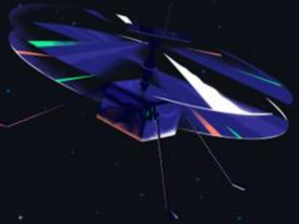
GitHub Copilot Training for Developers

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

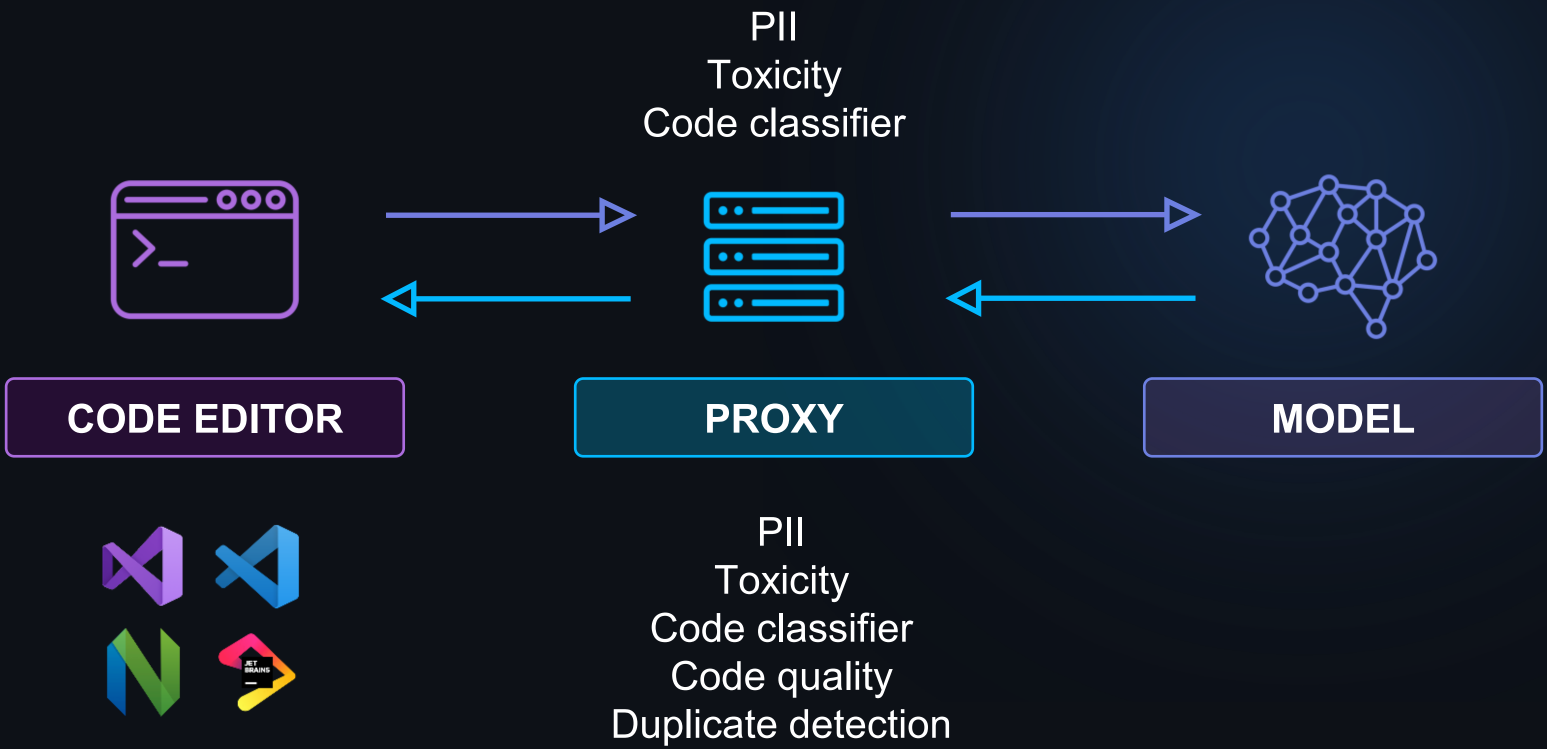
- GitHub Copilot is there to enhance daily work
  - It’s an intelligent “pair-programmer”
  - Like a smart assistant or mentor by your side

- Powered by OpenAI
  - Copilot uses a transformative model
  - Think of something like Google Translate

- Trained on large datasets to ensure accuracy
  - Languages with large representation produce best results

- Copilot generates new code in a probabilistic way
  - Unlikely to produce the same code as a snippet that occurred in training.

