GitHub Copilot Developers Training

Andrew Scoppa

GitHub Copilot

Resources

- Getting started with GitHub Copilot
- Configuring GitHub Copilot in your environment
- Using GitHub Copilot Chat in your IDE - GitHub Docs
- Video - Get Started with the Future of Coding: GitHub Copilot
- Tutorial: GitHub Copilot and VS Code
- Copilot Exercises

Choose Workshop Track

Workshop for those who want to see how to build web application

Primary IDE: VS Code
Primary OS: Independent

Workshop for those who want to see how to build Java applications

Primary IDE: IntelliJ
Primary OS: Independent

Workshop for those who want to see how to build .NET applications

Primary IDE: Visual Studio IDE
Primary OS: Windows

AGENDA

GitHub Copilot - Introduction

Coding

Best practices & prompt engineering

Workshop (1 - 2 hours long)

Secure coding

Wrap-up, Q&A

Outcome of this training

You will achieve...

- Get answers to specific use case scenario questions
- Increase existing Copilot skills by following a specific workshop tutorial catered to your needs
- Learn in-depth tips and tricks and best practices on how to best utilize GitHub Copilot

Covered in Copilot Fundamentals

We will not talk about...

- ❌ Statistics around Copilot usage and satisfaction
- ❌ Successful customer case study
- ❌ Enterprise & Organization administrator interface

GitHub Copilot Fundamentals Recap

The software process can be broken down into two steps:

1) Design

2) Implementation

The first step is driven by you.

The second step is where Copilot can assist you with the development effort.

Design

IMPLEMENTATION

GitHub Copilot

Helps developers stay in the flow throughout the entire SDLC

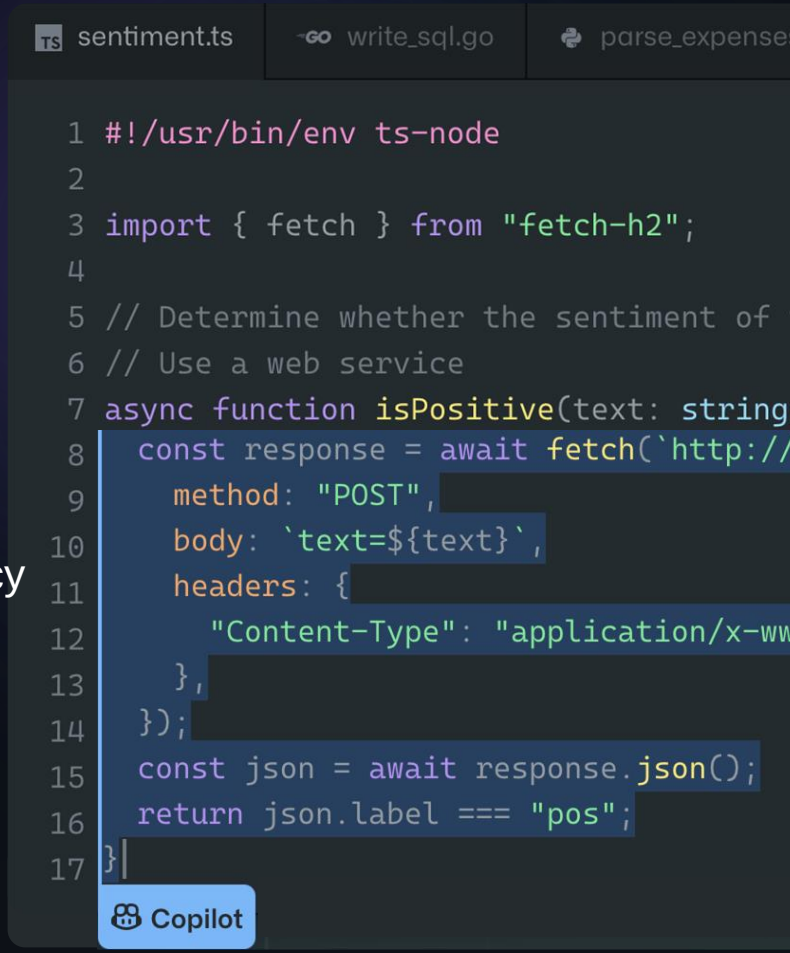
Refactoring code (code translate)
Reviewing code (code explain)
Documentation

