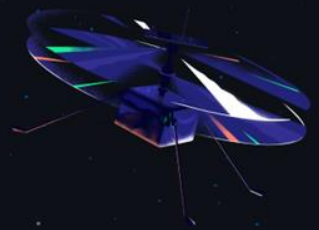
GitHub Copilot for QA Professionals

Andrew Scoppa



AGENDA

GitHub Copilot - Introduction

Prompt Engineering

Quality Assurance

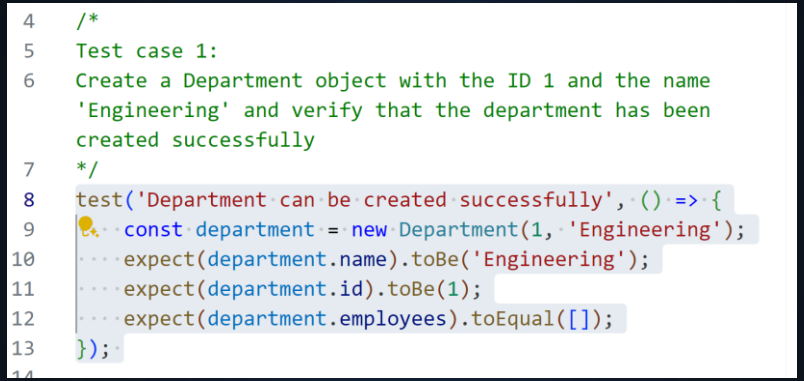
Detect and Remediate Vulnerabilities

Q & A

GitHub Copilot Fundamentals Recap

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

- GitHub Copilot is there to enhance daily work
  - 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!
- Available as an extension to IDEs and editors



```
4   /*
5   Test case 1:
6   Create a Department object with the ID 1 and the name
    'Engineering' and verify that the department has been
    created successfully
7   */
8   test('Department can be created successfully', () => {
9       const department = new Department(1, 'Engineering');
10      expect(department.name).toBe('Engineering');
11      expect(department.id).toBe(1);
12      expect(department.employees).toEqual([]);
13  });
```

