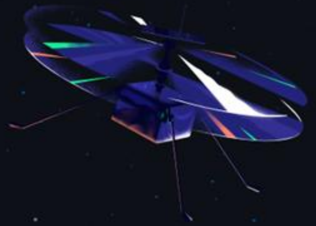
GitHub Copilot Training for Developers

Andrew Scoppa



AGENDA

GitHub Copilot - Introduction

Best practices & prompt engineering

In-class coding demos using copilot and copilot chat

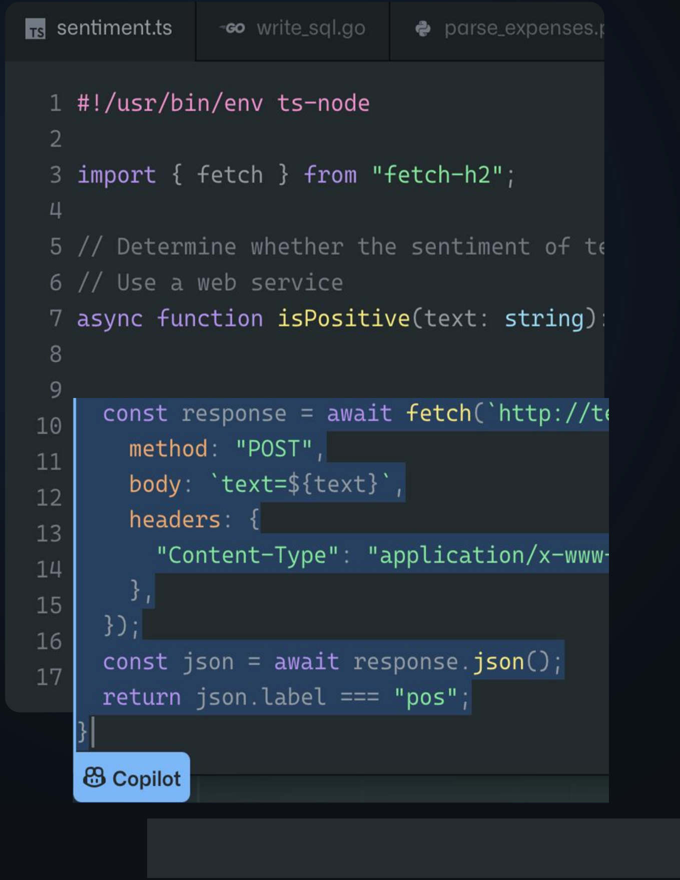
Secure coding

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
  - 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.



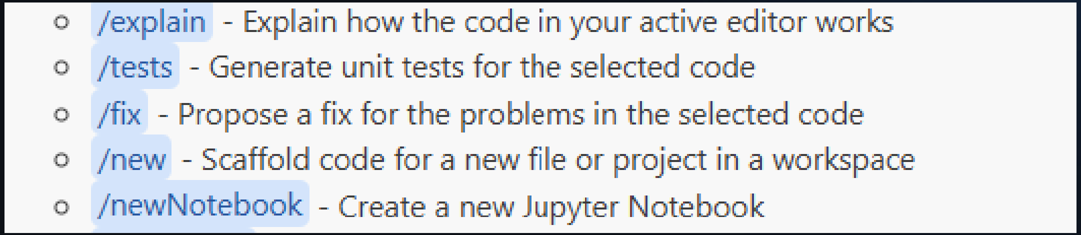
```
#!/usr/bin/env ts-node

import { fetch } from "fetch-h2";

// Determine whether the sentiment of te
// Use a web service
async function isPositive(text: string):

  const response = await fetch(`http://te
    method: "POST",
    body: `text=${text}`,
    headers: {
      "Content-Type": "application/x-www-
    },
  });
  const json = await response.json();
  return json.label === "pos";
}
```