Unit testing (TDD and BDD)
Finding code errors
Debugging
Code review
AI Pull Requests

Convert comments to code
Autofill for repetitive code
Show alternatives

GitHub Copilot

- An intelligent pair programmer
- Draws context from comments & code in open tabs to suggest individual lines and whole functions
- Powered by OpenAI Codex
  - Copilot uses a transformative model
  - Trained on large datasets to ensure accuracy
- Available as extensions to popular IDEs
- Programming Languages and Technology available in Public code base all are supported



Data flow through the Copilot ecosystem

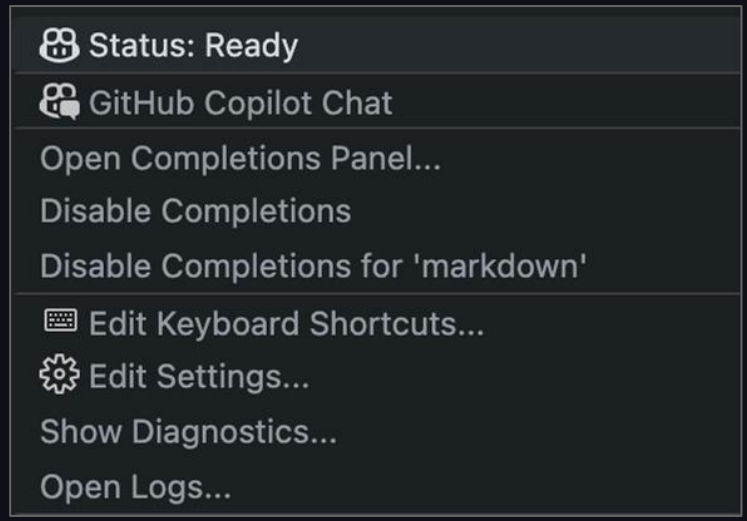
PII
Toxicity
Code classifier

CODE EDITOR | **PROXY** | **MODEL**

PII
Toxicity
Code classifier
Code quality
Duplicate detection

VS Code Settings

Configuration Options



Configuring GitHub Copilot in your environment

Coding and Prompt Engineering

Copilot vs Copilot Chat

Copilot	Copilot Chat
Direct Code Writing	In-Depth Assistance
Seamless IDE Integration	Learning & Teaching
Solo Development	Collaborative Scenarios

Copilot Prompt Engineering

What

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.

Why

- Maximizes utility of Copilot
- Minimizes repeated iterations
- Determines code suggestion quality
- Skill prompts guide Copilot to understand context & nuances
- Refined interactions reduces misunderstandings

How

- Neighboring Tabs
- Zero-Shot Prompting
- One-Shot Prompting
- Few-Shot Prompting
- Let's Think Step by Step

Prompting Best Practices

Improving results

These are best practices, now let’s see how those can be used in different ways

Provide references
Improve relevance of the response by providing an example and context

Write clear instructions
Refine your prompt, provide context, write clearly, and give Copilot ample input for better results

Split up big tasks
Breaking down complex tasks minimizes errors and utilizes previous outcomes for efficiency

Allow time to think
Requesting Copilot’s thought process will enhance Copilot accuracy, but it may prolong wait times.

Test changes systematically
Measure performance, watch for prompt changes' side effects, and use test suites for implementation

Providing Context

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.)
- Use descriptive function / method / type names
- Use sample code as a starting point
- Have related content open in other tabs

Helpful Patterns

* Use descriptive variable names to make your intentions clear.

```
totalSampleCount = 1000
```

* Define method signatures with unambiguous parameter names and types.