Copilot vs Copilot Chat

Copilot

Direct Code Writing

Seamless IDE Integration

Solo Development

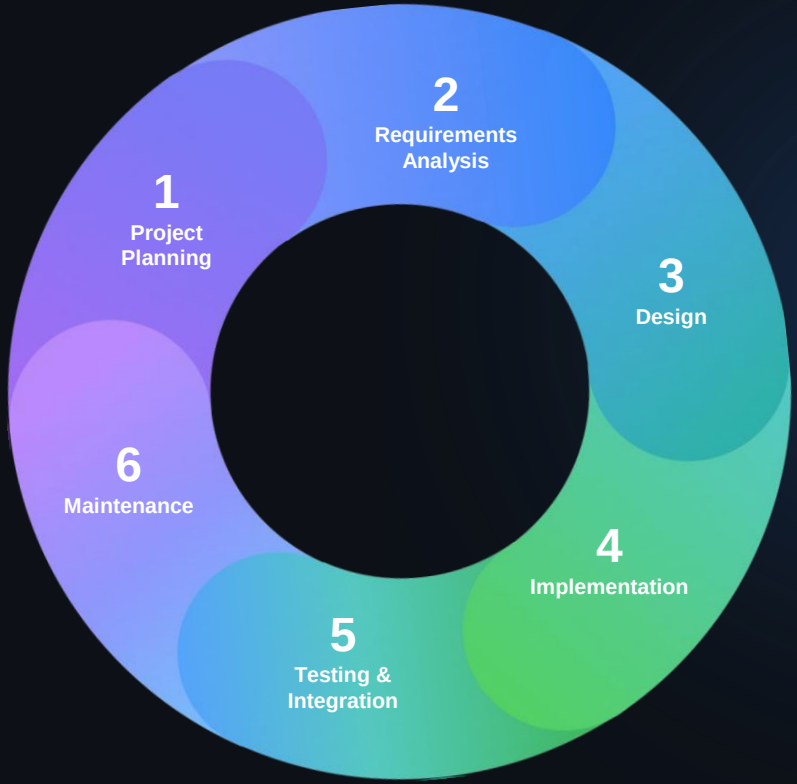
Copilot Chat

In-Depth Interactive Assistance

Learning & Teaching

Collaborative Scenarios

GitHub Copilot + Chat



Helps IT professionals stay in the flow throughout the entire SDLC

Data flow through the Copilot ecosystem

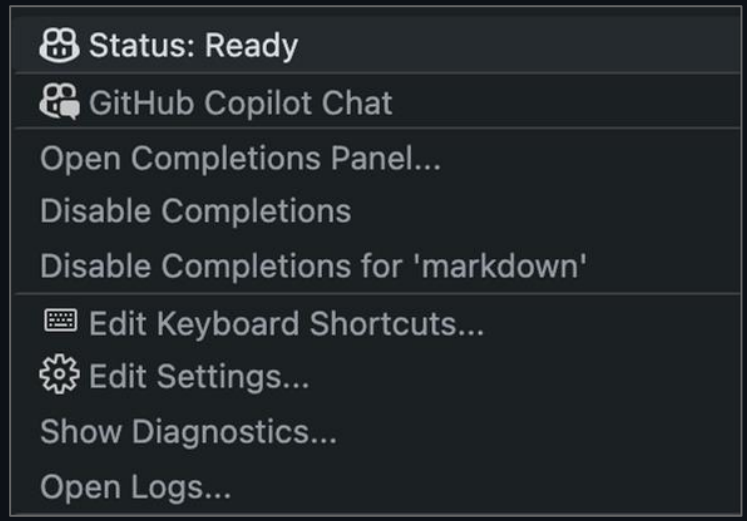
PII
Toxicity
Code classifier

CODE EDITOR | PROXY | MODEL

PII
Toxicity
Code classifier
Code quality
Duplicate detection

VS Code Settings

Configuration Options



Configuring GitHub Copilot in your environment

Prompt Engineering

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.

- Copilot

Best Practices

> **To be understood by the machine, you have to behave like an engineer**

- Write Clear Instructions
- Split Complex Tasks into Simpler Subtasks
- Use External Tools
- Provide References
- Give the Model Time to Think
- Test Changes

Source: OpenAI's Prompt Engineering Strategies

Prompt Quality
For better prediction

Vague Prompt

Q: “How do I implement this API?”

A: “You can implement an API by writing the necessary code and setting up your server to handle requests”

- General
- Non-actionable
- Too broad
- Not very helpful

General Prompt, Little Context

Q: “How do I implement an API in Java?”

A: “To implement an API in Java, you need to use a framework like Spring Boot. You will define your endpoints, handle HTTP requests, and set up your server to process these requests.”

- Some context (framework)
- Narrows down response
- Still lacks depth

Specific Prompt, Detailed Context

Q: “How do I implement a RESTful API in Java using Spring Boot? Provide steps to set up a basic API with a sample endpoint that returns a list of users

A: “1) Set up your project …
2) Create your Main application class:

```
package com.example.demo;
import org.springframework.boot.SpringApplication
…
```

- Clear, detailed & comprehensive
- Instructions and code samples
- More accurate

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.)
- 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

