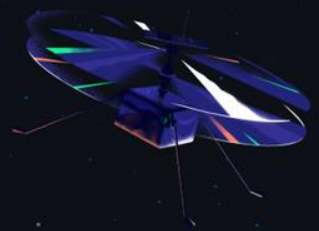
GitHub Copilot Developer Training

Level: Intermediate



Let’s build from here

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

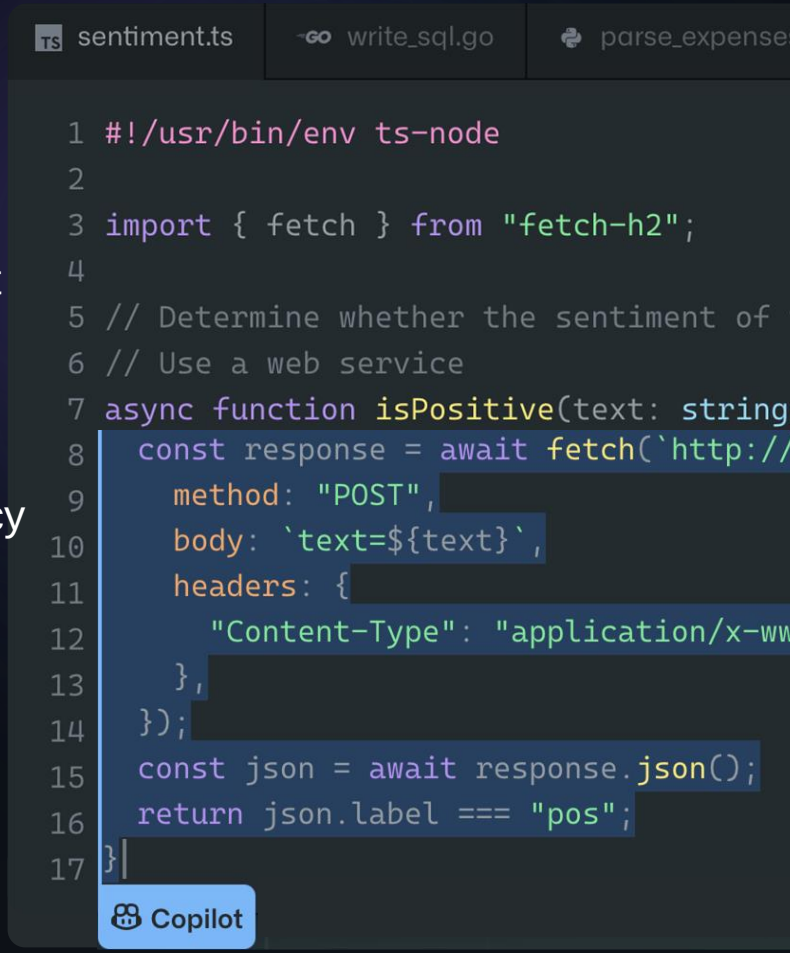
Covered in Copilot Fundamental

We will not talk about...

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

GitHub Copilot

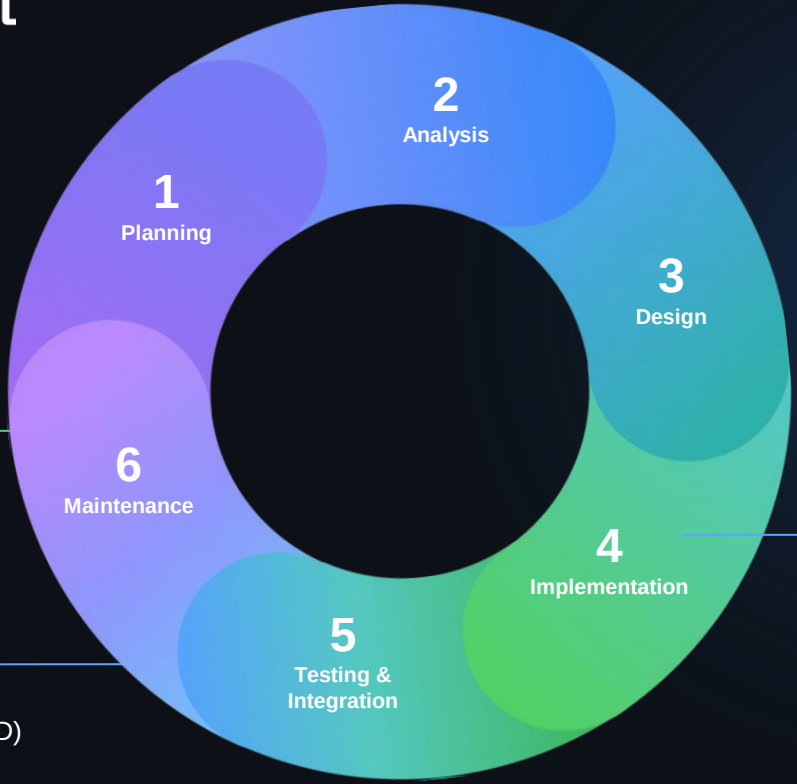
- An intelligent pair programmer
- Draws context from comments & code to suggest individual lines and whole functions
- Powered by OpenAI Codex
  - Copilot uses a transformative model
  - Trained on large datasets to ensure accuracy
- Available as extensions to popular IDEs
- Programming Languages and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



Design

IMPLEMENTATION

GitHub Copilot



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

Once enabled…

OpenAI Codex

runtime.go | course.rb | time.js | IsPrimeTest.java

```go
package main

type Run struct {
    Time int // in milliseconds
    Results string
    Failed bool
}
```

Visual Studio | Neovim | VS Code | JetBrains IDEs

February 2023

GitHub Copilot works something like this

1) Code starts in the user’s editor. The Copilot extension gathers context from open files. That context is called the "prompt."

2) The prompt is sent with SSL encryption to a proxy server living in a GitHub-owned Azure subscription.

3) At the proxy server, we apply filters to scrub PII, tokens, vulnerabilities etc.

4) The proxy server sends the scrubbed prompt to the model.

5) The model produces ‘n’ suggestions based on the prompt, then deletes the prompt from memory.

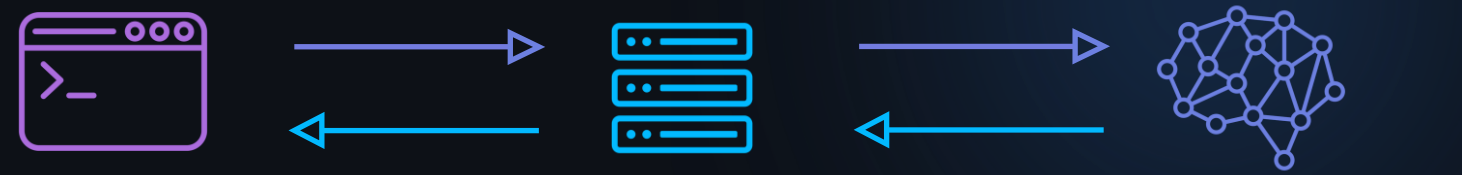
6) The suggestions produced by the model are retained for a period of up to 30 days

7) The responses are sent back to the proxy server for another scrubbing.

8) The proxy server sends the suggestions back to the editor via HTTPS where they are displayed.