Copilot vs Copilot Chat

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

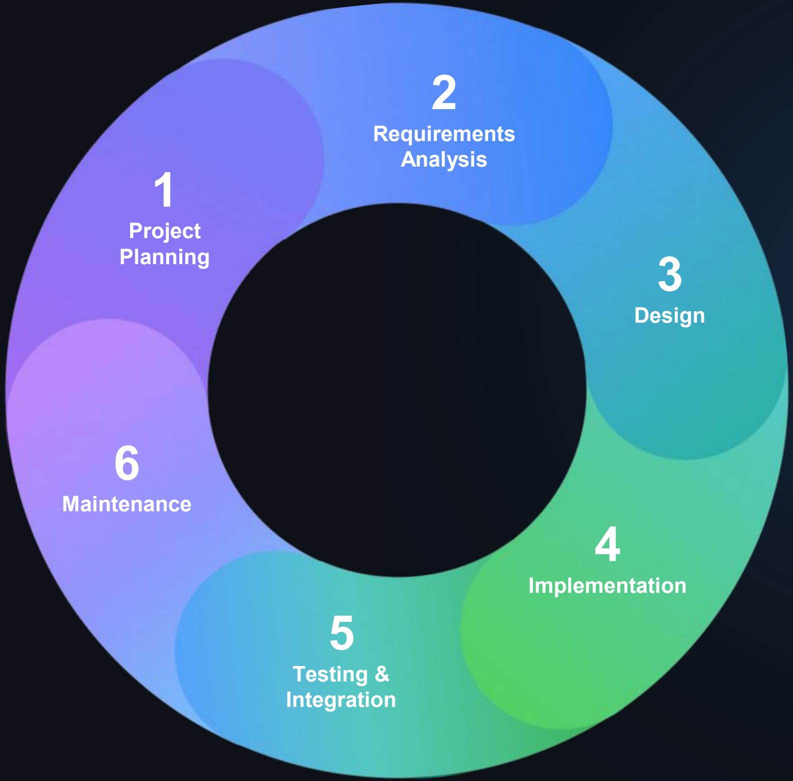
Slash Commands ("/")

- Provide quick access to specific functionality
- Perform common actions quickly without typing full instructions
- Supported by built-in and extension-provided chat participants



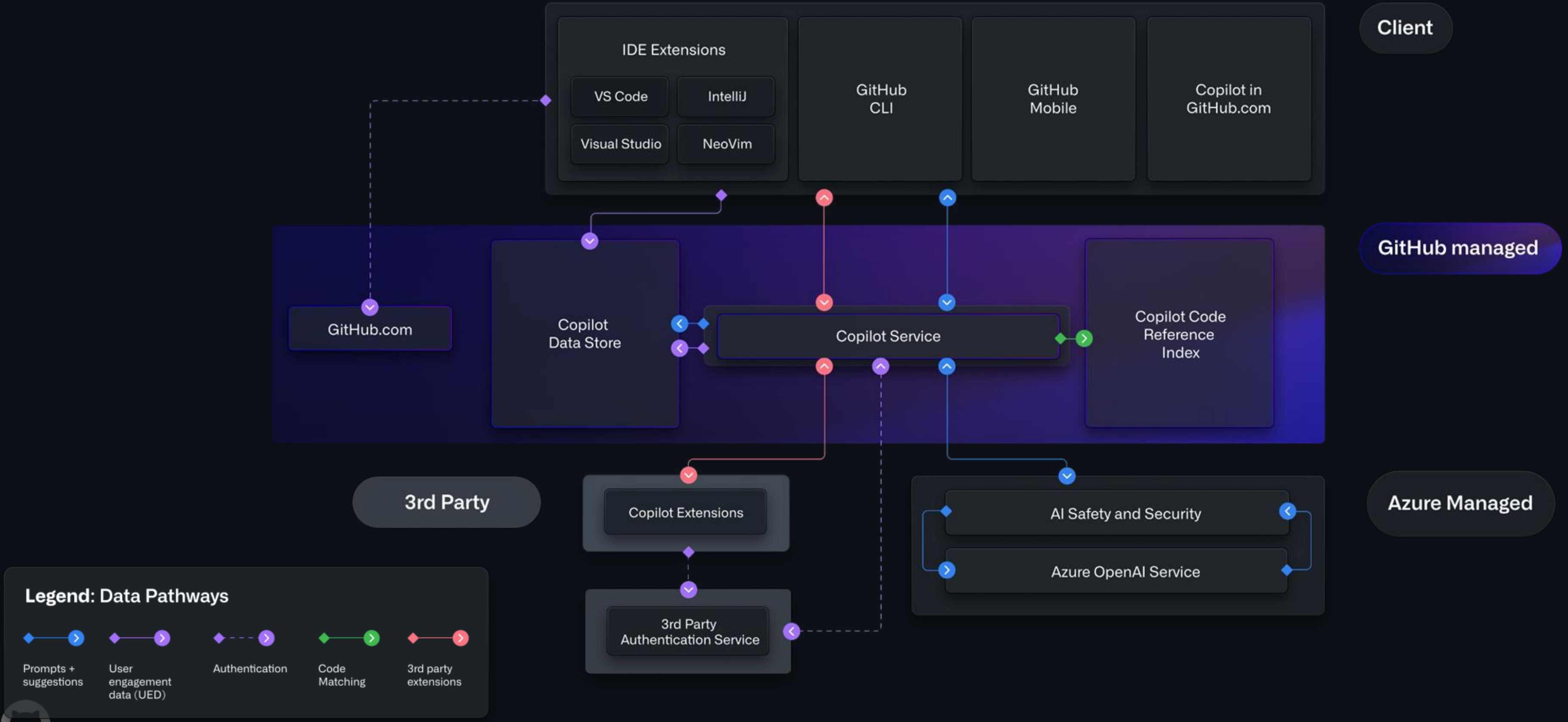
- /explain - Explain how the code in your active editor works
- /tests - Generate unit tests for the selected code
- /fix - Propose a fix for the problems in the selected code
- /new - Scaffold code for a new file or project in a workspace
- /newNotebook - Create a new Jupyter Notebook