Using Copilot Chat

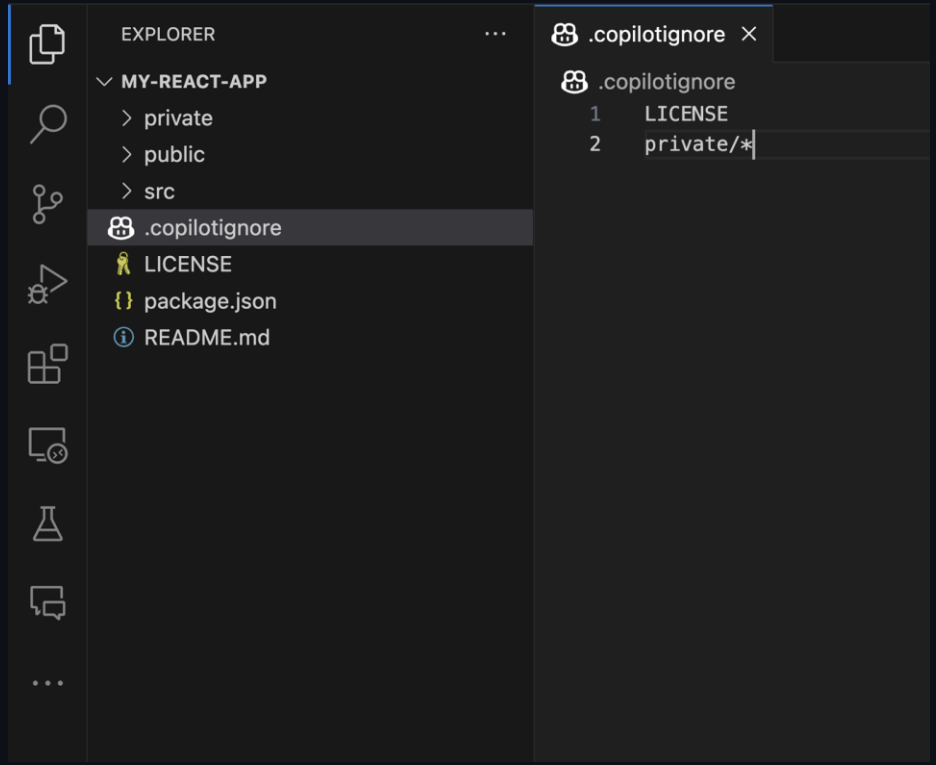
GitHub Copilot Chat

- Refactor Code
- Generate Tests
- Debug
- Create Workspace
- Documentation

Detect and Remediate Vulnerabilities

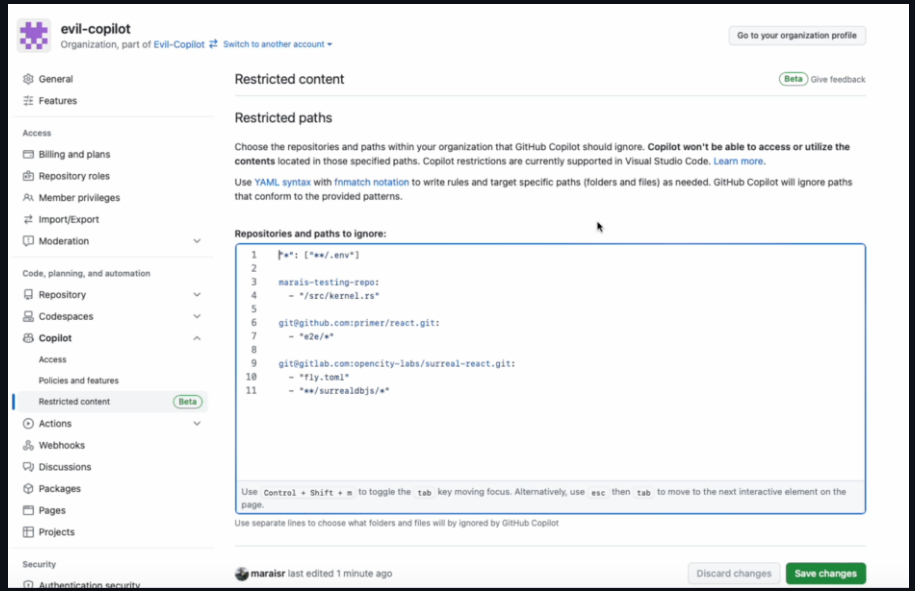
Block files from Copilot

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



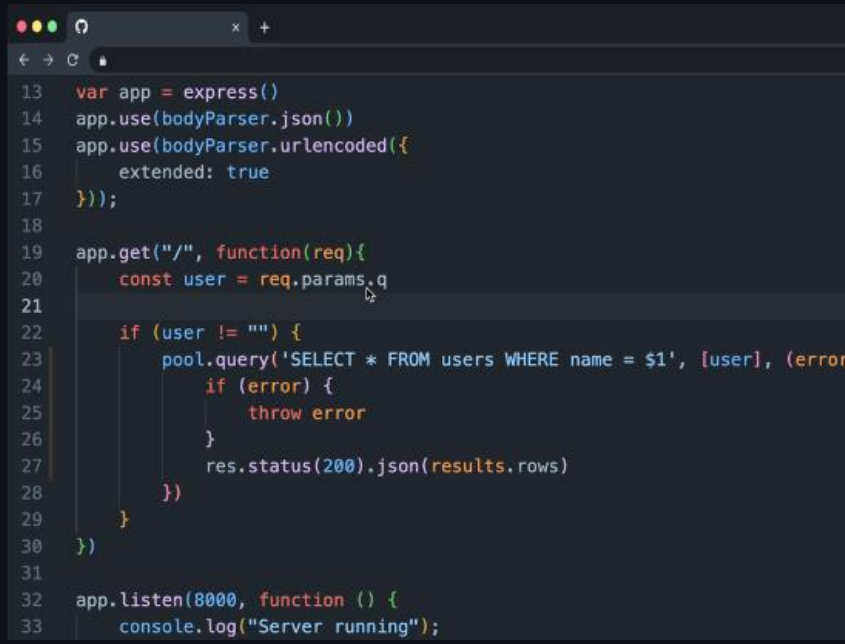
Block files from Copilot

Ignore specific repositories and paths in Restricted Content