Data flow through the Copilot ecosystem



PII
Toxicity
Code classifier

CODE EDITOR

PROXY

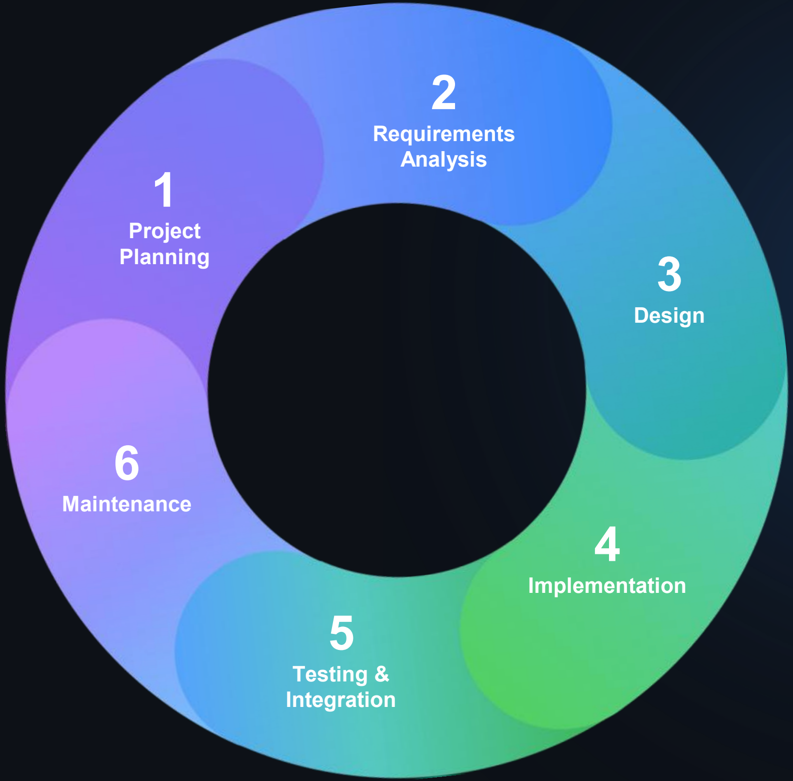
MODEL

PII
Toxicity
Code classifier
Code quality
Duplicate detection

Copilot vs Copilot Chat

Copilot	Copilot Chat
Direct Code Writing	In-Depth Interactive Assistance
Your “coding assistant”	Your “research assistant”
Solo Development	Collaborative Scenarios

GitHub Copilot + Chat



1
Project Planning

2
Requirements Analysis

3
Design

4
Implementation

5
Testing & Integration

6
Maintenance

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

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]

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

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
- #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 strategic crafting of user inputs to guide the AI towards producing desired code outputs.

This involves techniques like role prompting, contextual detailing, and iterative refinement, which collectively enhance the interaction between the developer and Copilot.

Structured Prompt Engineering

- **Define Goal**
  - One sentence on what you want.

- **Provide Context**
  - Key constraints, environment, and inputs.

- **Include Sources and Links**
  - Only the needed code/spec links or snippets.

- **Specify Expectations**
  - Exact output format and style.

Example:

Role-play as a senior Python reviewer mentoring a programmer.
Goal: Generate deterministic pytest unit tests for the Producer–Consumer in apps/python/ex1.
Context: Python 3.13, stdlib only; use delay=0 in tests; enforce ValueError on negative count or delay; stop on unique SENTINEL; queue empty after run.
Sources: apps/python/ex1/specs.md and the files producer.py, consumer.py, run.py, logging_utils.py, __init__.py in copilot-base.
Expectations: Output exactly one test file (tests/test_producer_consumer.py) in a single fenced code block with the filepath comment; cover count=0 and normal flow; assert sentinel consumed once; keep assertions concise; no prose; ≤80 lines.

Use Prompting Across the SDLC

Observe the technique being used in the following prompt:

> “You are an experienced software engineer.
>
> - 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 Python examples of tasks or activities within each phase where Copilot can be particularly beneficial.”
>
> - The output should be formatted as markdown with clear delineation of phases.