GitHub Copilot + Chat



Helps developers stay in the flow throughout the entire SDLC

GitHub Copilot Architecture



Client

IDE Extensions

VS Code

IntelliJ

Visual Studio

NeoVim

GitHub CLI

GitHub Mobile

Copilot in GitHub.com

GitHub managed

GitHub.com

Copilot Data Store

Copilot Service

Copilot Code Reference Index

3rd Party

Copilot Extensions

3rd Party Authentication Service

Azure Managed

AI Safety and Security

Azure OpenAI Service

Legend: Data Pathways

- Prompts + suggestions
- User engagement data (UED)
- Authentication
- Code Matching
- 3rd party extensions

GPT Models

Copilot is always evolving and much of this evolution is powered by the evolving models behind it

Past (Codex)

- AI Hallucination problem
- Outdated/Deprecated code
- Lack of contextual understanding
- Failing to suggest best practices
- But it worked!

Present (GPT-3.5 GPT-4)

- Faster
- More efficient in regards to GPU usage
- Larger context window
- More recent training data

Future (Specific Models)*

- Code-specific models
- Enhanced experience and quality through fine-tuning
- Customization of models

* GPT keeps evolving and the latest can be found in github.com/blog

Prompt Engineering

What is a Prompt?

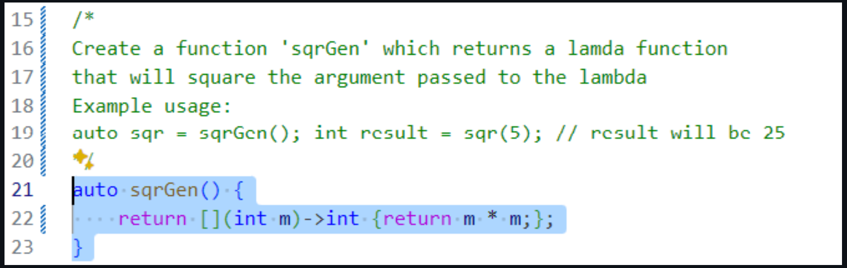
In the context of Copilot, a prompt is a piece of code or natural language description that is used to generate code suggestions. It is the input that Copilot uses to generate its output.

