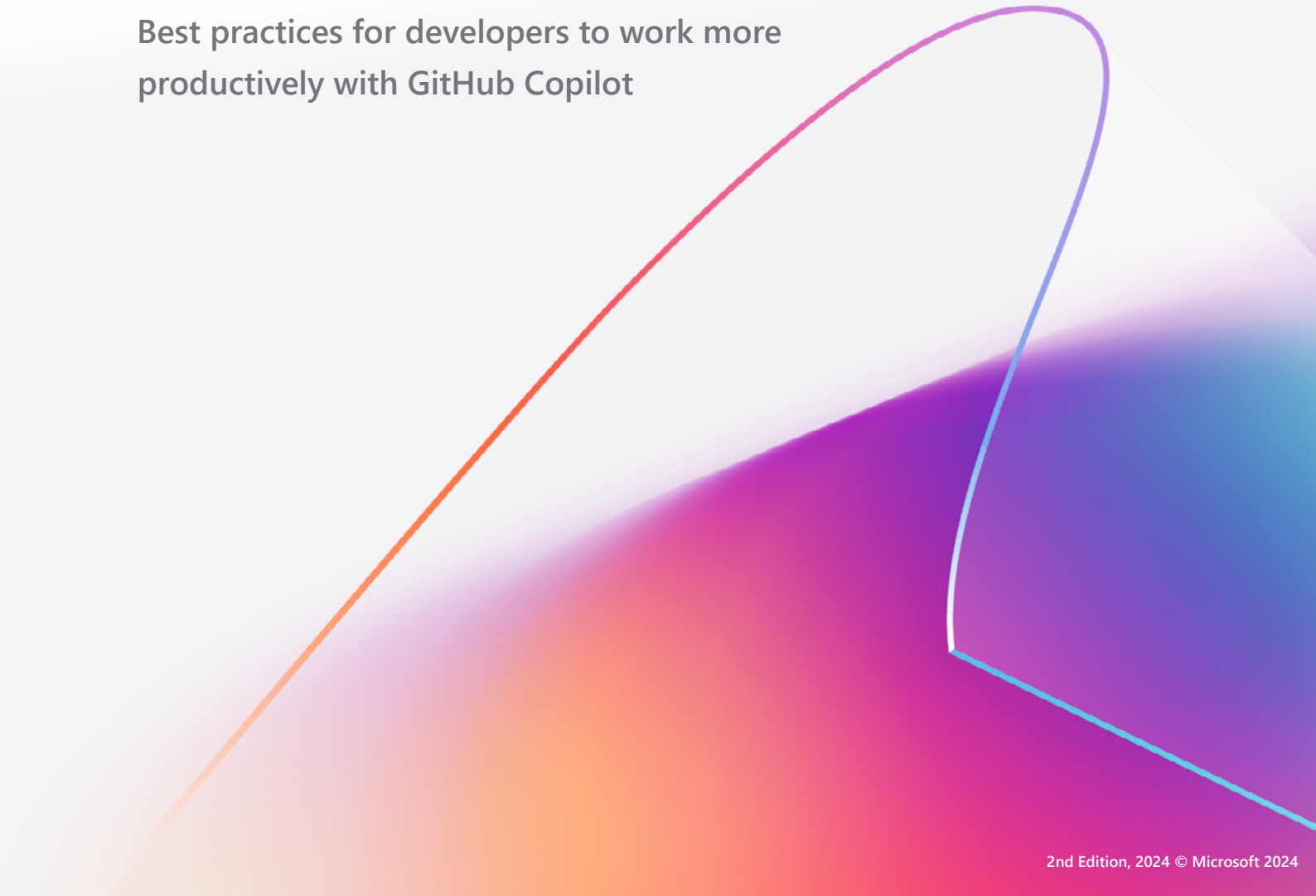
Microsoft Azure

GitHub Copilot

Tips & Tricks Vol. 2

Best practices for developers to work more productively with GitHub Copilot



Dennis Gassen

Foreword

During my tenure at Microsoft Germany, I have had the privilege of contributing to several e-books. The first e-book – if I recall correctly – was the Visual Studio Island Book 2013, featuring insights on Visual Studio, Team Foundation Server, Windows 8, and Windows Azure (as it was known back then). Subsequent e-books followed on topics such as Visual Studio, Visual Studio Code, and efficient remote collaboration among development teams shortly after the outbreak of Covid. For the latter e-book, Thomas Dohmke, who is now the CEO of GitHub, wrote about the cultural transformation at GitHub. The next wave of change is already underway – naturally, I am referring to GitHub Copilot.

Back in 2015, Satya Nadella, CEO of Microsoft, declared that every company would become a software company. Satya Nadella has reiterated this statement in various forms to this day. The essence always revolves around empowering development teams or so-called Citizen Developers to use their creativity to the fullest in solving problems and creating new products and solutions that provide value. Therefore, the focus is on enabling the creation of new things with technology and concentrating on what's essential.

