Best Practices with Prompt Engineering

Best Practices with Prompt Engineering for GitHub Copilot

On this page

Best Practices with Prompt Engineering

Getting accuracy closer to expectation with Prompt Engineering

Problems

1. Copilot fails to produce answer or keep repeating
2. Copilot generates incorrect result
3. Library/module version discrepancy
4. Copilot suggests non-optimal solution

Prompts and Prompt Engineering

Working at a project level

GitHub Copilot originally started by working with a single file. That is, as you start typing within a file, Copilot is there to assist by suggesting you with auto completion feature based on keywords you typed, whether it is in terms of comments, function name, file name, etc.

Although this is definitely helpful to increase programmers' productivity, as many developers can attest to this, many enterprises most of time do not work with a simple new project. Instead, they are likely to work with quite complex multiple projects, application services, and database services that interact with each other. Therefore, it is not surprising that many customers ask whether Copilot can do something with their existing projects. This ask can be about getting context from one service to another, suggest refactoring based on existing data or source codes, or generating entire suite of unit tests, integration tests based on already written 10,000 lines of codes spread out through in multiple files or even repositories.

Neighboring Tabs

If I have to name one type of questions that most customers ask about GitHub Copilot usage, it is about Copilot's context. That is, everyone wants to know where GitHub Copilot derives the context and make them more relevant. For example, we have these data SQL database table, can GitHub Copilot read from it and make suggestions? Or, we use this unit testing library called Mockito, so can Copilot helps to generate unit testings around that code testing framework?

Short answer, right now, is that GitHub Copilot yet cannot make out from the context from external services, external directories, or packages. Instead, GitHub Copilot can only pull the context based on literal keywords. There is maybe a possibility that GitHub Copilot later can be updated to pull data from other services, but this is not supported at the moment.

It is possible that GitHub Copilot can suggest based on what's opened in an IDE. For example, if you create a project named "CloudService" and then create a file like "deploy-to-azure.java," Copilot will likely start suggesting codes relevant to Azure cloud deployment. But there is no guarantee that GitHub Copilot will always pick up that context. So, how can we make them more relevant?

This is Neighboring Tab comes in. Neighboring Tabs is the technique that allows GitHub Copilot to process all of the files open in a developer's IDE instead of just the single one the developer is working on.

By opening all files relevant to their project, developers automatically invoke GitHub Copilot to comb through all of the data and find matching pieces of code between their open files and the code around their cursor—and add those matches to the prompt.

Copilot Chat

Earlier, it was explained that GitHub Copilot started with a feature that activates inside individual code files. That feature still exists and works perfectly well to accelerate your development process

and make coding more fun. However, working with existing codes and working at project level presented a challenge. This is where GitHub Copilot Chat comes in.

GitHub Copilot Chat is another extension within your IDE, and it lets you have a conversation with GitHub Copilot. And GitHub Copilot Chat is extremely helpful to work at project level. For example, you can ask it to start to create a ReactJS project with port exposed in port 3000, and GitHub Copilot can generate a series of steps with code samples or command samples.

In addition, GitHub Copilot is extremely help when you want to work with existing codes. You can highlight codes or refer files by mentioning it, then you can ask to analyze existing codes, to generate unit tests, or to suggest code refactoring, and GitHub Copilot Chat can output the result related to that ask.

Leveraging Copilot to increase code quality

GitHub Copilot originally started by working with a single file. That is, as you start typing within a file, Copilot is there to assist by suggesting you with auto completion feature based on keywords you typed, whether it is in terms of comments, function name, file name, etc.

Although this is definitely helpful to increase programmers' productivity, as many developers can attest to this, many enterprises most of time do not work with a simple new project. Instead, they are likely to work with quite complex multiple projects, application services, and database services that interact with each other. Therefore, it is not surprising that many customers ask whether Copilot can do something with their existing projects. This ask can be about getting context from one service to another, suggest refactoring based on existing data or source codes, or generating entire suite of unit tests, integration tests based on already written 10,000 lines of codes spread out through in multiple files or even multiple repositories.

GitHub listened, and there are now some features available in GitHub Copilot that can help to improve code quality. We will explore these techniques.

Create and Generate Tests

Probably the easiest way to improve your codes is by asking GitHub Copilot Chat to refactor your code. Refactoring is a programming practice of taking existing codes and making them better to be more reusable, more efficient, more testable, or object oriented.