- 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.



```
/*
Create a function 'sqrGen' which returns a lamda function
that will square the argument passed to the lambda
Example usage:
auto sqr = sqrGen(); int result = sqr(5); // result will be 25

auto sqrGen() {
    return [](int m)->int {return m * m;};
}
```

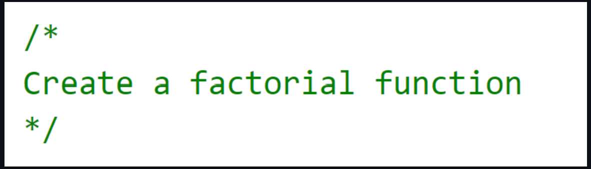
Providing Context to Copilot

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

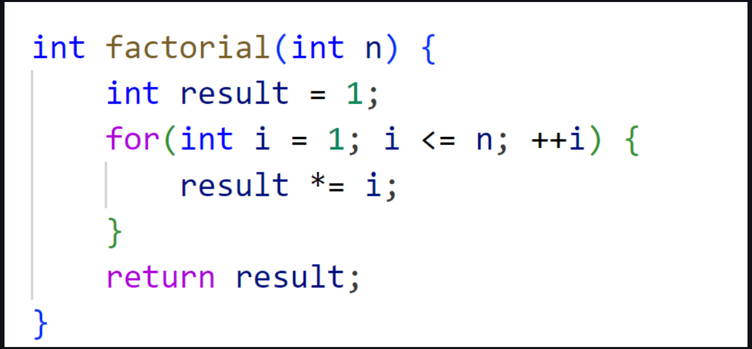
Strategic Prompt Engineering

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

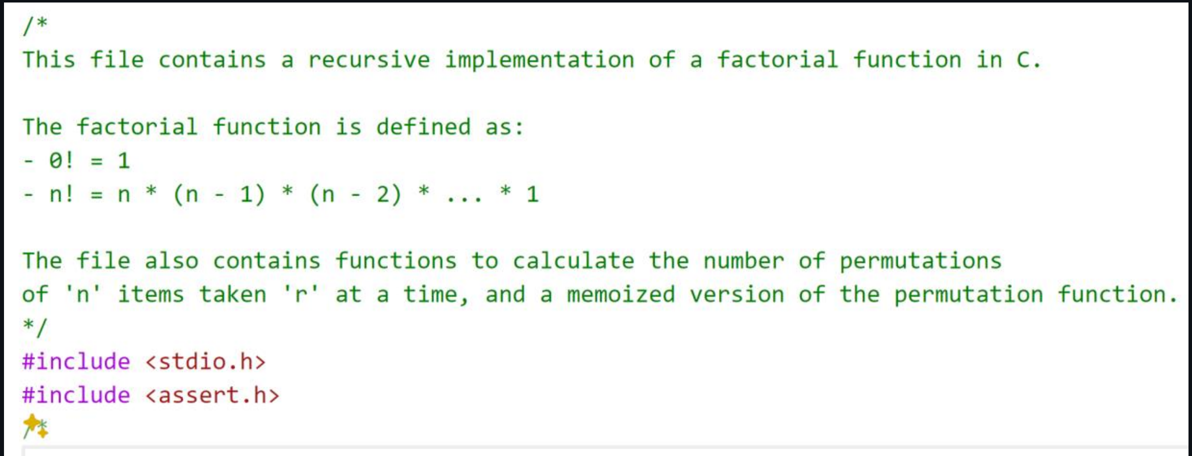
- While that is correct, it’s possibly not what you want!
- Instead:
- - You would like the function to be named named ‘fact’, have an
- unsigned int argument ‘n’, and return an unsigned long long
- - You also want the function to be recursive