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

* 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
```

* Maintain consistent naming conventions for variables and functions.

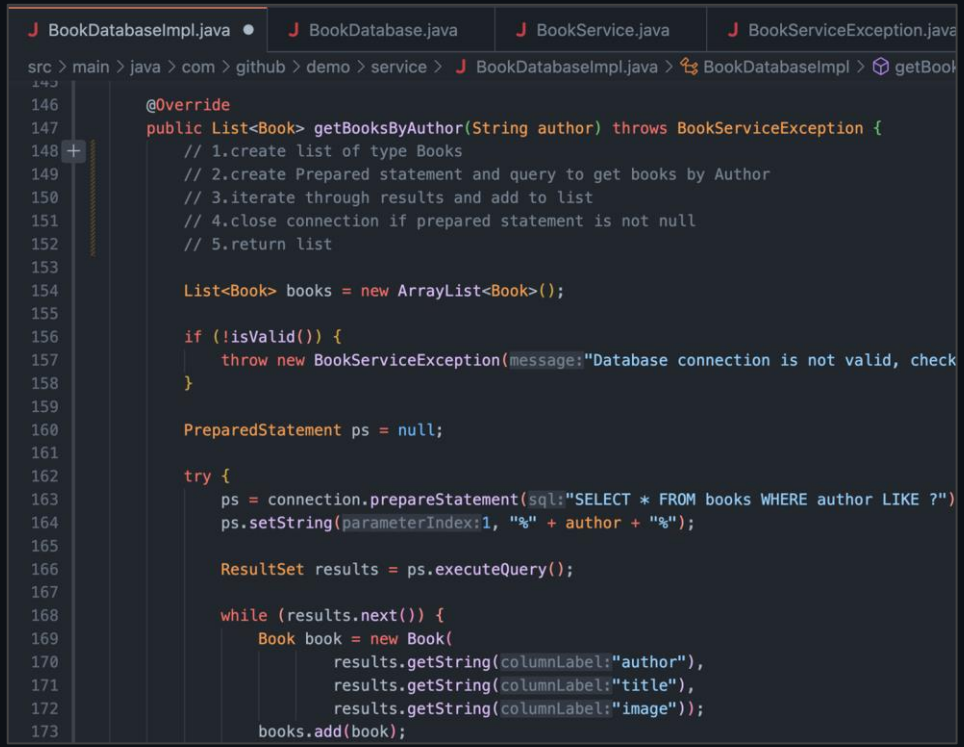
```
i.e. using camelCase for variable names consistency
```

Helpful Patterns

```
* 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.

* 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.

* Test your code.

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

Neighboring Tabs

Keep relevant files open, things related to your current code



If at first you don't succeed, iterate!

Zero Shot Prompt

```
//Write a JavaScript function that finds
//the maximum value in an array.
function max(array) {
    return Math.max.apply(null, array);
}
```

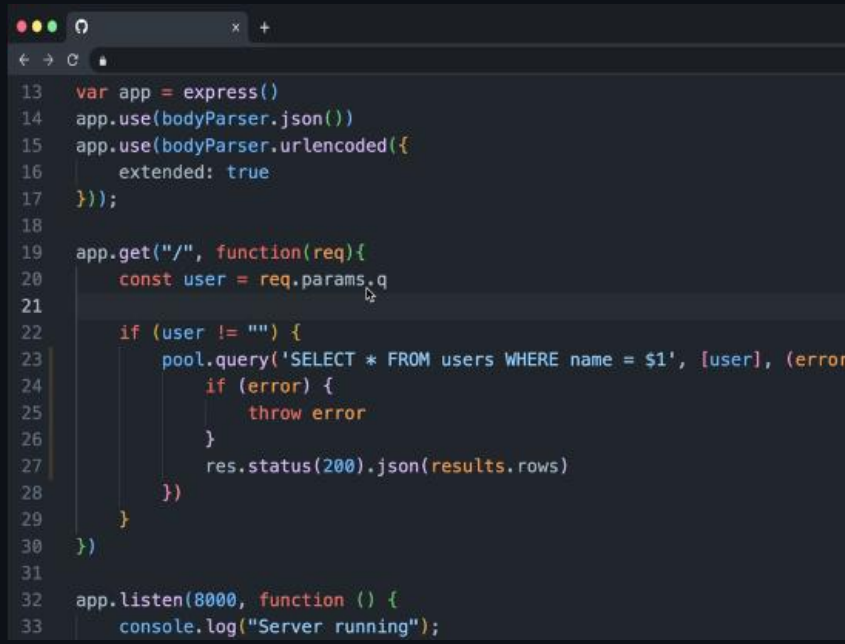
One-Shot Prompt

```
//Create a JavaScript function, `findMax`, that takes an array of strings as input
//and returns the string with the maximum length.
//Ensure that the function works correctly for arrays that are empty,
//as well as arrays that contain both strings and numbers.
function findMax(array) {
    var max = array[0].length;
    array.map(v => max = Math.max(max, v.length));
    result = array.filter(v => v.length == max);
    return result;
}
```