As you can see in this example, I have number of different sorting algorithms like quick sort, bubble sort, and insertion sort. After highlighting my code blocks for quick sort, I ask GitHub Copilot Chat to refactor code to be space efficient, and GitHub Copilot Chat generated the result. Note that GitHub Copilot is very generative in nature, so the suggestion you get right now is most likely to be different than the suggestion you try next time.

But you may ask this question. What if I want to refactor dozens of functions in a file or even multiple files? Can GitHub Copilot help with that. Yes, it is possible – you can ask something like “in my file called ‘sorting-algorithm.py’, can you ask to refactor the functions to be more efficient?” But to GitHub Copilot, this is less ambiguous than just highlighting code and work with few lines of codes. So, the result is likely less satisfactory.

Another way to improve your code quality is by putting your codes under tests. These tests can be unit tests, functional tests, behavior tests, or performance tests. Either way, Copilot Chat is very efficient to generate unit tests based on codes you have.

In this example, I ask Copilot Chat to create unit tests for each function in my file named “sorting-algorithm.py”, and it generated unit tests based on Assertion method. Again, remember that Copilot is very generative in nature, so answer today might be satisfactory but answer you get next time might not be too satisfactory.

Slash Commands

But sometimes, even with Copilot Chat, you might not always get desired result because Copilot will likely “to guess” what you are looking for. For example, let’s say you ask “Help me to improve my code” in Copilot Chat. As humans, that is very clear to you. However, to Copilot Chat, that can be interpreted as number of different ways because of way it breaks into tokens.

To alleviate that, Copilot recently introduced / commands in Copilot. As you will see next, this is available in GitHub Copilot Chat as well as in-file features. In Copilot Chat, it has number of different options like /fix, /simplify, /tests, etc. Because those commands are dedicated to certain functions, you will likely get more accurate results based on those contexts.

One thing you will notice in this example, though, is that the result from GitHub Copilot Chat sometimes often greyed out with a message “Oops, your response got filtered. Vote down if you think this shouldn't have happened.” There is a limit to how many tokens that get returned from GitHub Copilot. So, what to do?

In-file Copilot Options

Copilot now offers more an advanced feature that you can select certain lines in files and ask to improve your codes. In this example, I highlighted my code block and CTRL + Touchpad click in Mac (right click in Windows). That will bring a menu option, so I selected Copilot option which presented different options.

From there, I can run / commands like /fix, /tests, etc just as I have done for Copilot Chat. While Copilot Chat gives a great way to work from larger project level perspective, this in-file Copilot option can be more accurate because it works with more focused scope.

There are time when in-file Copilot suggestion does not return any result, and this usually happens when you select a lot of code blocks within a file. Thus, use Copilot Chat if you work with large project level scope but use in-file Copilot suggestion when you want to get higher accuracy.

Production Layer

But you may ask “what about my codes in other file directories, GitHub, cloud, or server? Can Copilot help to scan through those and help to identify places where we can improve?”

GitHub Copilot right now only interfaces with 4 different types of IDEs, so that is your gate to interact with GitHub Copilot. However, GitHub Copilot will be integrated with GitHub platform, starting with GitHub Copilot Chat, GitHub Pull Request. In future, GitHub Copilot can be available in other GitHub features like GitHub Actions and GitHub Advanced Security, so stay tuned.

Addressing concerns around security exploits

Prompt Injection

One of most potential risks with using any Large Language Models (LLM) is a risk of security exploit. One of known attacks with LLM is called “Prompt Injection.” Prompt injection is a vulnerability in LLM where attackers use carefully crafted prompts to make the model ignore its original instructions or perform unintended actions. This can lead to authorized access, data branches, or manipulation of the model’s responses. In simpler terms, think of prompts as the questions or instructions you give to an AI. The way you phrase these prompts and the inputs you provide can significantly influence the AI's response.

Please take look at these articles (feel free to share):

- What is prompt injection
- Direct prompt injections

Because of these type of security exploits, it is extremely important that type of LLM you use enforce highly secure standards and governance procedures. Rest assured, GitHub Copilot implemented extensive filtering mechanisms to prevent prompt injection and other threats.

However, it is still very important for development teams to adopt other security prevention and coding practices to be extra secure.