Data flow through the Copilot ecosystem

PII
Toxicity
Code classifier



CODE EDITOR | PROXY | MODEL

PII
Toxicity
Code classifier
Code quality
Duplicate detection

Coding

GH Copilot vs GH Copilot Chat

Slash commands

Data Generation & Testing

Code Refining & Refactoring

Block Files & Debugging

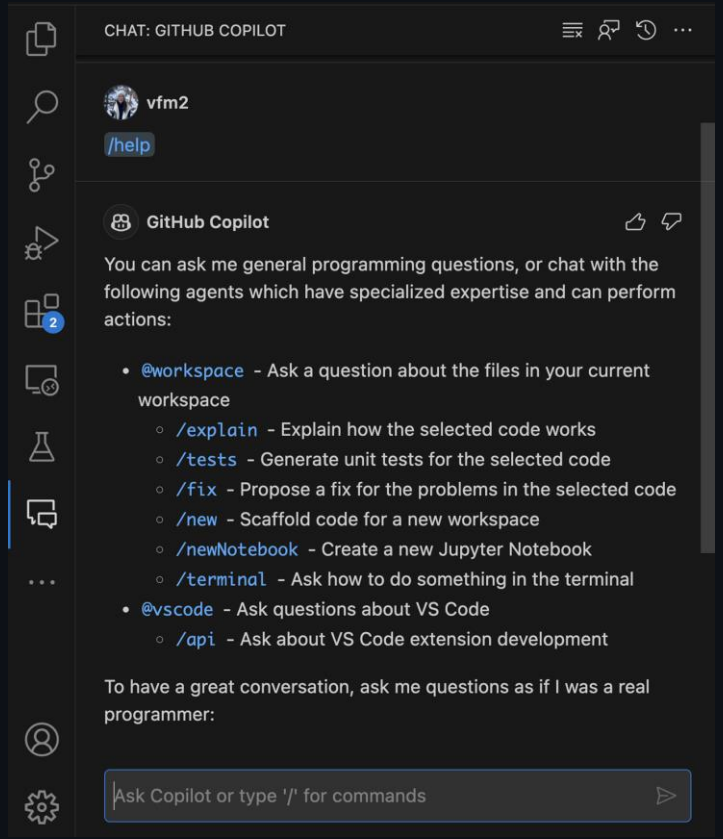
GH Rest APIs

When to use Copilot vs Copilot Chat

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

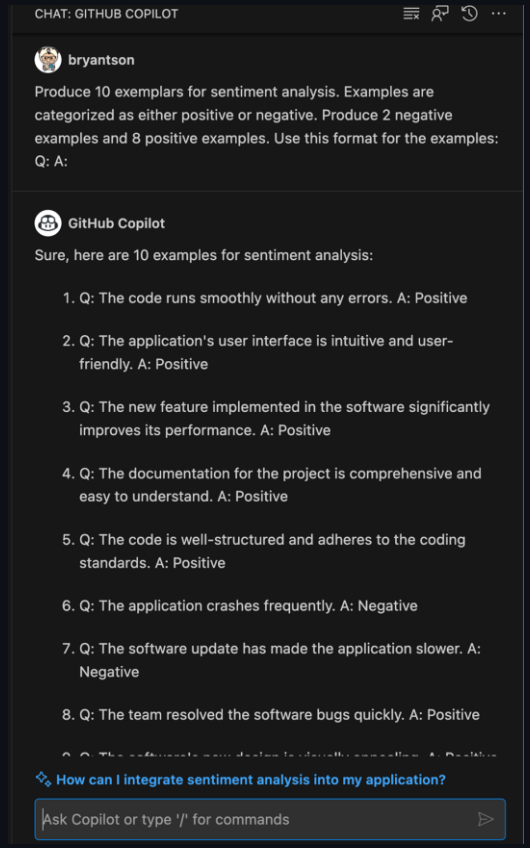
Slash Commands

/help to find available commands in your IDE



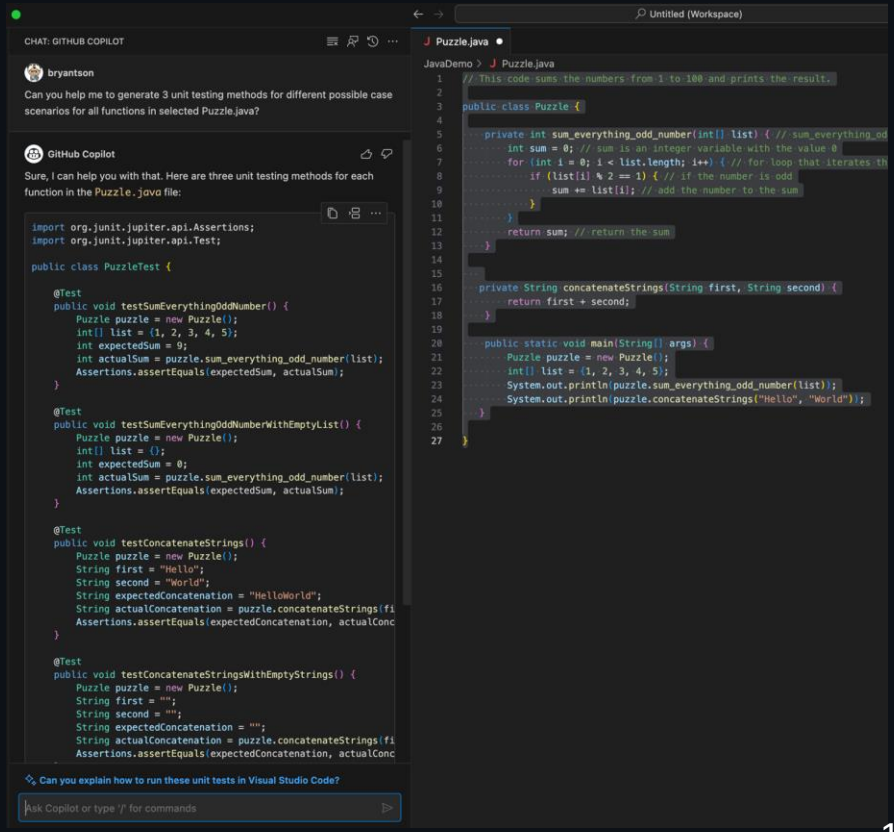
Data Generation

Ask Copilot to
generate data



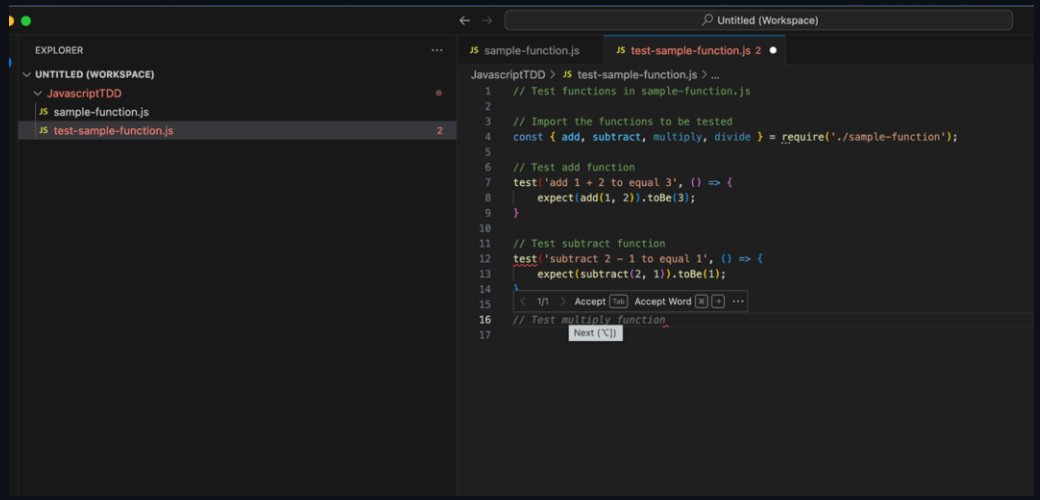
Unit Testing Generation

Works best for general unit testing case generation



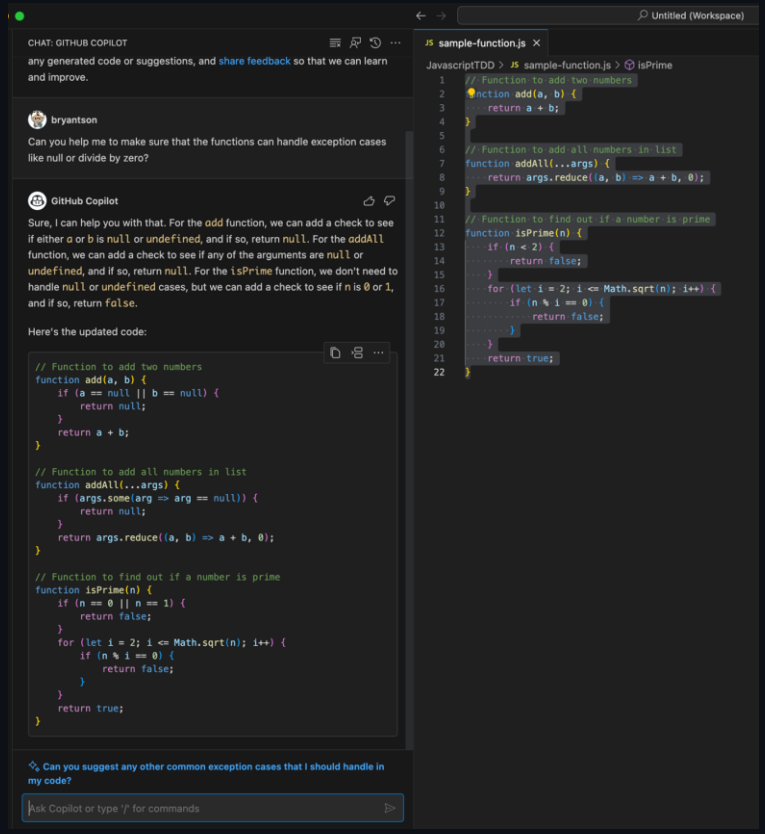
Test Driven Development (TDD)