Secure coding

Copilot and Secure Coding

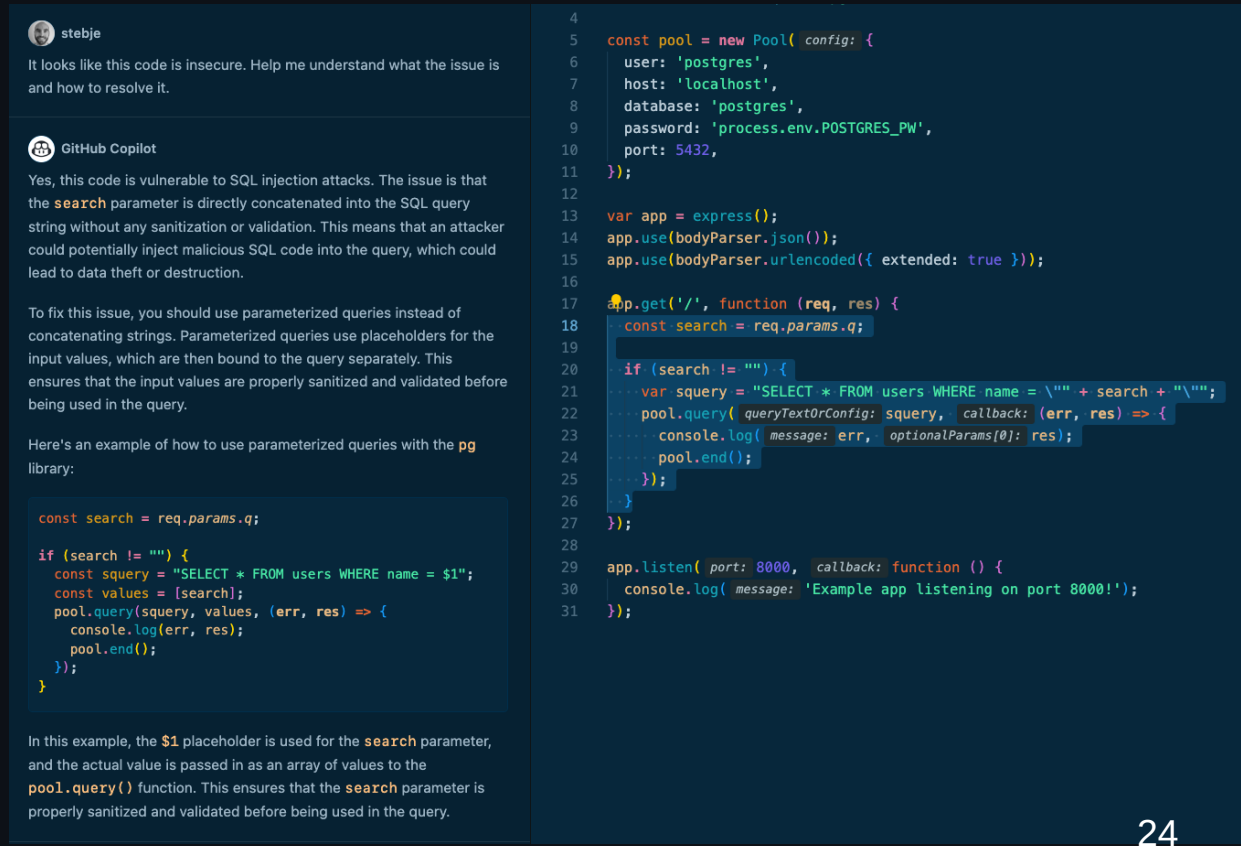
- AI-based vulnerability system that helps prevent insecure coding patterns (e.g. SQL script injection)
- Vulnerability filters are applied only to the Copilot-generated suggestion itself
  - ⊗ It cannot detect downstream vulnerabilities introduced by the code e.g. on deployment infrastructure
  - ⓘ **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
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



stebje

It looks like this code is insecure. Help me understand what the issue is and how to resolve it.

