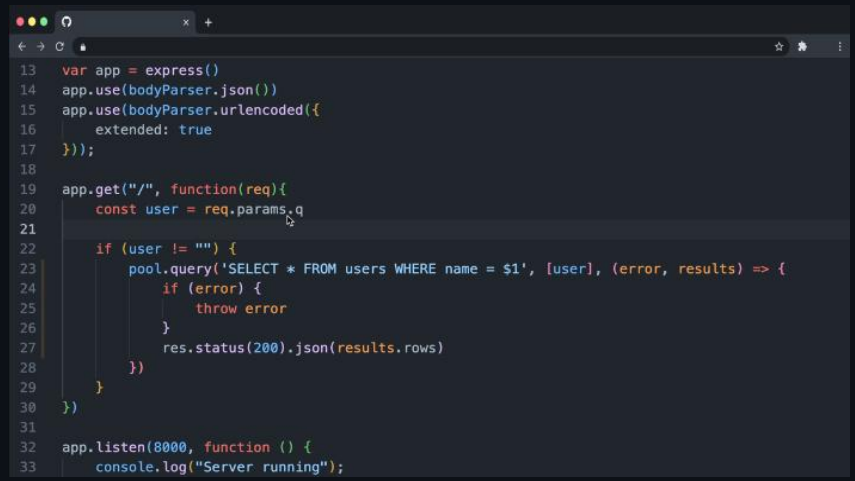
Block files from Copilot

- The Copilot Restricted Content feature allows repository owners to control the usage of Copilot across their codebases.
- This includes specific files, folders, or entire repositories, even if they're not hosted on GitHub.
- The goal is to provide CfB customers a flexible way of controlling what content Copilot can access for prompt crafting and where it can insert code.



Copilot and Secure Coding

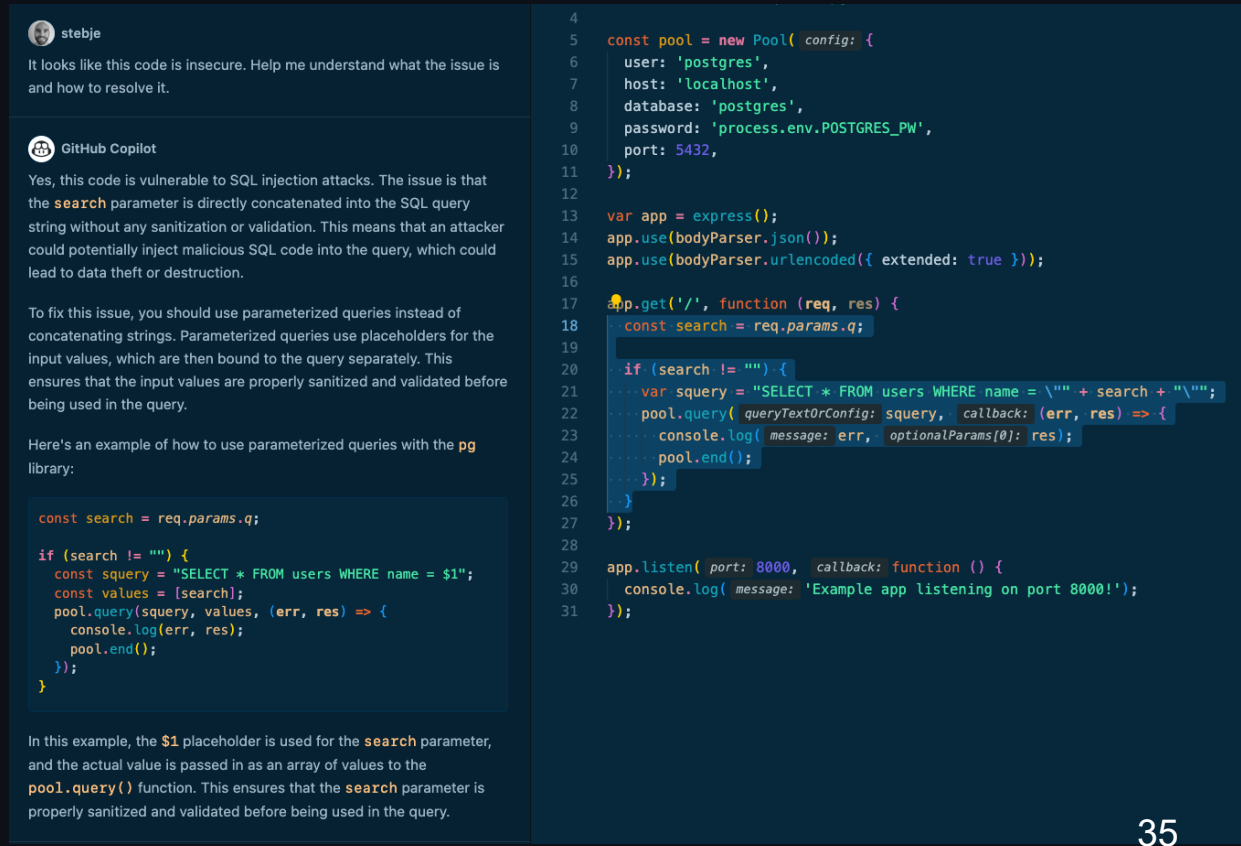
- AI-based vulnerability system that helps prevent insecure coding patterns (e.g. SQL script injection)
- Vulnerability filters are applied only to the Copilot-generated suggestion itself
  - ⊗ It cannot detect downstream vulnerabilities introduced by the code e.g. on deployment infrastructure
  - ⓘ **We recommend taking the same precautions you take with code written by engineers (linting, code scanning, etc.)**
- Copilot Chat can be used to query code for known vulnerabilities



```
13  var app = express()
14  app.use(bodyParser.json())
15  app.use(bodyParser.urlencoded({
16      extended: true
17  }));
18
19  app.get("/", function(req){
20      const user = req.params.q
21
22      if (user != "") {
23          pool.query('SELECT * FROM users WHERE name = $1', [user], (error, results) => {
24              if (error) {
25                  throw error
26              }
27              res.status(200).json(results.rows)
28          })
29      }
30  })
31
32  app.listen(8000, function () {
33      console.log("Server running");
```

