

GitHub Copilot for QA Professionals

Andrew Scoppa



Helpful Resources

- Getting started with GitHub Copilot
- Configuring GitHub Copilot in your environment
- GitHub Copilot Trust Center



AGENDA

GitHub Copilot - Introduction

Prompt Engineering

Quality Assurance

Detect and Remediate Vulnerabilities

Q & A





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 OpenAl
 - 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



Copilot vs Copilot Chat

Copilot

Direct Code Writing

Seamless IDE Integration

Solo Development

Copilot Chat

In-Depth Interactive Assistance

Learning & Teaching

Collaborative Scenarios

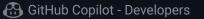


GitHub Copilot + Chat

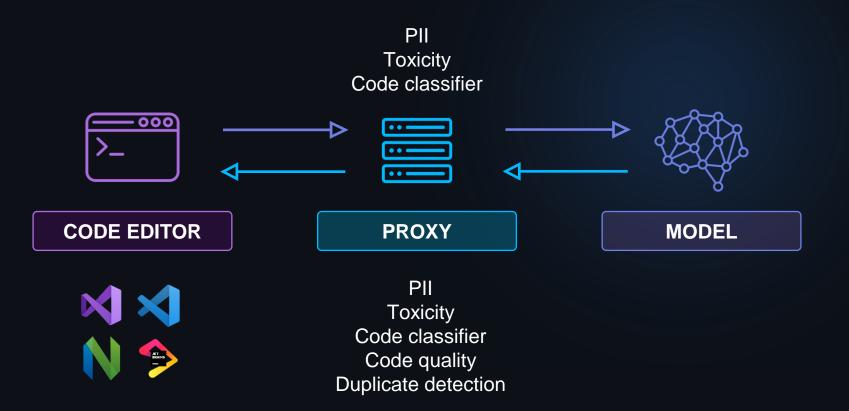


Helps IT professionals stay in the flow throughout the entire SDLC





Data flow through the Copilot ecosystem

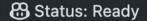


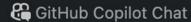


VS Code Settings

Configuration Options







Open Completions Panel...

Disable Completions

Disable Completions for 'markdown'

Edit Keyboard Shortcuts...

문화 Edit Settings...

Show Diagnostics...

Open Logs...

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

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

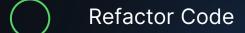
To help Copilot generate accurate suggestions:

- 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



- Generate Tests
- Debug
- Create Workspace
- Documentation

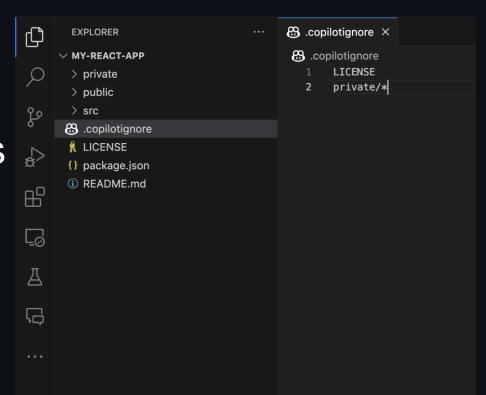


Detect and Remediate Vulnerabilities



Block files from Copilot

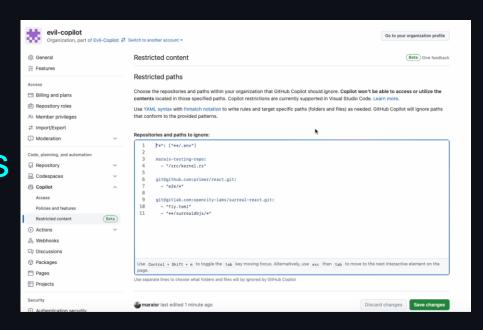
Use .copilotignore to block files and folders from being used by Github Copilot





Block files from Copilot

Ignore specific repositories and paths in Restricted Content





Copilot and Secure Coding

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

```
••• O
     var app = express()
     app.use(bodyParser.json())
     app.use(bodyParser.urlencoded({
         extended: true
     app.get("/", function(reg){
          const user = req.params.q
         if (user != "") {
             pool.query('SELECT * FROM users WHERE name = $1', [user], (error
                  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 SecretScanning 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: 'postares'.
 host: 'localhost'.
  database: 'postgres'.
  password: 'process.env.POSTGRES_PW',
 port: 5432.
var app = express():
app.use(bodyParser.ison()):
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

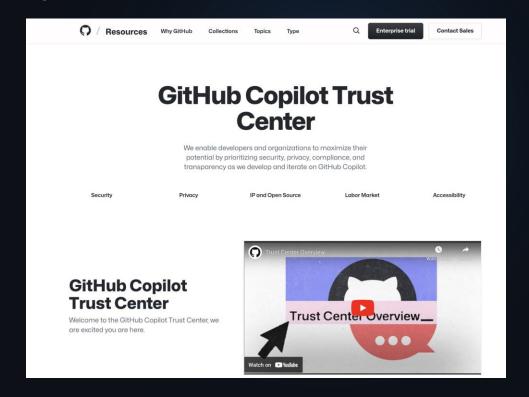
- Copilot is not a replacement of GHAS features
- Copilot can be used in tandem with GHAS features to detect and remediate vulnerabilities earlier during the SDLC
 - O GHAS Code scanning results
 - O GHAS Secret scanning



Security & Trust

Copilot Trust Center

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





Thankyou