Test Driven Development (TDD) practice is possible



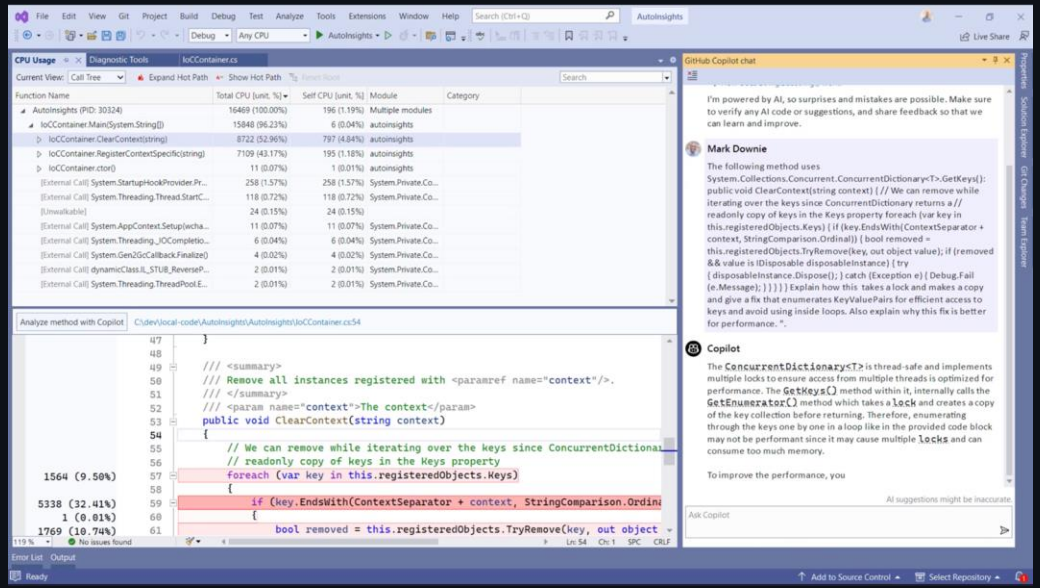
Code Refactoring

Refactoring is possible through GitHub Copilot Chat



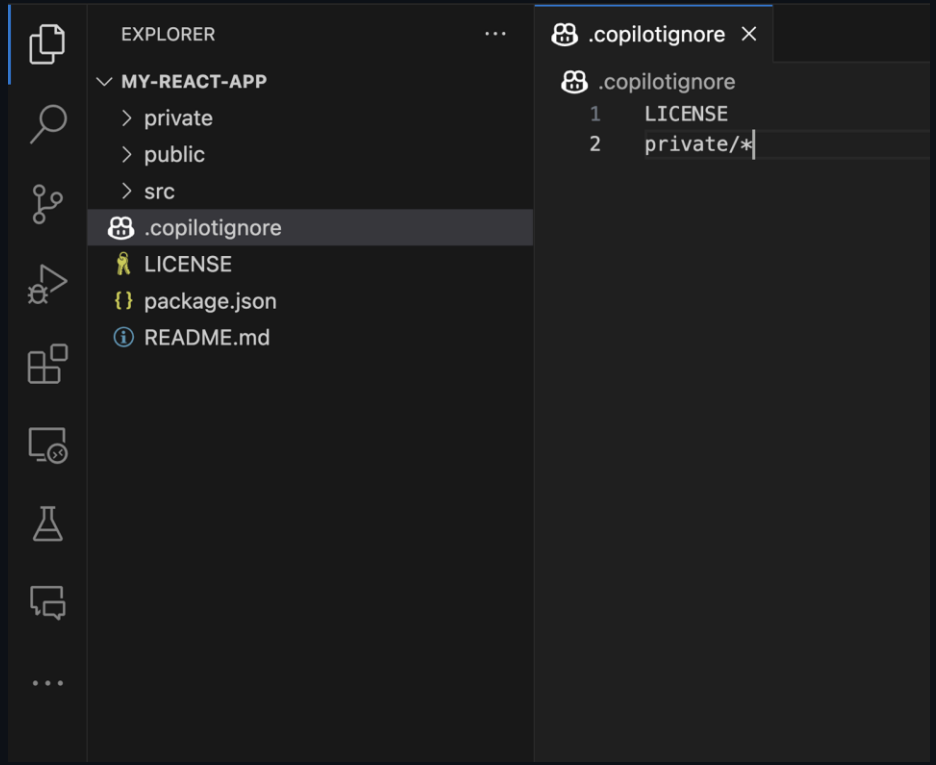
Code Refining

GH Copilot CPU usage tool



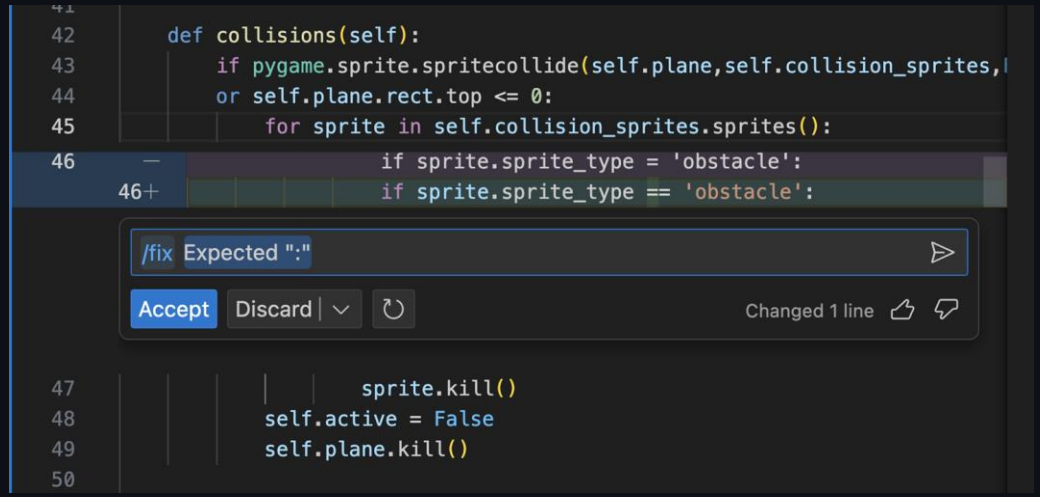
Block files from Copilot

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



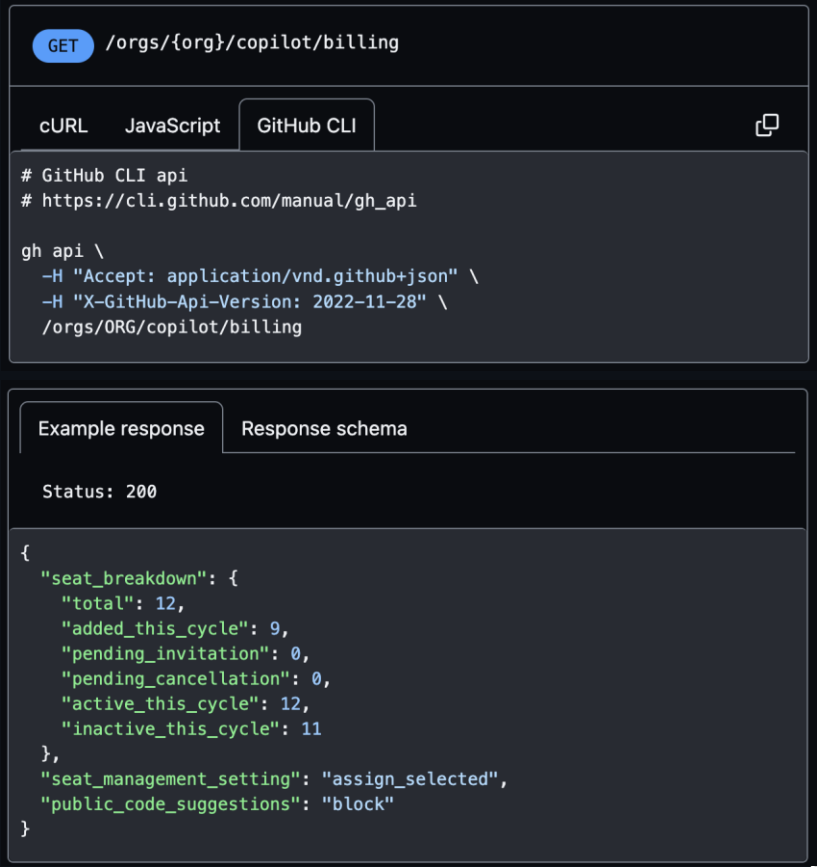
Code Debugging

Debugging with GitHub Copilot



GitHub Rest API

GitHub Copilot Rest API



Demo: Coding practices

Best practices

Getting accuracy closer to the expectation

- Prompt Engineering

Working at a project(s) level

Leveraging Copilot to increase code quality

Addressing concerns around security exploit

Getting accuracy closer to expectation

Problems

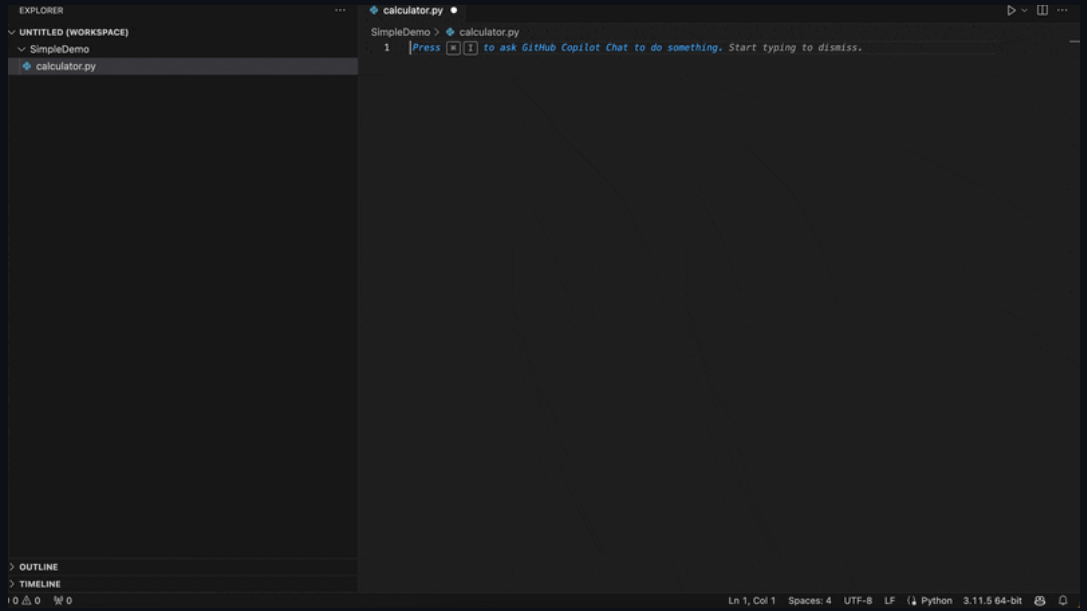
- Copilot fails to produce answer or to keep repeating
- Copilot generates incorrect result
- Library/module version discrepancies issue
- Copilot suggests non-optimal solution

Problems #1:

Copilot fails to produce answers or will keep repeating

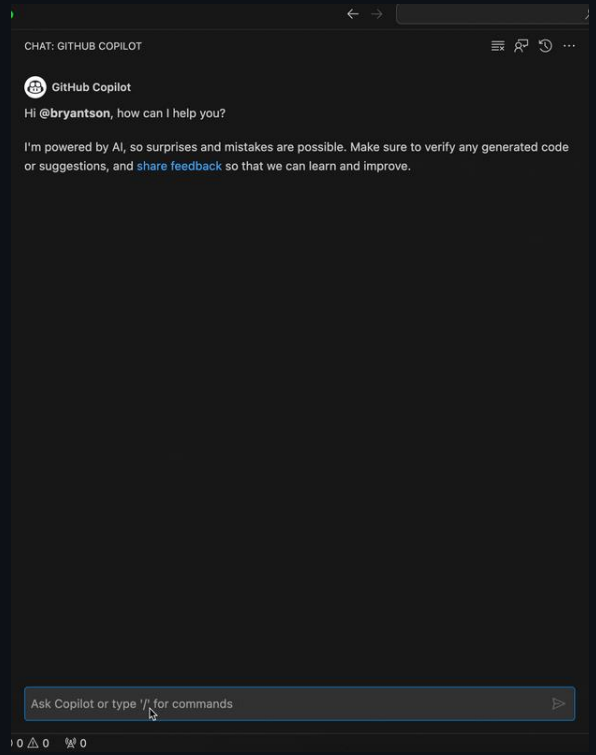
Some problems

- Fails to produce answer
- Hallucination - Keeps repeating



Problems #2:

Copilot generates incorrect result



Reasons why...

- Not enough context
- Old trained data
- Generative in nature

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.

- Copilot

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

Why is Prompt Engineering Important?