GitHub Copilot

Yes, this code is vulnerable to SQL injection attacks. The issue is that the `search` parameter is directly concatenated into the SQL query string without any sanitization or validation. This means that an attacker could potentially inject malicious SQL code into the query, which could lead to data theft or destruction.

To fix this issue, you should use parameterized queries instead of concatenating strings. Parameterized queries use placeholders for the input values, which are then bound to the query separately. This ensures that the input values are properly sanitized and validated before being used in the query.

Here's an example of how to use parameterized queries with the `pg` library:

```
const search = req.params.q;

if (search != "") {
  const squery = "SELECT * FROM users WHERE name = $1";
  const values = [search];
  pool.query(squery, values, (err, res) => {
    console.log(err, res);
    pool.end();
  });
}
```

In this example, the `$1` placeholder is used for the `search` parameter, and the actual value is passed in as an array of values to the `pool.query()` function. This ensures that the `search` parameter is properly sanitized and validated before being used in the query.

```
4
5  const pool = new Pool( config: {
6    user: 'postgres',
7    host: 'localhost',
8    database: 'postgres',
9    password: 'process.env.POSTGRES_PW',
10   port: 5432,
11 });
12
13 var app = express();
14 app.use(bodyParser.json());
15 app.use(bodyParser.urlencoded({ extended: true }));
16
17 app.get('/', function (req, res) {
18   const search = req.params.q;
19
20   if (search != "") {
21     var squery = "SELECT * FROM users WHERE name = \"" + search + "\"";
22     pool.query( queryTextOrConfig: squery, callback: (err, res) => {
23       console.log( message: err, optionalParams[0]: res);
24       pool.end();
25     });
26   }
27 });
28
29 app.listen( port: 8000, callback: function () {
30   console.log( message: 'Example app listening on port 8000!');
31 });
```