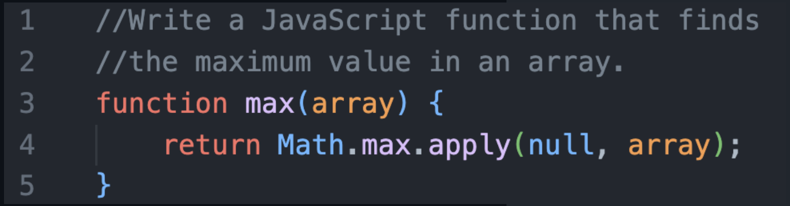
Interactive Coding with Copilot

To help Copilot generate accurate suggestions when writing code:

- 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

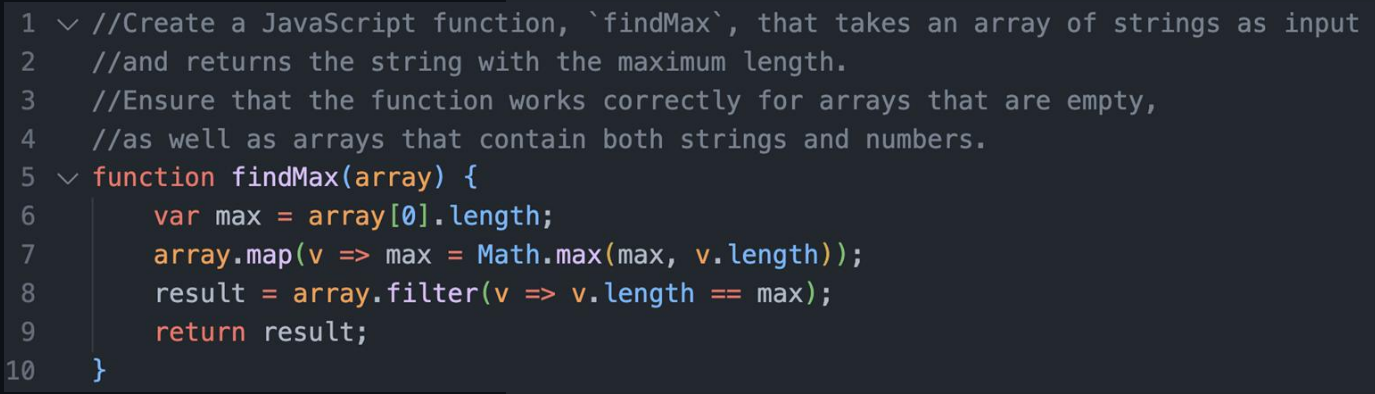
AI Prompt Techniques

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;
}
```

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.
```