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

Techniques

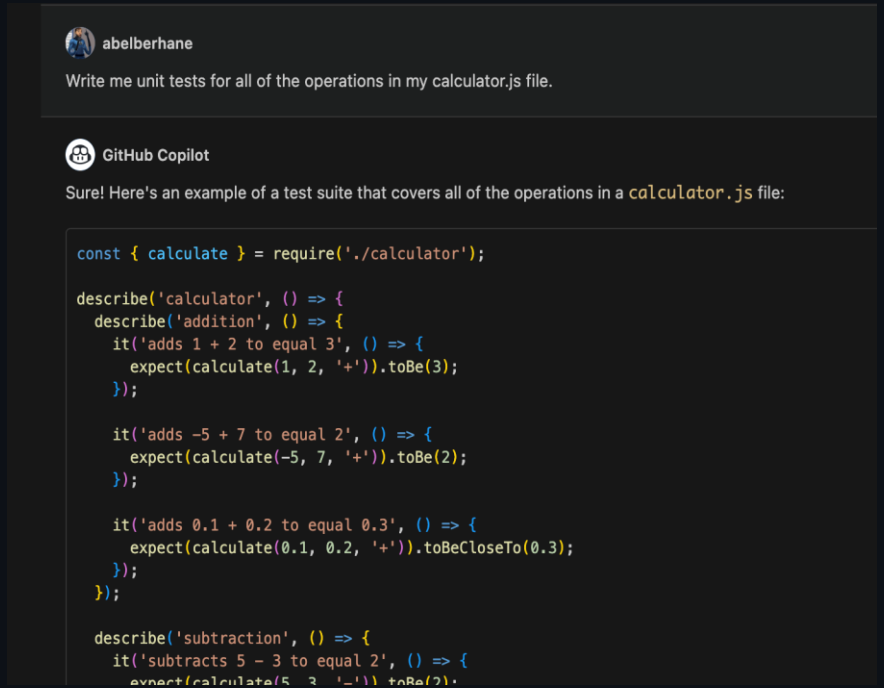
Zero-Shot Prompting

One-Shot Prompting

Few-Shot Prompting

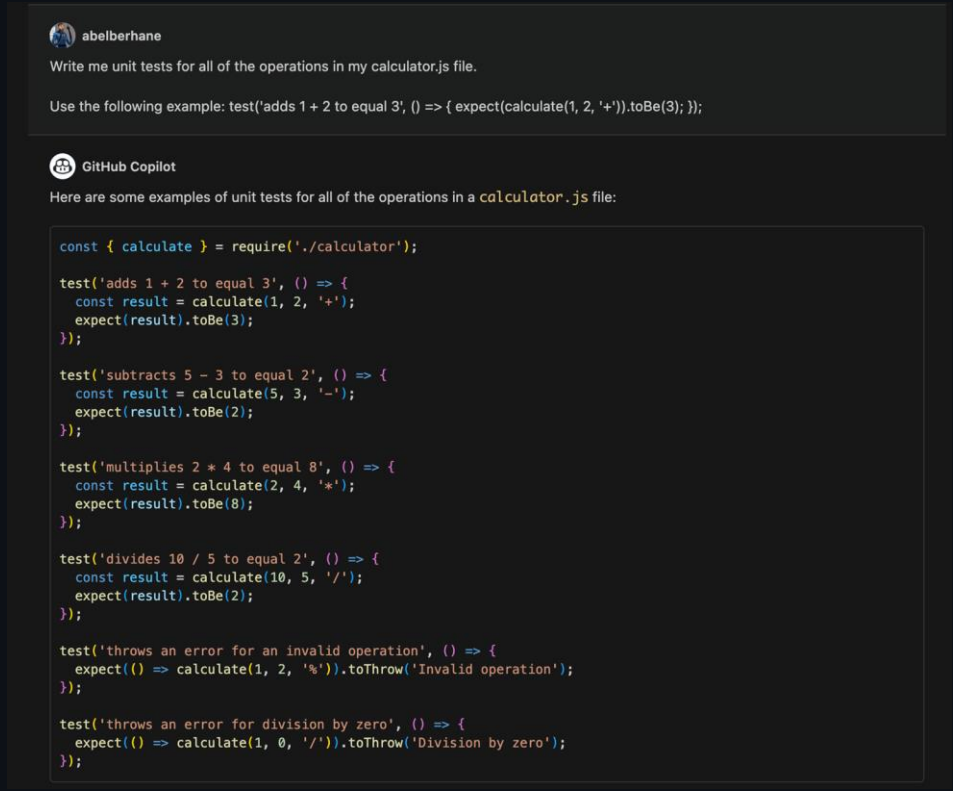
Techniques

Zero-Shot Programming with Copilot



Techniques

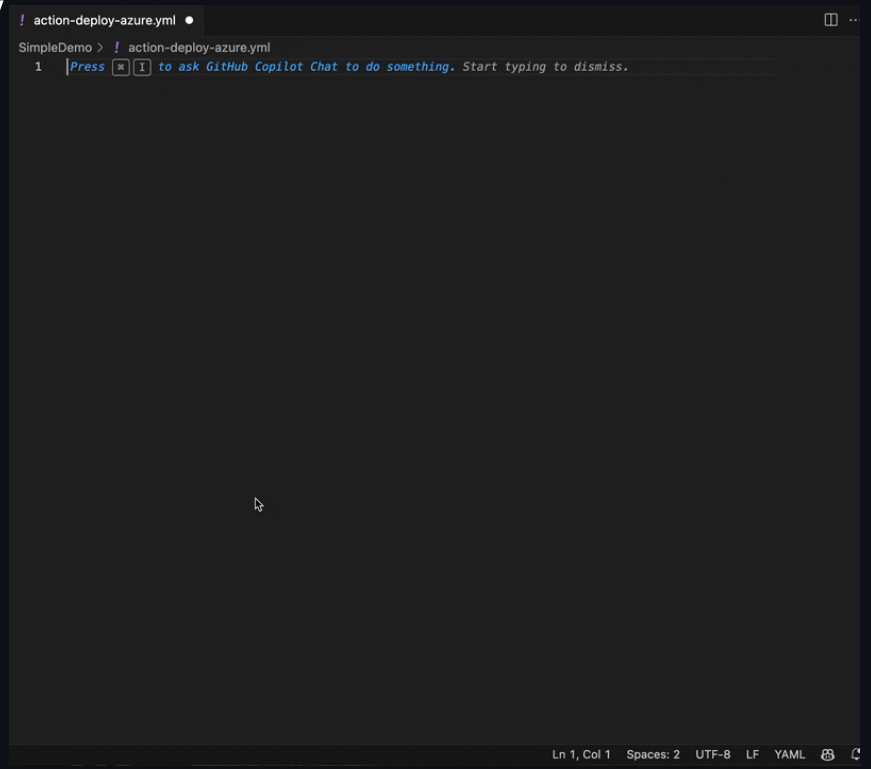
One or few shot Programming with Copilot



Demo: Prompt Engineering

Problems #3:

Library/module version discrepancy

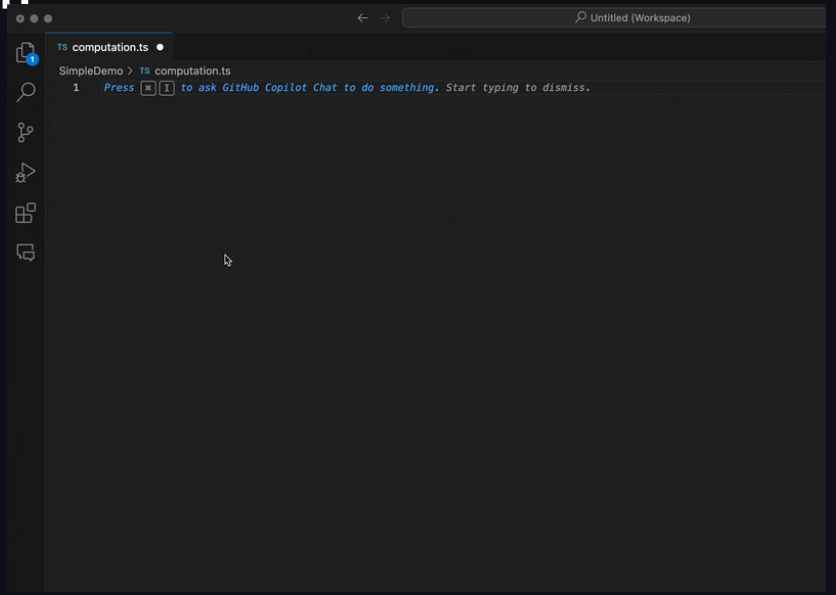


Old trained data

- While packages go through frequent updates, Copilot does not use latest data

Problems #4:

Copilot suggests non-optimal solution



Although solution works...

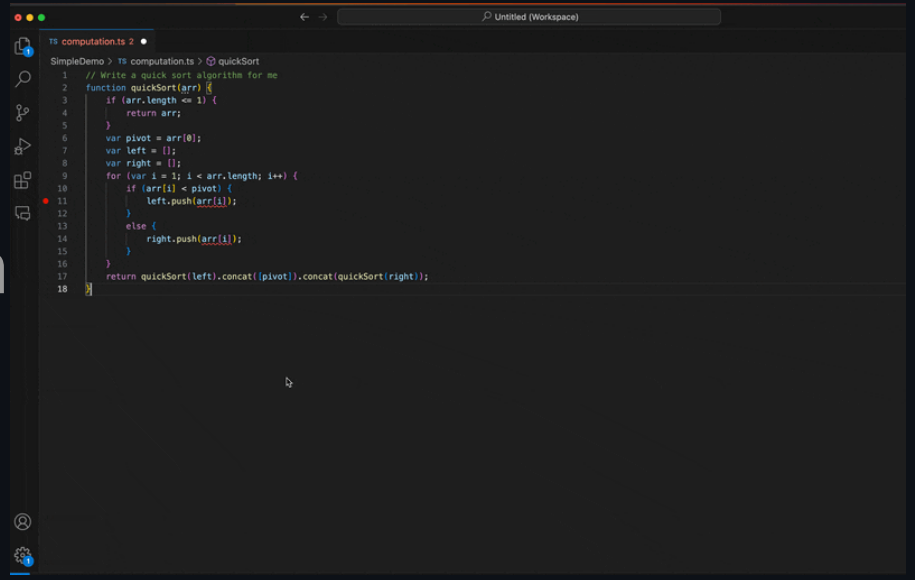
Suggested solution is not optimal because Quick Sort can be implemented in O(1) complexity, meaning no space required

