()

GitHub Copilot Developers Training

Andrew Scoppa



Resources

- Getting started with GitHub Copilot
- Configuring GitHub Copilot in your environment
- Using GitHub Copilot Chat in your IDE GitHub Docs
- O Tutorial: GitHub Copilot and VS Code



GitHub Copilot - Introduction

Best practices & prompt engineering

In-class Demos

Secure coding

Wrap-up, Q&A



AGENDA





GitHub Copilot Fundamentals Recap



The software process can be broken down into two steps:

- 1) Design
- 2) Implementation

The first step is driven by you.

The second step is where Copilot can assist you with the development effort. Design





















GitHub Copilot

Helps developers stay in the flow throughout the entire SDLC

Refactoring code (code translate) Reviewing code (code explain) Documentation

Unit testing (TDD and BDD)
Finding code errors
Debugging
Code review
Al Pull Requests



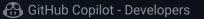
Convert comments to code Autofill for repetitive code Show alternatives



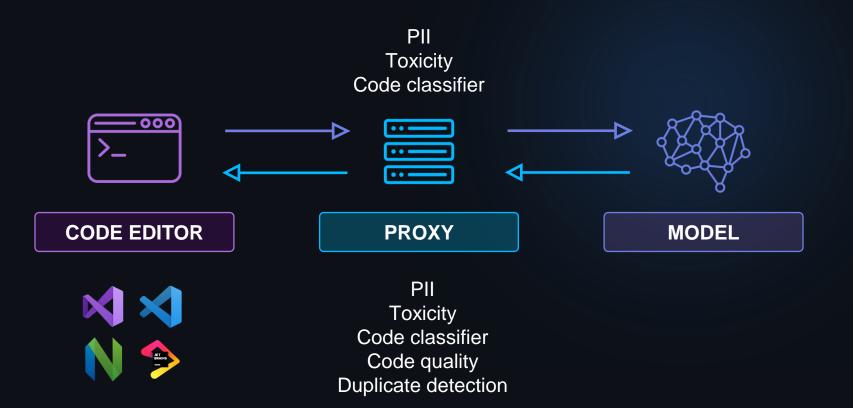
GitHub Copilot

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

```
sentiment.ts
                              parse expense
 1 #!/usr/bin/env ts-node
 3 import { fetch } from "fetch-h2";
 5 // Determine whether the sentiment of
 6 // Use a web service
 7 async function isPositive(text: string
     const response = await fetch(`http://
       method: "POST",
       body: `text=${text}`,
       headers: {
          "Content-Type": "application/x-ww
12
     const json = await response.json();
     return json.label === "pos";
17
    Copilot
```



Data flow through the Copilot ecosystem

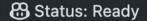


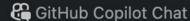


VS Code Settings

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



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



Prompting Techniques

In the realm of GitHub Copilot, Zero-shot, One-shot, and Few-shot prompting refer to guiding the AI with varying levels of examples.



Zero-shot prompting doesn't provide any prior examples, expecting Copilot to understand and generate relevant code purely from the given task description.

<u>One-shot prompting</u> provides a single example to set the context, assisting Copilot in generating a similar outcome.

Few-shot prompting involves offering multiple examples to establish a clearer pattern for the desired code output.

By understanding these techniques, developers can better instruct Copilot, ensuring more accurate and context-aware code suggestions.



If at first you don't succeed, iterate!

Zero Shot Prompt

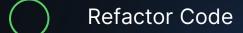
```
//Write a JavaScript function that finds
//the maximum value in an array.
function max(array) {
    return Math.max.apply(null, array);
}
```