Strategic Prompt Engineering

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

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

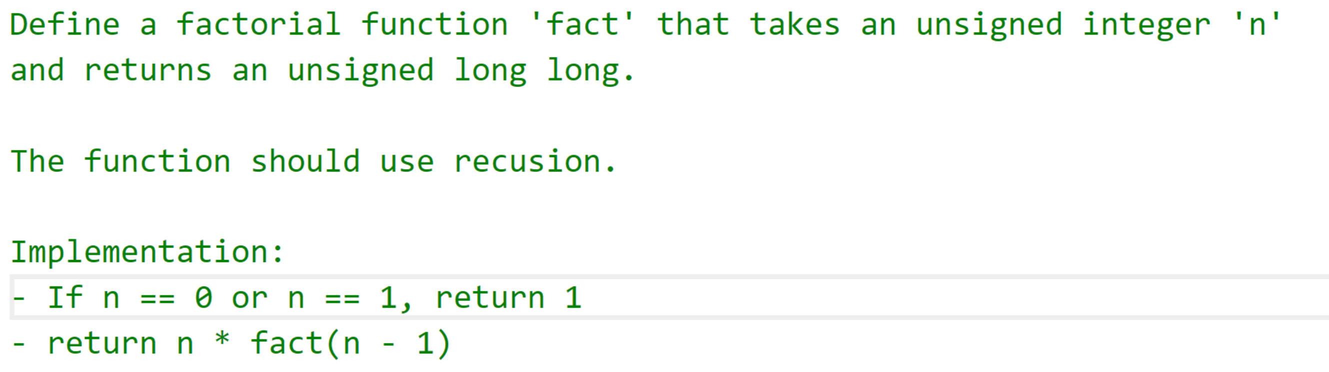
- Include the necessary header files you may be requiring.



```c
/*
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

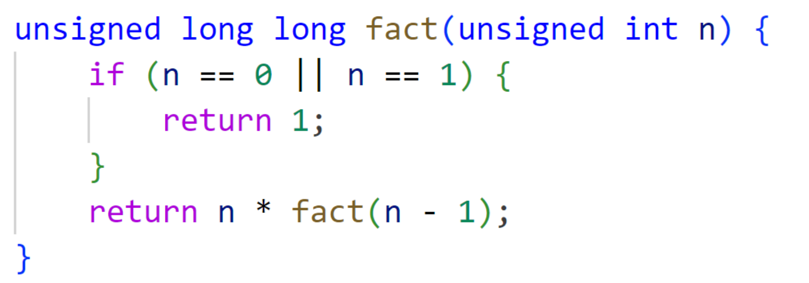
Rewrite the function comment providing as much detail as necessary:



```
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

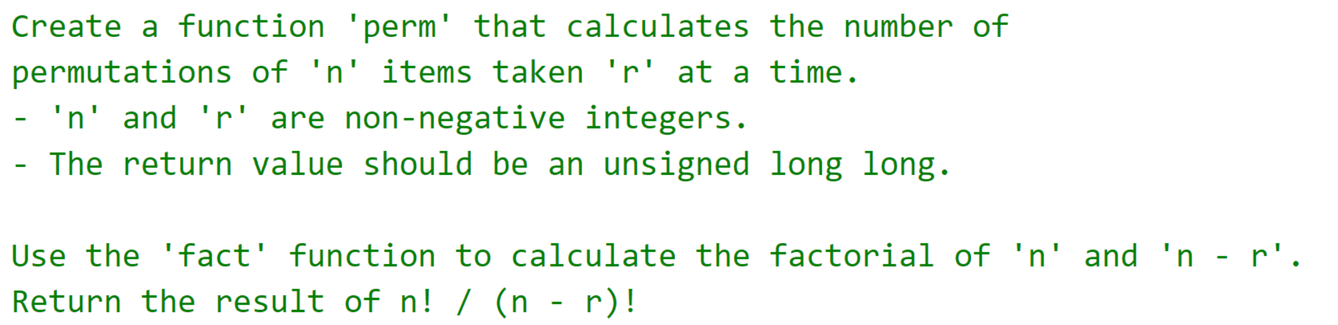
Now the resulting response is acceptable:



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

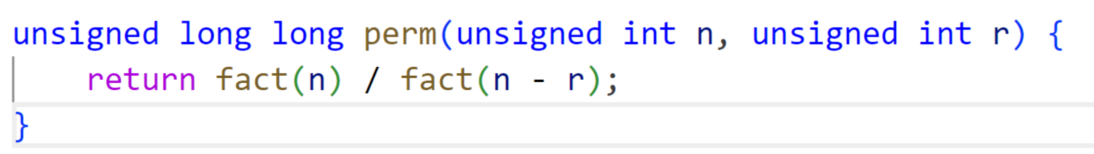
Strategic Prompt Engineering

- 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:



```c
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.
```