How to improve Problem #4

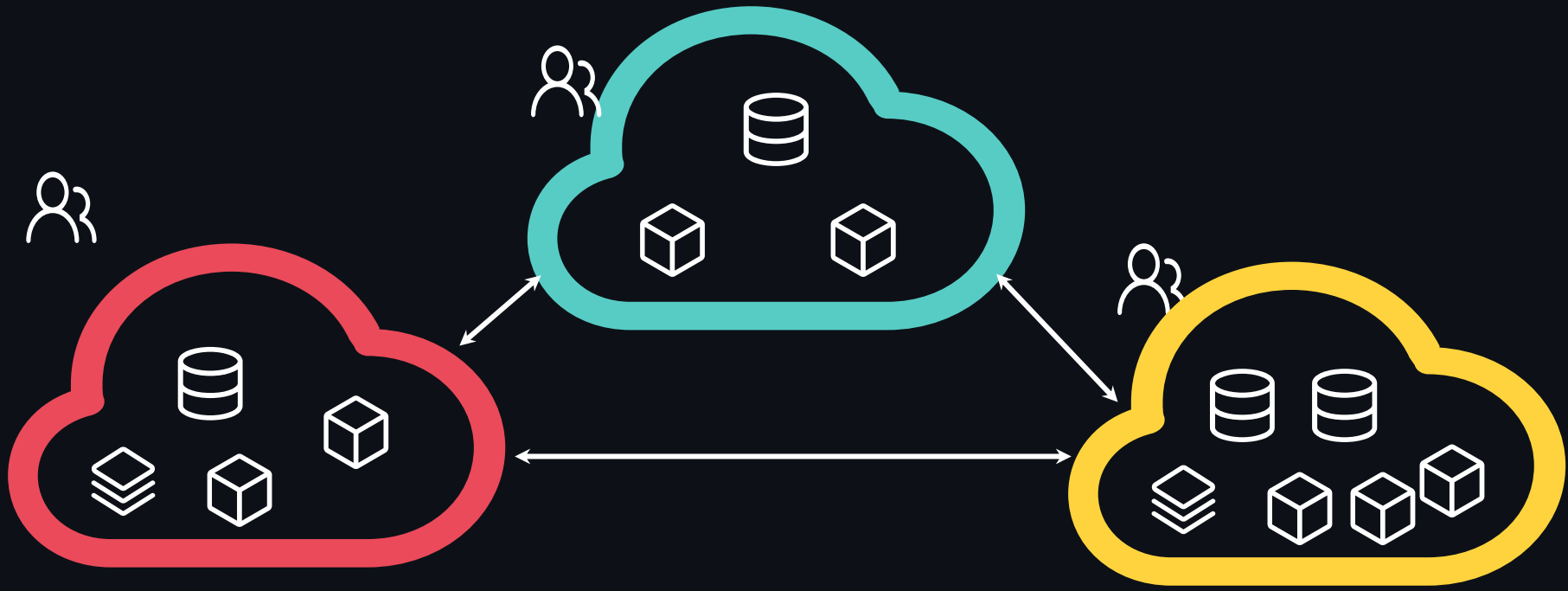
Copilot suggests non-optimal solution

How to improve

- Ask in Copilot Chat
- Use in-line suggestion from Copilot



How to work at project(s) level



Neighboring Tabs

sentiments.ts | write_sql.go | parse_expenses.py | addresses.rb

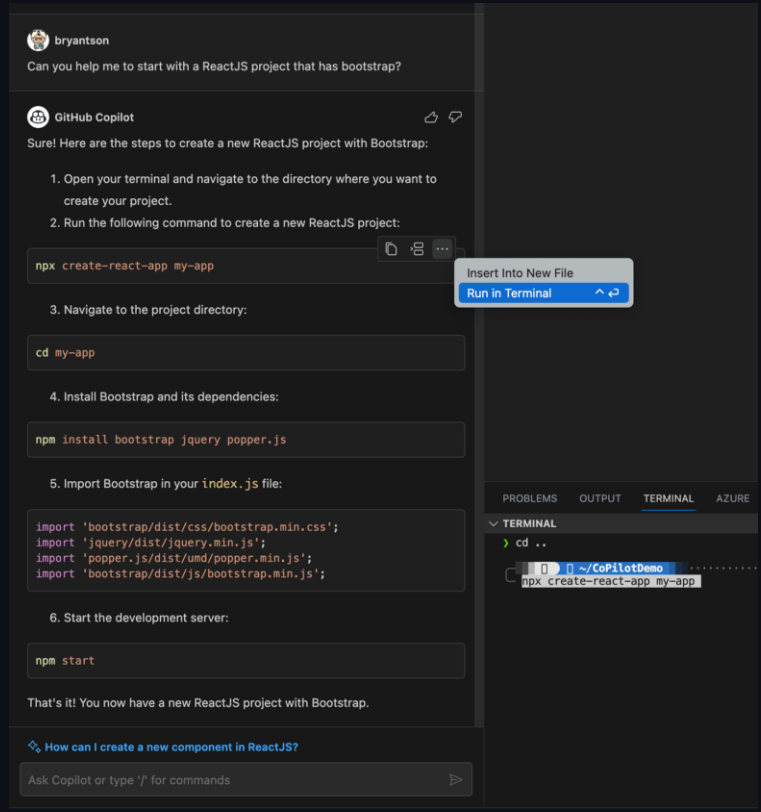
```ts
#!/usr/bin/env ts-node

import { fetch } from "fetch-h2";

// Determine whether the sentiment of text is positive
// Use a web service
async function isPositive(text: string): Promise<boolean> {
  const response = await fetch(`http://text-processing.com/api/sentiment/`, {
    method: "POST",