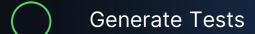
One-Shot Prompt



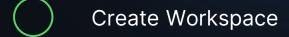
Using Copilot Chat

GitHub Copilot Chat









Documentation



Copilot vs Copilot Chat

Copilot

Direct Code Writing

Seamless IDE Integration

Solo Development

Copilot Chat

In-Depth Assistance

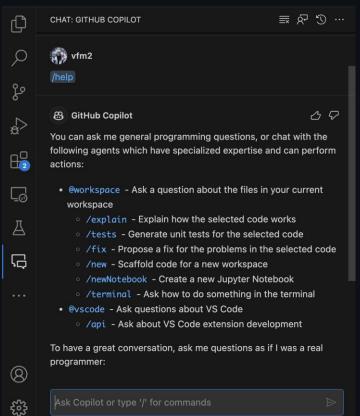
Learning & Teaching

Collaborative Scenarios



Copilot Chat: Slash Commands

/help to find available commands in your IDE





In-file Copilot Chat

Copilot offers in-file Copilot feature to selectively improve

```
sorting-algorithm.py 6
        SimpleDemo > ♦ sorting-algorithm.py > ⊕ insertionSort
              def guickSort(array) {
                   var pivot = array[0];
                   var left = []
                   for (var i = 1; i < array.length; i++) {
                      array[i] < pivot ? left.push(array[i]) : right.push(array[i]);
                   return quickSort(left).concat(pivot, quickSort(right));
               def bubbleSort(array) {
                      swapped = false:
                       for (var i = 0; i < array.length; i++) (
                              var temp = array[i];
                               swapped = true;
                   for (var i = 1: i < length: i++) {
                       for (var i = i - 1; i >= 8 55 array[i] > temp; i--) (
                      array[j + 1] = temp;
№ ⊗4∆2 ₩0
                                                                                                                                    Ln 30, Col 4 Spaces: 4 UTF-8 LF () Python 3.11.5 64-bit & (
```

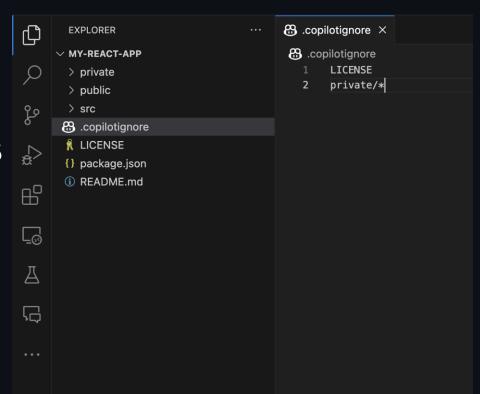


Secure coding



Block files from Copilot

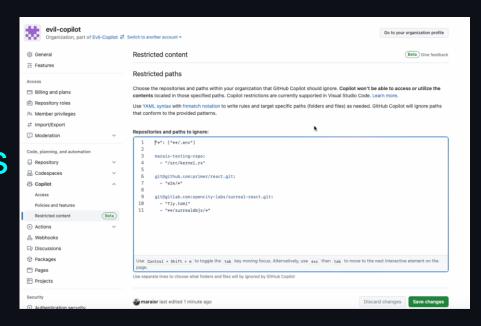
Use .copilotignore to block files and folders from being used by Github Copilot





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

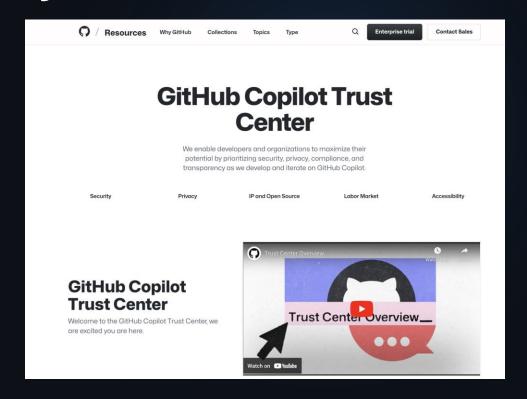
- Copilot is not a replacement of GHAS features
- Copilot can be used in tandem with GHAS features to detect and remediate vulnerabilities earlier during the SDLC
 - O GHAS Code scanning results
 - O GHAS Secret scanning



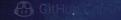
Security & Trust

Copilot Trust Center

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







Wrap Up

Thankyou