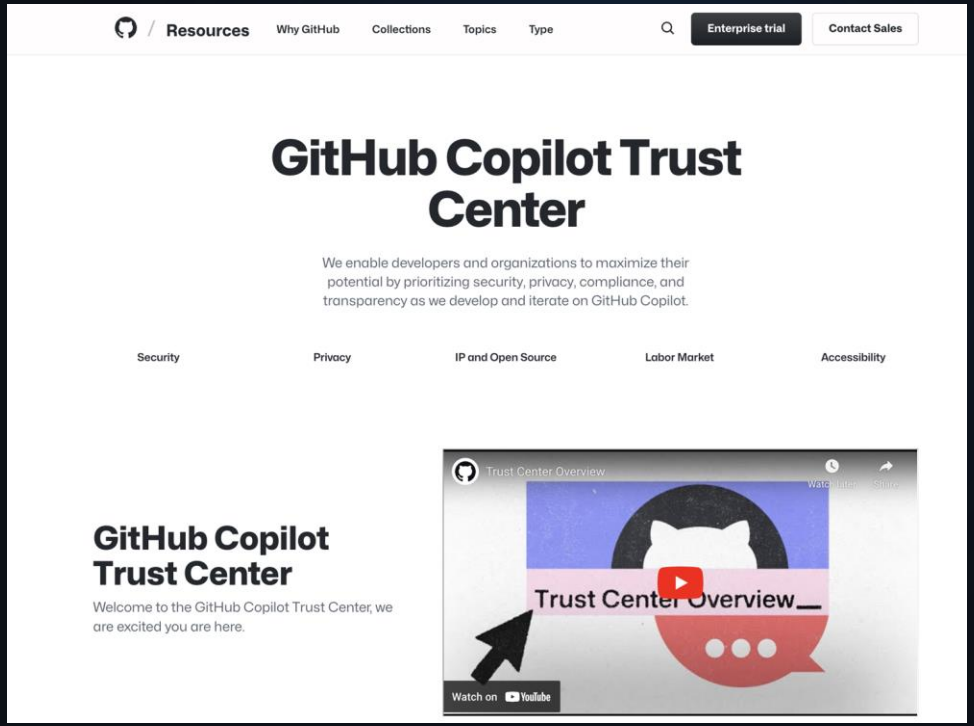
Copilot + GHAS

- 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

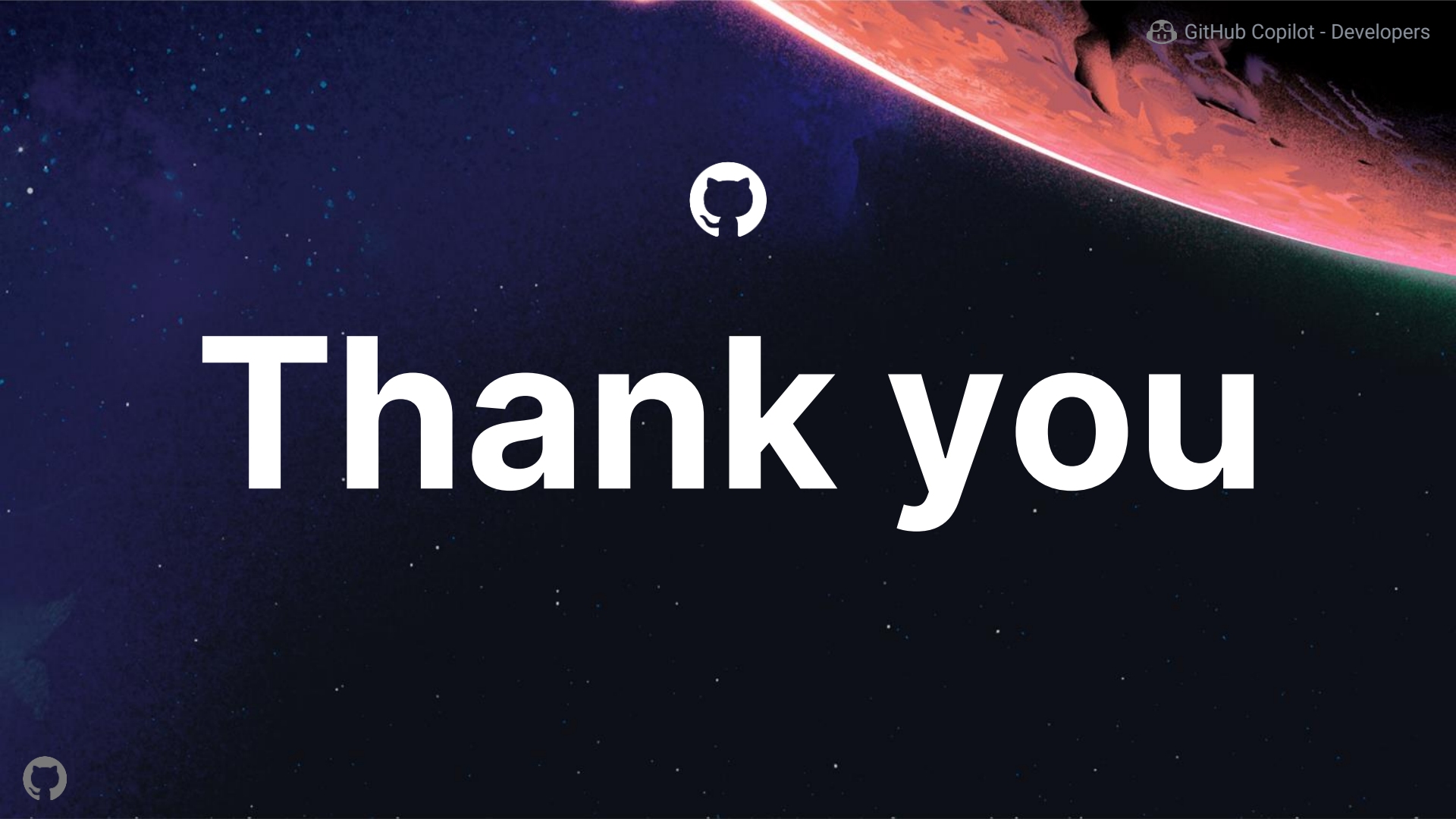
Security & Trust

Copilot Trust Center

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



Wrap Up


Thank you