    body: `text=${text}`,
    headers: {
      "Content-Type": "application/x-www-form-urlencoded",
    },
  });
  const json = await response.json();
  return json.label === "pos";
}
```

Working in a project level

Ask in GitHub Copilot Chat to get started

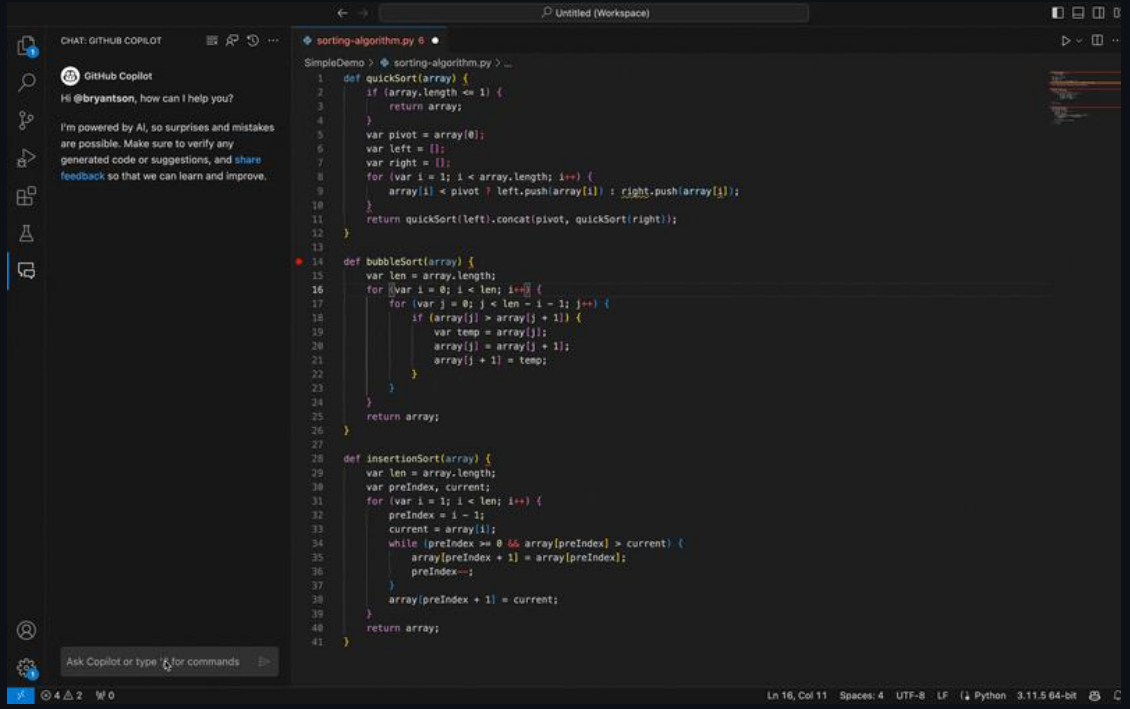


GitHub Copilot Chat (default)

Using Copilot to improve code quality

Copilot Chat: Ask to Refactor

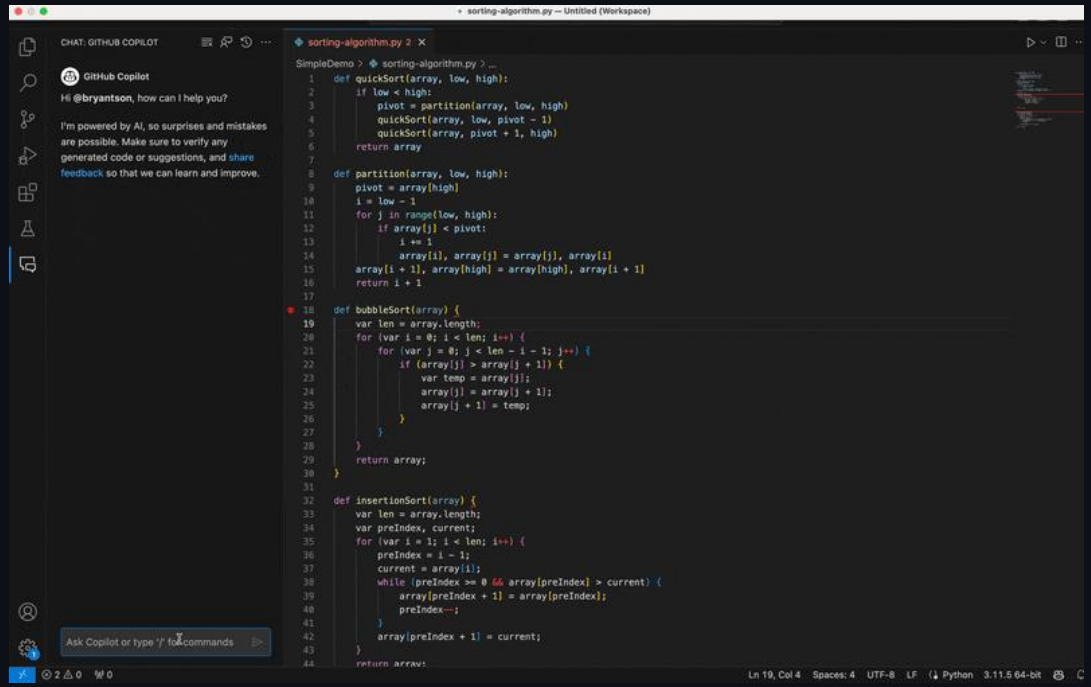
Ask GitHub Copilot Chat to refactor your code



GitHub Copilot Chat (default)

Copilot Chat: Ask to generate tests

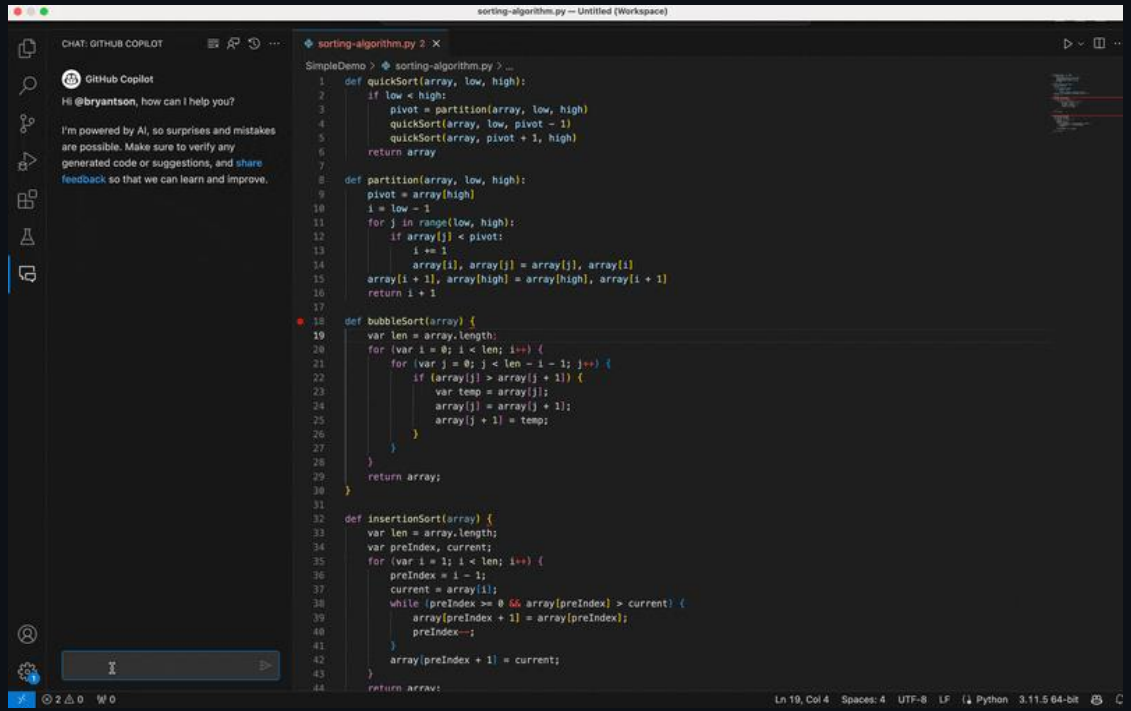
Ask GitHub Copilot Chat to generate tests