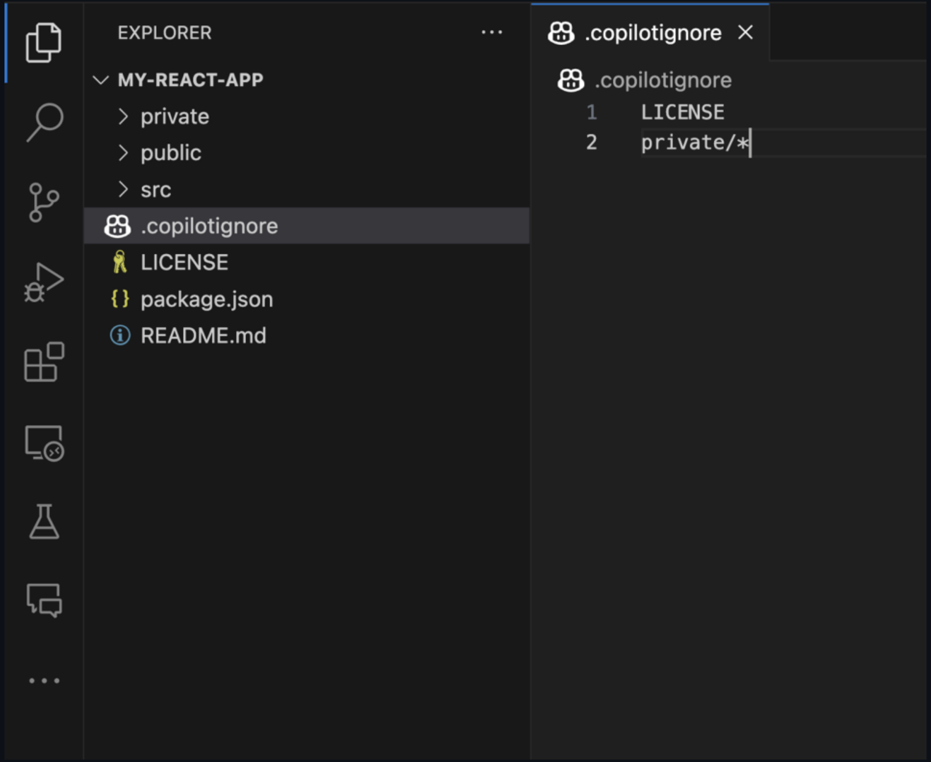
Helpful Patterns

```
* 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



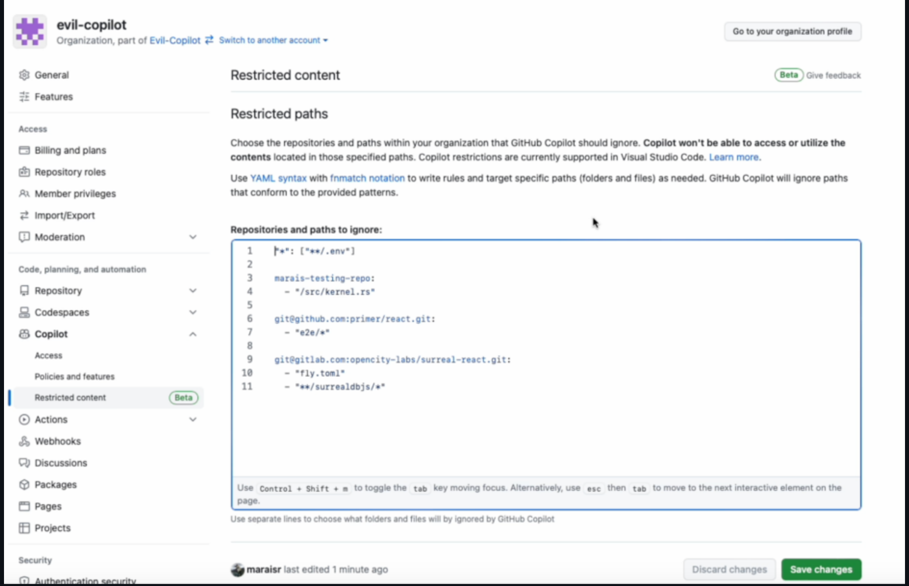
```
LICENSE
private/*
```

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.



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/*"
```