Copilot and Secure Coding

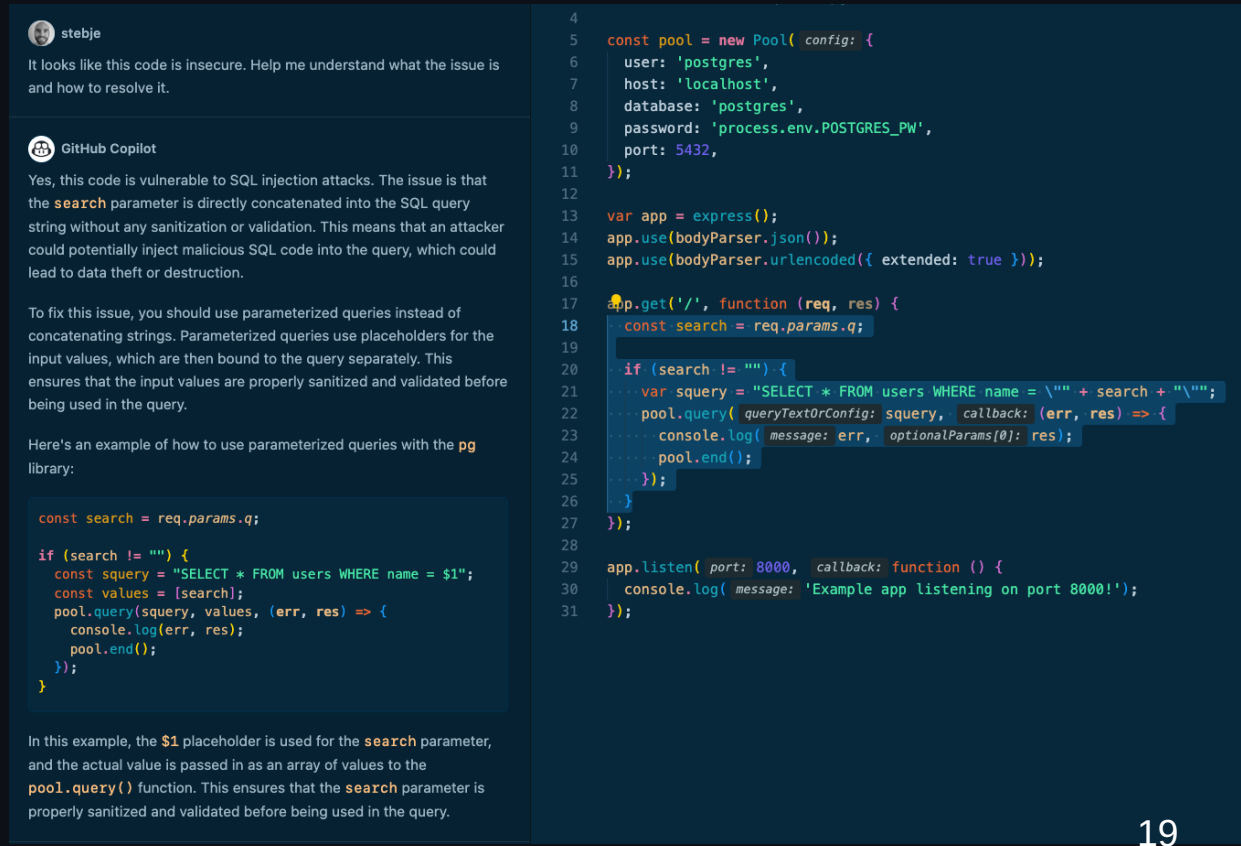
- AI-based vulnerability system that helps prevent insecure coding patterns (e.g. SQL script injection)
- Vulnerability filters are applied only to the Copilot-generated suggestion itself
  - ⊗ It cannot detect downstream vulnerabilities introduced by the code e.g. on deployment infrastructure
  - ⓘ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