GitHub Copilot Chat (default)

/fix, /tests and other / command options

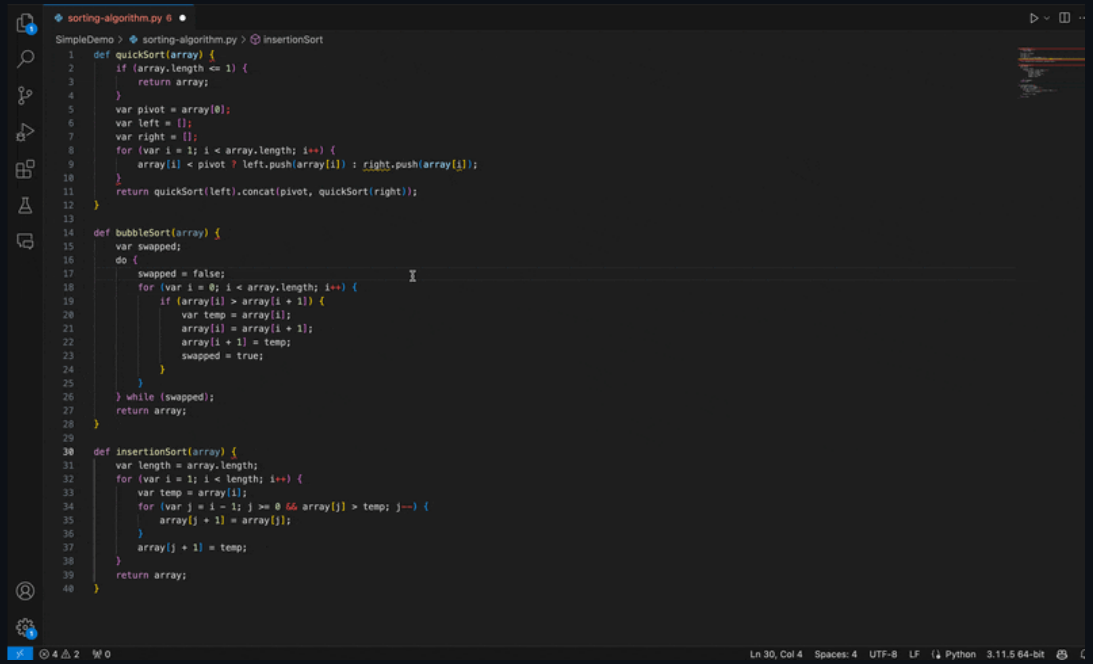
Leverage / command options that can help to improve your code



GitHub Copilot Chat (default)

In-file Copilot options

Copilot now offers in-file Copilot feature to selectively improve

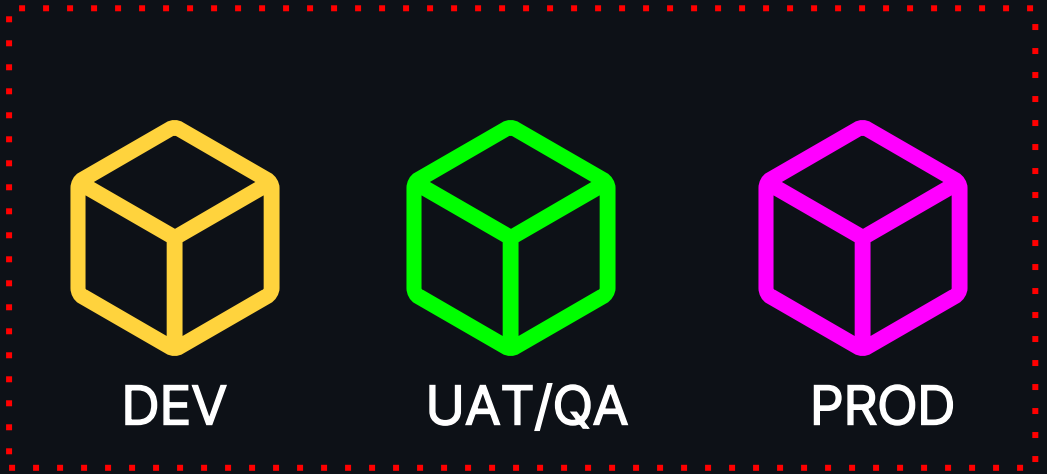


GitHub Copilot Chat (default)

What about my production layer?

Local

Cloud/Server



Current Copilot

“Copilot X”

Copilot Security Exploit?

Prompt Injection

Carefully crafted prompts to make the model to ignore its original instruction or perform unintended actions