Detection & Remediation

- Prompt for insecure coding patterns
- Use GHAS Code Scanning results
- Write custom CodeQL queries
- Increase your knowledge of secure coding patterns
- Create custom Secret Scanning patterns



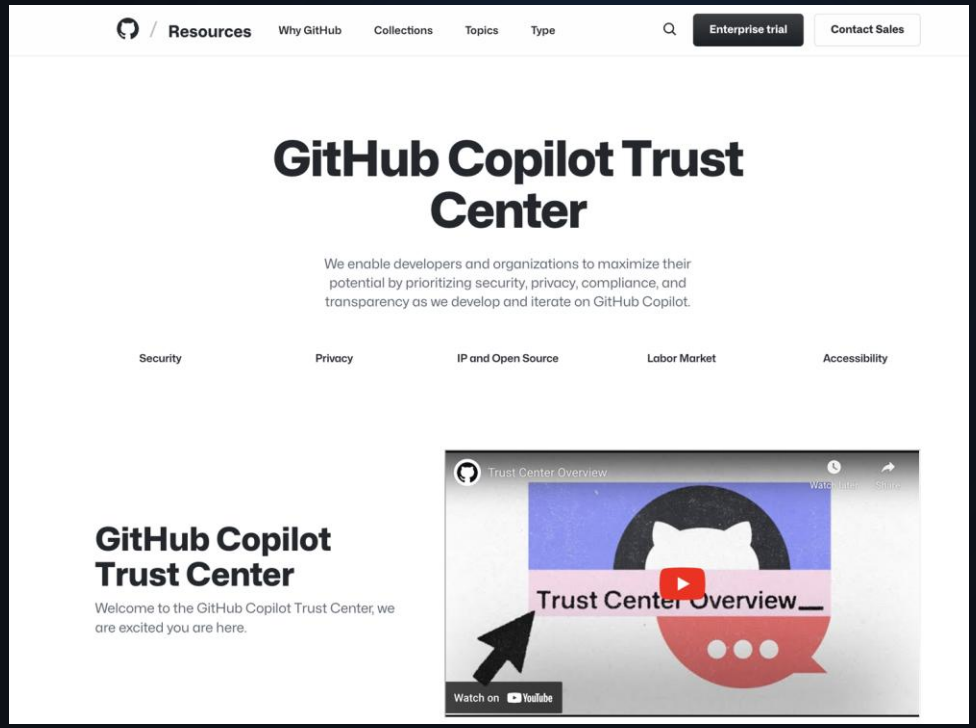
Copilot + GHAS

- Copilot is not a replacement of GHAS features
- Copilot can be used in tandem with GHAS features to detect and remediate vulnerabilities earlier during the SDLC
  - GHAS Code scanning results
  - GHAS Secret scanning