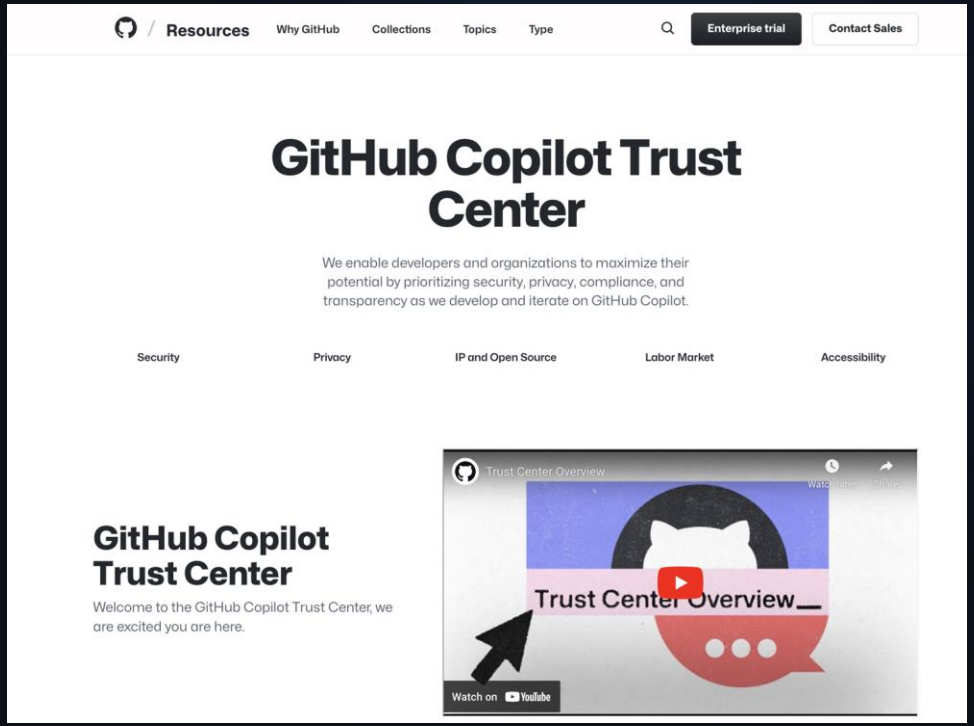
Copilot + GHAS

- 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

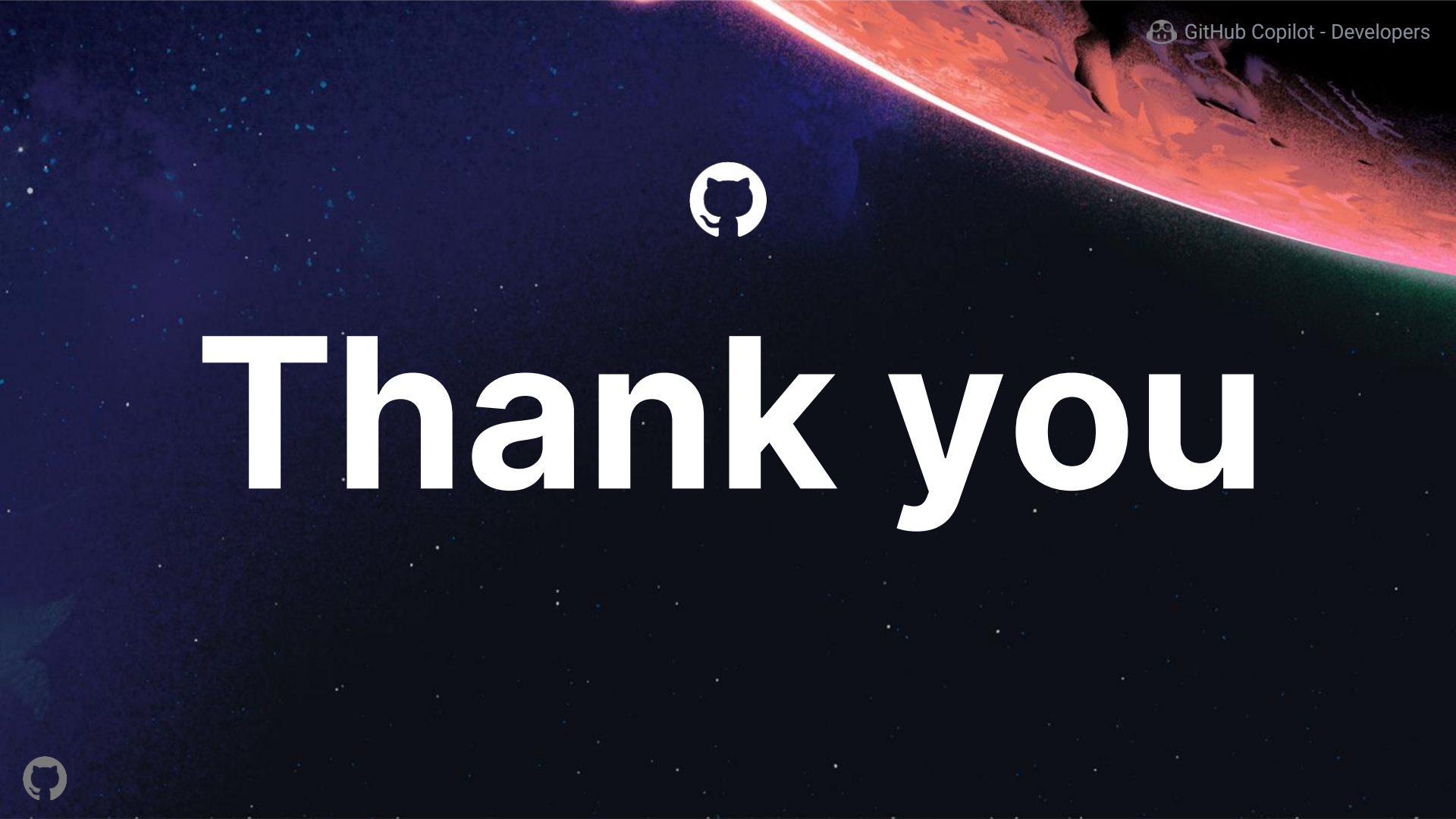
Security & Trust

Copilot Trust Center

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



GitHub Copilot Trust Center - GitHub Resources


Thank you