Workshop

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

Secure coding

Copilot and secure coding

Copilot + GitHub Advanced Security

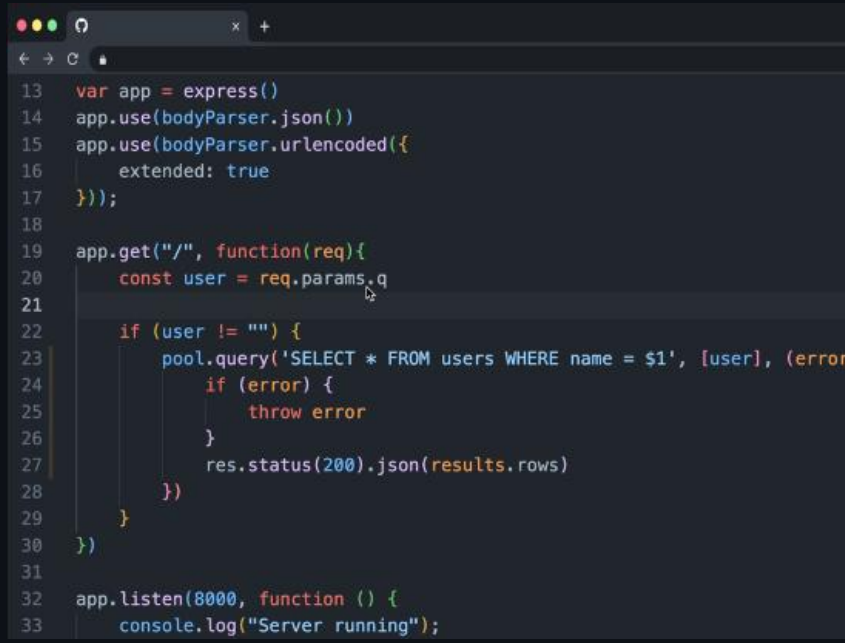
Vulnerability detection

Vulnerability remediation

Copilot Trust Center

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



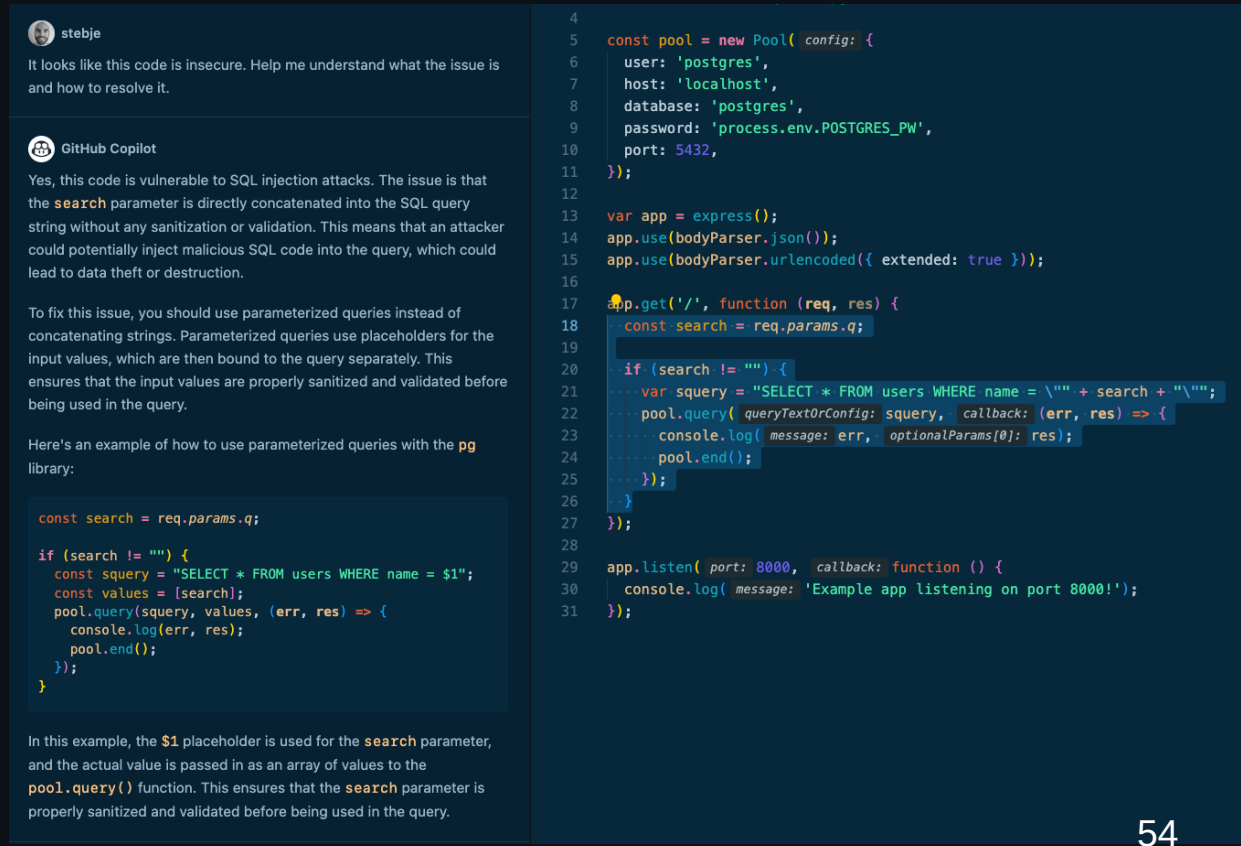
```
13  var app = express()
14  app.use(bodyParser.json())
15  app.use(bodyParser.urlencoded({
16      extended: true
17  }));
18
19  app.get("/", function(req){
20      const user = req.params.q
21
22      if (user != "") {
23          pool.query('SELECT * FROM users WHERE name = $1', [user], (error
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");
```

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

Vulnerability detection

- 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 });
```

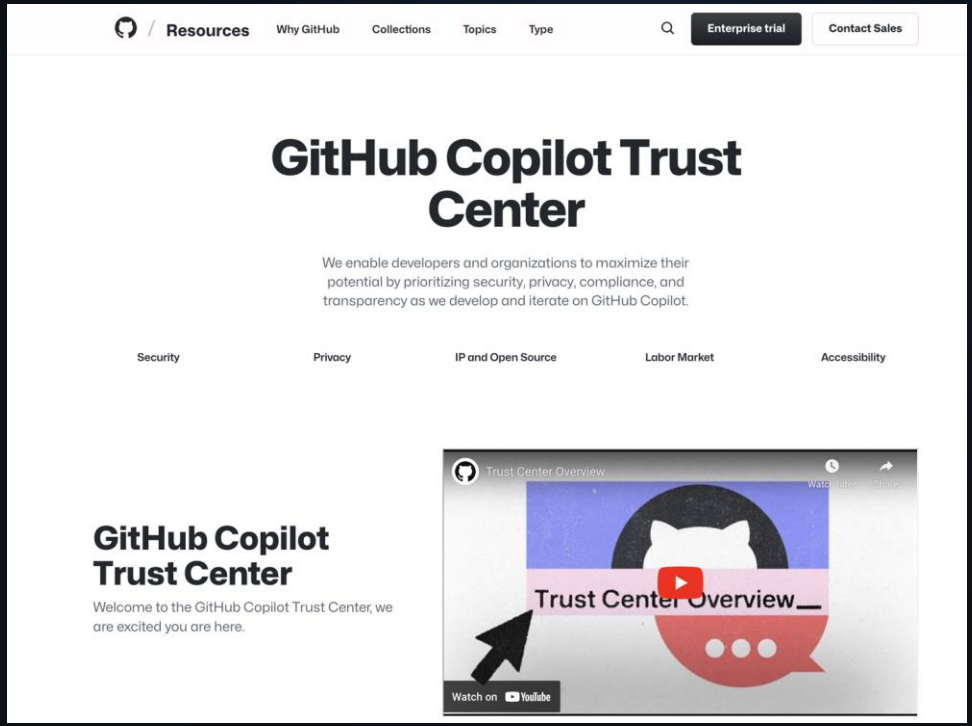
Vulnerability remediation

- Copilot helps prevent suggestions that contain insecure coding patterns in real-time
- Automatically fix insecure code based on Copilot suggestions
- Validate and improve existing CodeQL queries

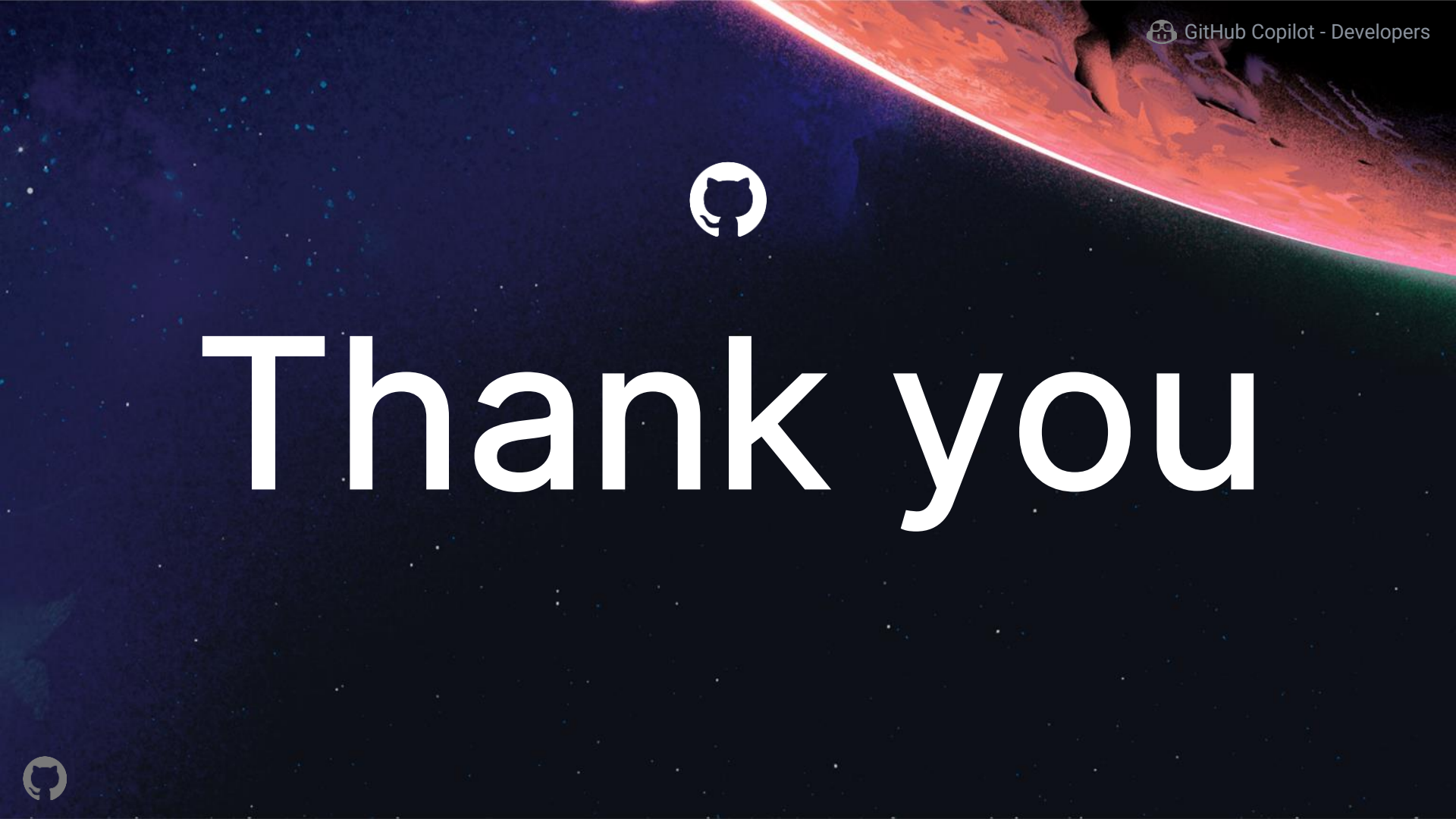
Security & Trust

Copilot Trust Center

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



Wrap Up


Thank you