Security & Trust

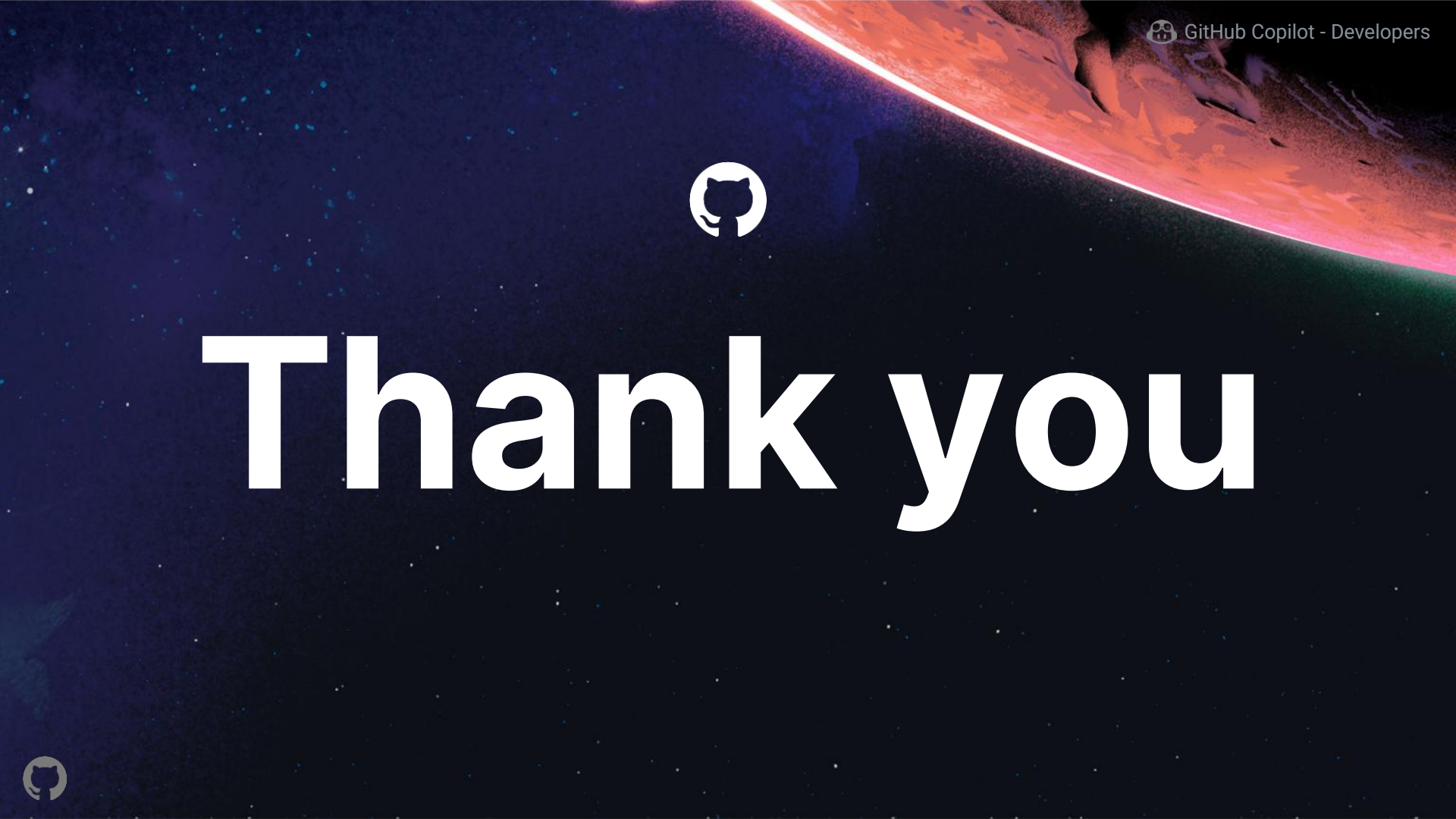
Copilot Trust Center

- Security
- Privacy
- Data flow
- Copyright
- Labor market
- Accessibility
- Contracting



GitHub Copilot Trust Center - GitHub Resources

Wrap Up


Thank you