And this is precisely where GitHub Copilot comes in. To bring you this new offering from the user's perspective, we decided it was time to publish another e-book featuring real-life tips and tricks about GitHub Copilot. The idea is to continually expand the e-book with new tips and tricks. If you would like to contribute to one of the upcoming editions, please feel free to email me at techwiese@microsoft.com. I would be delighted to hear from you.

Until then, I hope you enjoy reading the first edition of this e-book.

Dennis Gassen

Malte Lantin

As a Principal Solutions Engineer at GitHub, Malte Lantin supports strategic customers in Europe with the implementation of GitHub Enterprise as a central platform for secure software development and deployment.

Introduction: The future of software development with GitHub Copilot

by Malte Lantin

Just three years after GitHub's first internal whitepaper on AI-assisted software development in 2020, and the launch of GitHub Copilot in 2022, GitHub Copilot has already become a standard tool for many developers and enterprises. It's easy to forget that AI support in software development was once viewed as mere fiction. Today, GitHub Copilot represents a turning point in the development of software tools.

With continuous improvements and features specifically designed for professional use in businesses, GitHub Copilot has become an indispensable tool. This introduction explores its basic functionality and the added value it provides, which will be illustrated through numerous case studies in the following chapters.

Technological innovations and features

GitHub Copilot is built on years of research into language models, their secure deployment through Azure OpenAI Services, and seamless integration into the development process. Developers now have AI-powered code completion at their fingertips in Visual Studio Code, Visual Studio, the JetBrains IDEs, and other editors, drawing context from actively worked-on source files and information from the respective editor. This enables it to generate relevant and highly applicable code suggestions during the development process. The “Fill-in-the-Middle” (FIM) paradigm used allows for suggestions that are optimally adjusted to the current context and consider the style of the project.

By the end of 2023, the introduction of GitHub Copilot Chat significantly expanded the possibilities for AI-driven development. Integrated directly into the development environment, this chat interface now

allows for more complex instructions, advanced recommendations, code explanations, test generation, or even creating complete work environments by GitHub Copilot. With its iterative approach based on natural language, Copilot Chat is accessible and helps developers speed up software development, fix problems, and learn new technologies. The contextual knowledge that GitHub Copilot gains from the code editor is a key differentiator from other chat-based AI language models. GitHub Copilot Chat can access knowledge from requested files or command line statements, thus providing perfectly tailored suggestions at any time.

GitHub Copilot is continuously expanding and since spring 2024 has also been available as an extension of the GitHub CLI Tools to explain or generate commands on the command line.

Since February 2024, companies using GitHub Enterprise Cloud have also been able to benefit from an even deeper integration of GitHub Copilot into their development processes. This includes integrating GitHub Copilot Chat with the GitHub web interface, AI-powered editing of pull requests, access to the knowledge of their own source code repositories and documentation, and future customization of AI models to their own codebases. This support extends parts of the development process that occur outside the development environment, meaningfully enhanced by GitHub Copilot.

Impact on productivity and code quality

With over three billion lines of code generated, GitHub Copilot has proven its effectiveness and widespread acceptance within the developer community. Today, more than 1.8 million software developers use GitHub Copilot, and over 50,000 companies have opted to implement the technology. The impact of GitHub Copilot on developer productivity is both quantifiable and significant. In one study, it was found that over 30 percent of suggestions from GitHub Copilot are accepted, highlighting its relevance and utility in daily development. With a work speed increase of up to 55 percent, the benefits are clear. Equally important, however, is that developers report feeling more satisfied and productive when using GitHub Copilot, as surveys suggest. Studies also show that code quality improves, and code reviews are described as more purposeful and quicker. Developers report being able to work more focused for longer periods as GitHub Copilot helps minimize context switching and distractions.

The greatly enhanced development experience offers benefits beyond productivity gains. By automating routine and repetitive aspects, GitHub Copilot enables developers to focus on the more complex and creative aspects of their projects, increasing job satisfaction.

Conclusion

In summary, GitHub Copilot represents a significant milestone in the history of software development. It symbolizes the beginning of a new era of AI-assisted programming that significantly enhances developer productivity, improves code quality, and positively affects the overall well-being of developers. As GitHub Copilot continues to evolve and integrate more deeply into the software development process, its influence will grow, making it an indispensable tool in the field of software development.

The following chapters will describe practical applications and illustrate the everyday benefits of using GitHub Copilot. We hope you enjoy reading and using GitHub Copilot!

Further information:

- Copilot transforms GitHub into the AI-powered developer platform
- The economic impact of the AI-powered developer lifecycle and lessons from GitHub Copilot
- [1] Quantifying GitHub Copilot's impact on developer productivity and happiness
- [2] Quantifying GitHub Copilot's impact on code quality

Content

GitHub Copilot: Versions and features overview

GitHub Copilot introduces artificial intelligence into development, helping developers create better code faster. This AI assistant aids in writing individual lines or entire blocks of code and can suggest ways to expand or improve existing code. The result is a development process accelerated by more than half, as GitHub Copilot significantly enhances developers' workflows.