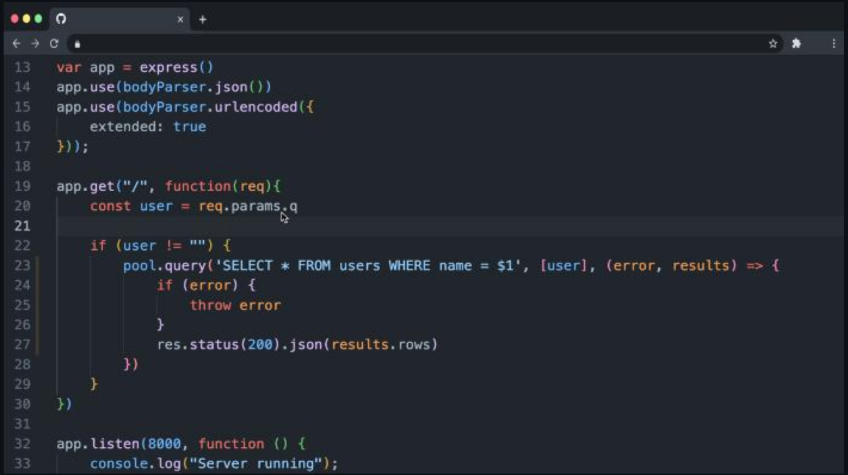
Use `Control + Shift + m` to toggle the `tab` key moving focus. Alternatively, use `esc` then `tab` to move to the next interactive element on the page.

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

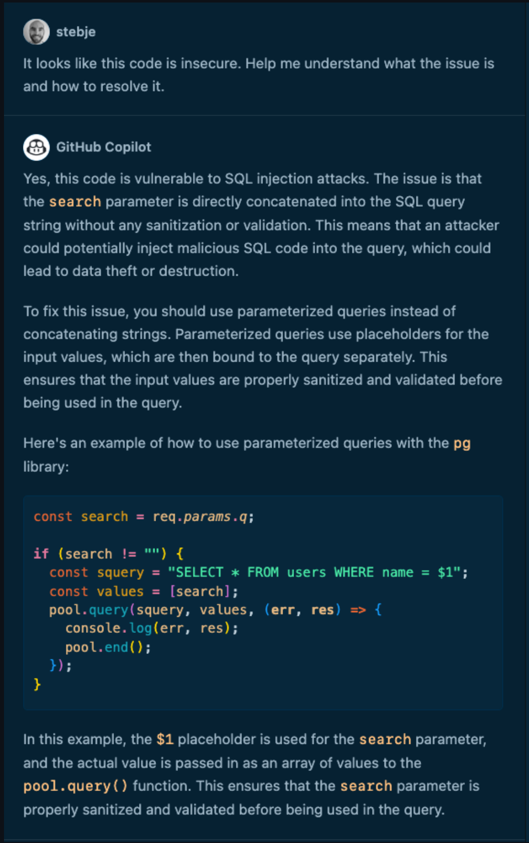
- AI-based vulnerability system that helps prevent insecure coding patterns (e.g. SQL script injection)
- Vulnerability filters are applied only to the Copilot-generated suggestion itself
  - ⊗ It cannot detect downstream vulnerabilities introduced by the code e.g. on deployment infrastructure
  - ⓘ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