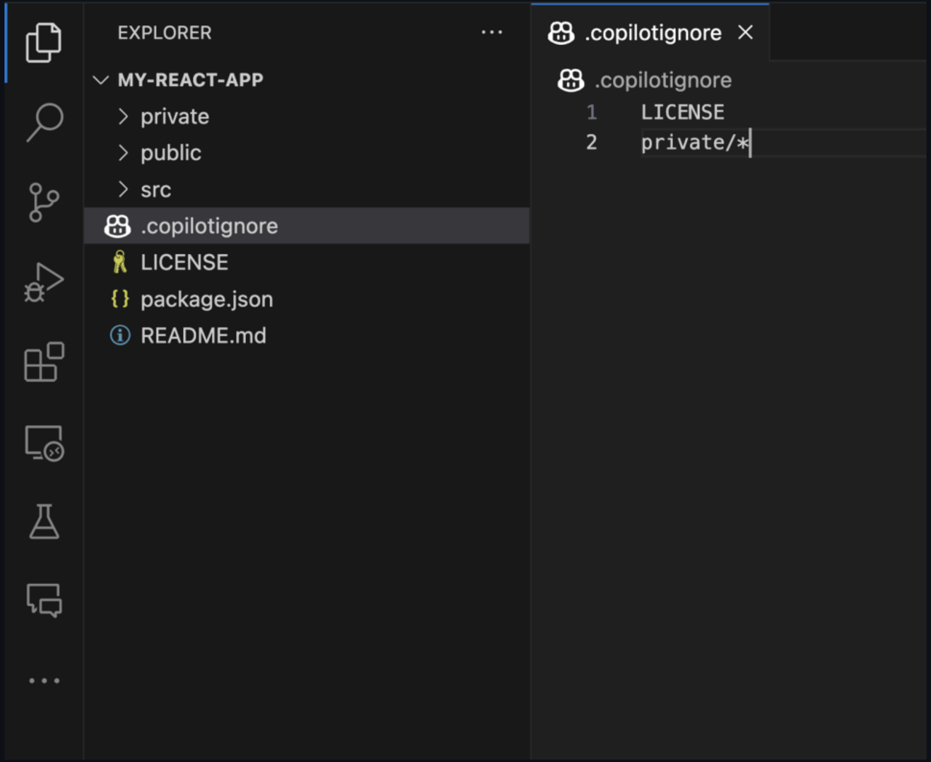
Helpful Patterns

```
* 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

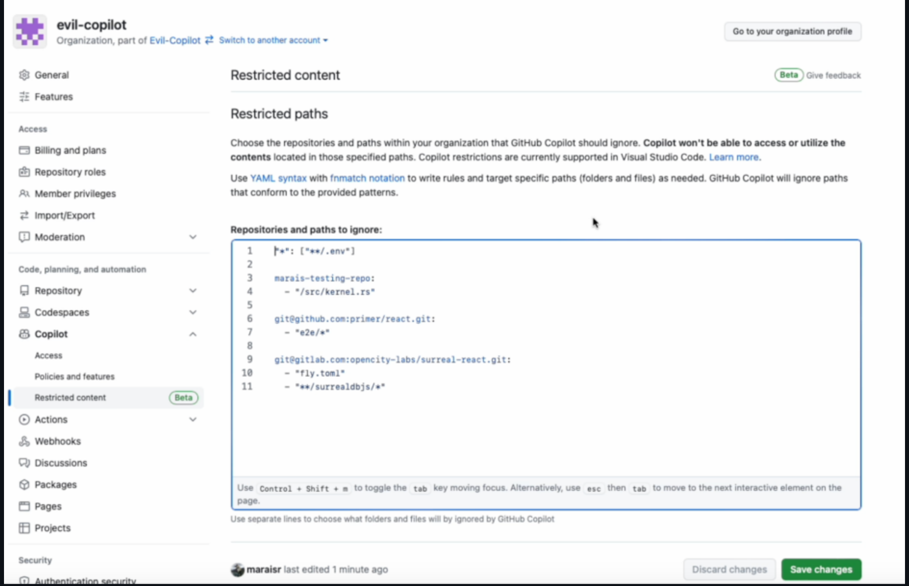
Use .copilotignore to block files and folders from being used by Github Copilot



```
LICENSE
private/*
```

Copilot Restricted Content feature

- 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.



Restricted content

Restricted paths

Choose the repositories and paths within your organization that GitHub Copilot should ignore. **Copilot won't be able to access or utilize the contents** located in those specified paths. Copilot restrictions are currently supported in Visual Studio Code. Learn more.

Use YAML syntax with fnmatch notation to write rules and target specific paths (folders and files) as needed. GitHub Copilot will ignore paths that conform to the provided patterns.

Repositories and paths to ignore:

```
"*": ["**/.env"]

marais-testing-repo:
  - "/src/kernel.rs"

git@github.com:primer/react.git:
  - "e2e/*"

git@gitlab.com:opencity-labs/surreal-react.git:
  - "fly.toml"
  - "**/surrealdbjs/*"
```

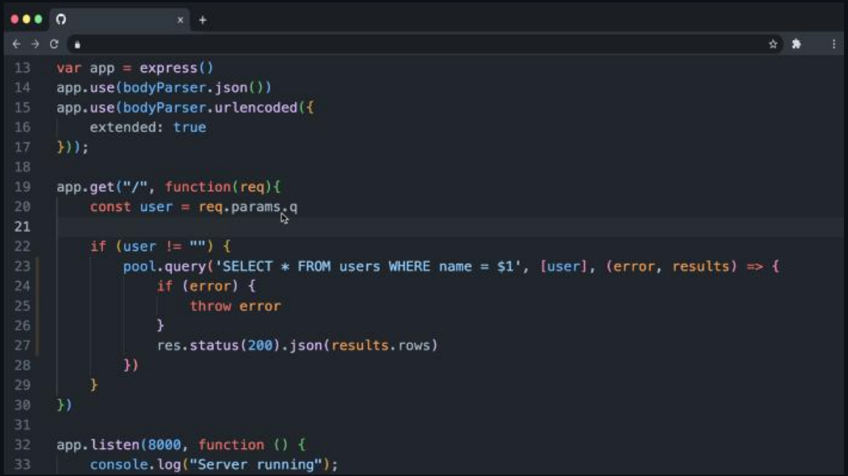
Use `Control + Shift + m` to toggle the `tab` key moving focus. Alternatively, use `esc` then `tab` to move to the next interactive element on the page.

Use separate lines to choose what folders and files will by ignored by GitHub Copilot

maraisr last edited 1 minute ago

Copilot and Secure Coding

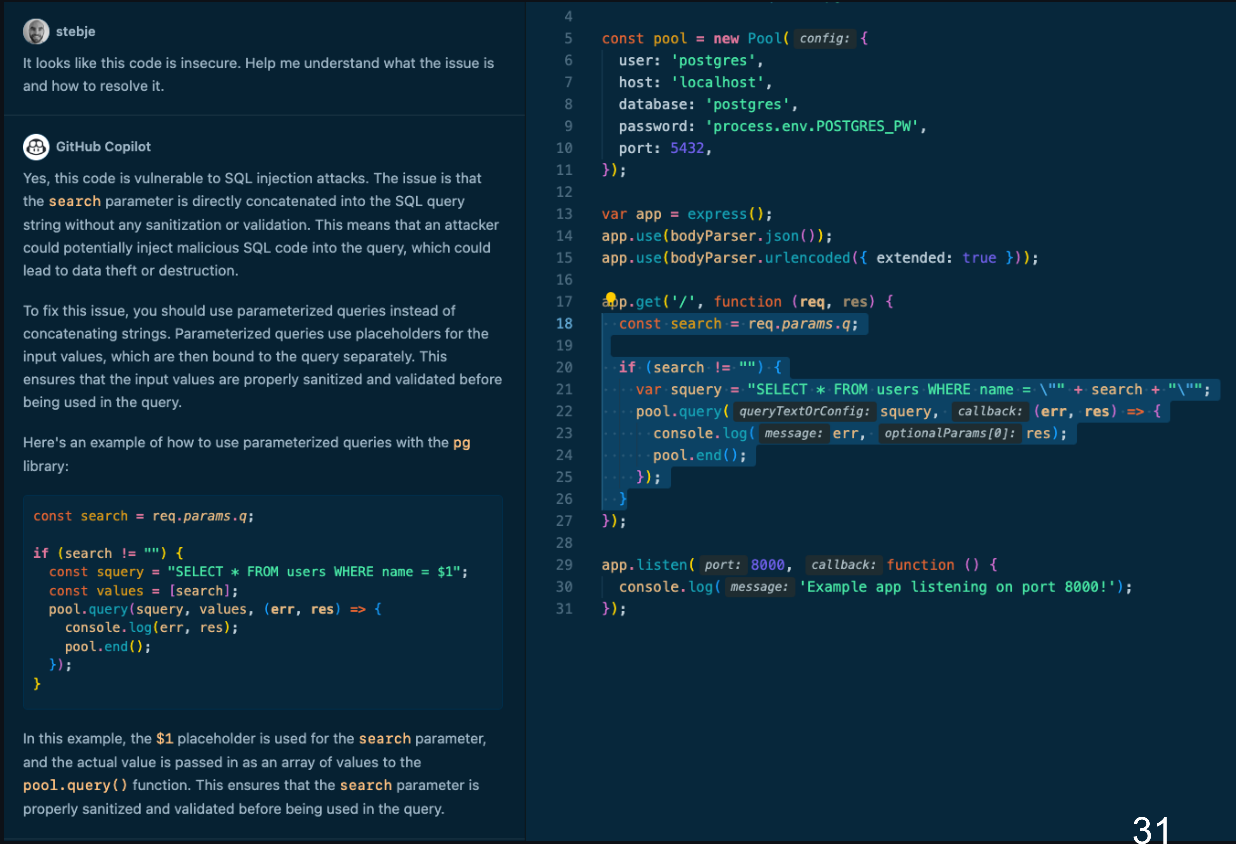
- AI-based vulnerability system that helps prevent insecure coding patterns (e.g. SQL script injection)
- Vulnerability filters are applied only to the Copilot-generated suggestion itself
    - ✕ It cannot detect downstream vulnerabilities introduced by the code e.g. on deployment infrastructure
   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