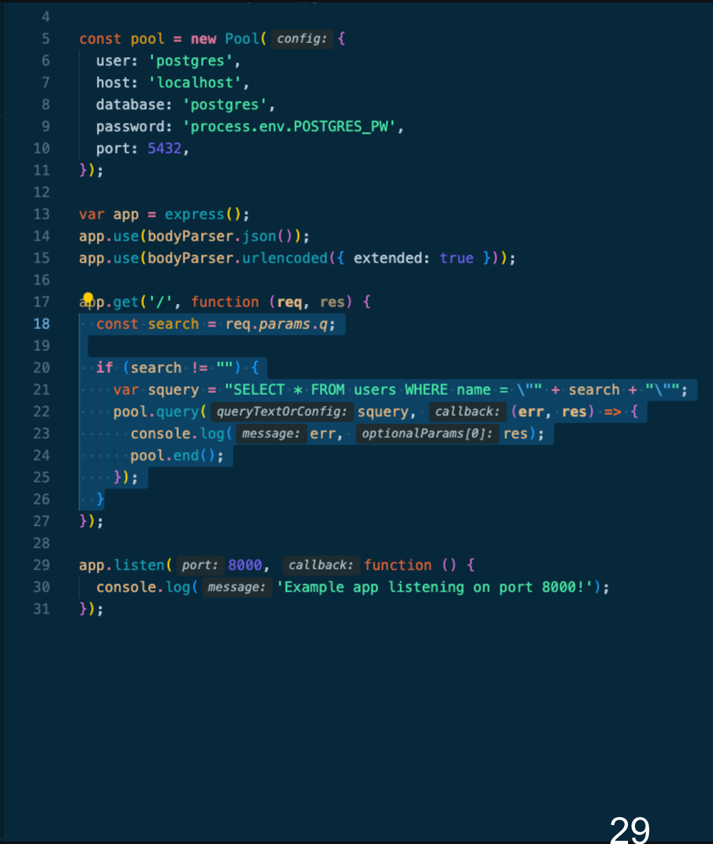
Yes, this code is vulnerable to SQL injection attacks. The issue is that the `search` parameter is directly concatenated into the SQL query string without any sanitization or validation. This means that an attacker could potentially inject malicious SQL code into the query, which could lead to data theft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

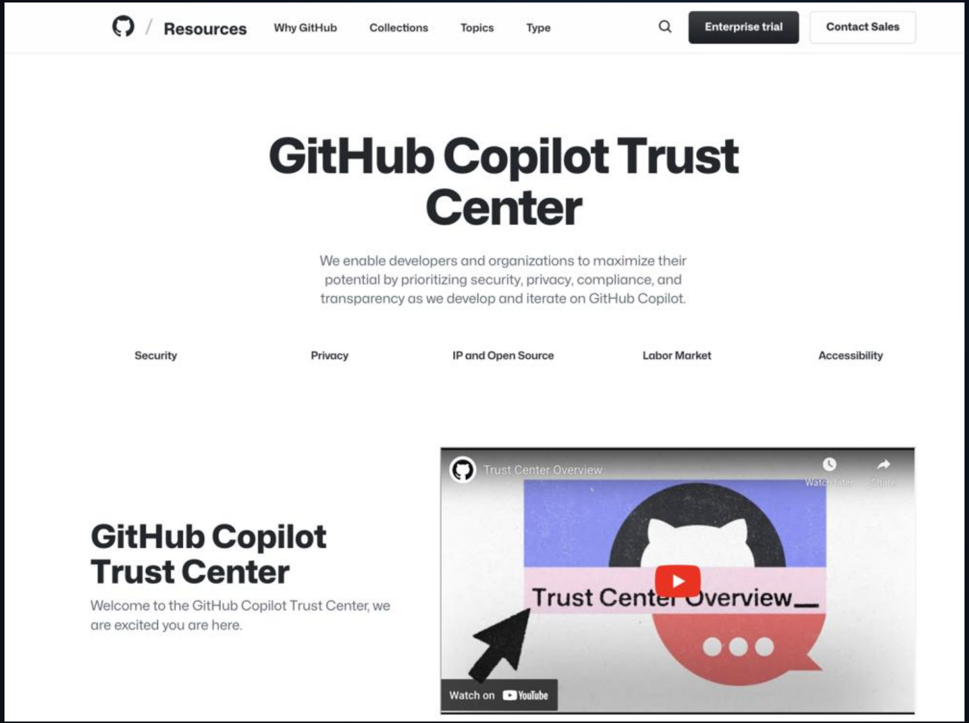
Copilot + GHAS

- 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