```
var app = express()
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({
    extended: true
}));

app.get("/", function(req){
    const user = req.params.q

    if (user != "") {
        pool.query('SELECT * FROM users WHERE name = $1', [user], (error, results) => {
            if (error) {
                throw error
            }
            res.status(200).json(results.rows)
        })
    }
})

app.listen(8000, function () {
    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.

```

const pool = new Pool( config: {
  user: 'postgres',
  host: 'localhost',
  database: 'postgres',
  password: 'process.env.POSTGRES_PW',
  port: 5432,
});

var app = express();
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.get('/', function (req, res) {
  const search = req.params.q;

  if (search != "") {
    var squery = "SELECT * FROM users WHERE name = \"" + search + "\"";
    pool.query( queryTextOrConfig: squery,  callback: (err, res) => {
      console.log( message: err,  optionalParams[0]: res);
      pool.end();
    });
  }
});

app.listen( port: 8000,  callback: function () {
  console.log( message: 'Example app listening on port 8000!');
});
```

Copilot + GHAS

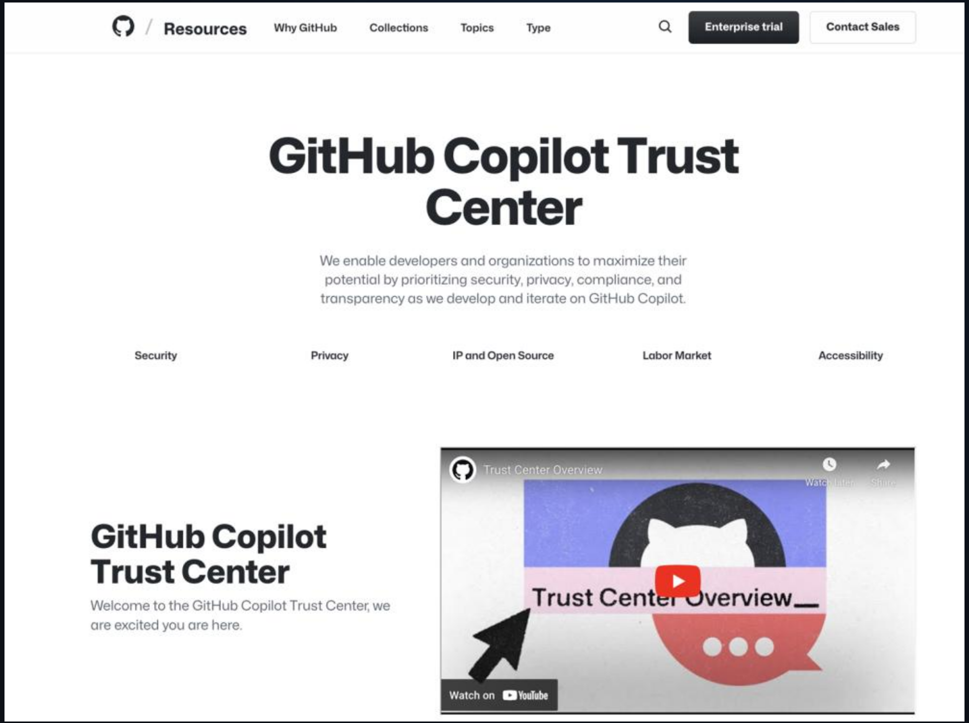
GitHub Copilot and GitHub Advanced Security (GHAS) serve different purposes and are not replacements of each other.

- 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

We enable developers and organizations to maximize their potential by prioritizing security, privacy, compliance, and transparency as we develop and iterate on GitHub Copilot.

GitHub Copilot Trust Center

Welcome to the GitHub Copilot Trust Center, we are excited you are here.

[GitHub Copilot Trust Center - GitHub Resources]

Wrap Up

Thank you