For example, with GitHub Copilot, you can write a comment that describes the desired logic and sets your preferred style conventions. AI handles the rest, allowing you to focus on the truly complex tasks. This enables you to work more productively and complete repetitive tasks much faster and with less stress. GitHub Copilot also helps developers master a new language or framework faster than would be possible through reading documentation or web research.

Support for numerous programming languages and the development of secure code

GitHub Copilot supports a wide range of programming languages, including C#, C++, Python, JavaScript, TypeScript, Go, and Ruby. More than 50,000 companies have already implemented this AI assistant, including a third of all Fortune 500 companies.

GitHub Copilot also enhances team collaboration. It indexes your repositories, understands the code stored within, and helps you get up to speed with new codebases more quickly. When you connect new repositories, you can build upon the work of others. Another application of GitHub Copilot is in creating functions and diagrams that developers specify via a prompt. GitHub Copilot avoids vulnerabilities and security gaps right from the code generation phase, as the system is optimized to generate secure code. Unsafe lines of code are blocked by AI.

Real-time AI-based suggestion: More security, fewer errors

The strength of GitHub Copilot lies in providing real-time, AI-based suggestions for code development. As you write lines of code, GitHub Copilot analyzes the code and automatically suggests completions. This creates perfect harmony between the developer and AI. Developers can also use GitHub Copilot to automatically create commit messages. There are many applications that significantly ease developers' daily lives.

GitHub Copilot not only helps in creating new code or improving existing lines of code. The AI assistant is also a valuable tool for finding errors in code based on a "conversation" in natural language. If you get stuck, just ask GitHub Copilot. GitHub Copilot acts intelligently, personalizing responses based on your expertise and available documentation, from which the AI models continue to learn.

You don't have to leave your preferred development environment to use it. You can integrate GitHub Copilot directly into your development environment through extensions, for example in Visual Studio, Visual Studio Code, Vim, Neovim, JetBrains Suite, and Azure Data Studio. GitHub Copilot can also assist in the CLI to provide help. Since 2024, GitHub Copilot has also been available for GitHub Mobile, allowing developers to work with GitHub Copilot on smartphones and tablets just as they would on a desktop computer.

GitHub Copilot: The right version for every application

There is a suitable version of GitHub Copilot for every application. Individual developers benefit just as much as small and medium-sized businesses or large organizations.
The latter choose either GitHub Copilot Business or GitHub Copilot Enterprise. For individual developers, there is GitHub Copilot Individual. All versions have one thing in common: they convert natural language into code directly in the IDE.

GitHub Copilot Business is primarily aimed at businesses. For $19 per user per month, developers in the company gain access to GitHub Copilot, including code completions, chat with GitHub Copilot over the IDE and GitHub Mobile, CLI support, filters for security gaps and public code, enterprise-level security and privacy, and code referencing. When GitHub Copilot recognizes vulnerable coding patterns, it blocks the lines. This also applies to the use of public code.

GitHub Copilot Enterprise offers additional features to the Business version, significantly increasing

the value for businesses. These include enhanced chat tailored specifically to the repositories of the respective company and support for pull request diff analysis for more precise code reviews. In code completion, users benefit from specially tailored models to further enhance the efficiency and precision of code creation. GitHub Copilot Enterprise can be used not only in IDE, CLI, and GitHub Mobile but also on GitHub.com.

GitHub Copilot Individual is aimed at individual developers who want to complete code. It also includes a chat with GitHub Copilot. This version is particularly suitable for individual developers and freelancers. It is also available to students, teachers, and supervisors.

Copilot does not copy code, it recreates it

Generally, none of the GitHub Copilot versions copy code directly; they generate it anew based on probability calculations. Although the models used by GitHub Copilot are trained with code from publicly accessible repositories, they do not contain code themselves. In generating code, GitHub Copilot utilizes the existing code in the IDE, especially the lines above and below the cursor. This includes information from training data and other files that you have linked with GitHub Copilot and the respective repository. Based on this data, GitHub Copilot generates the next lines of code or makes suggestions for them.

A look into the future of GitHub Copilot

The functionality of GitHub Copilot is continuously expanded, so you can regularly look forward to new innovative possibilities. At Microsoft Build 2024, for example, GitHub Copilot Extensions were introduced, allowing partner technologies to integrate directly with GitHub Copilot Chat. Early partners include DataStax, Docker, LambdaTest, Microsoft Azure, MongoDB, Octopus Deploy, Sentry, and Stripe.

A possible scenario for using the extensions: You need to fix a database error in your development project. Through GitHub Copilot Chat, you can use various tools and technologies such as Sentry or Azure to analyze and fix the problem without having to switch contexts.

This is just one of many examples of how GitHub Copilot is being developed. Current updates are always introduced on the GitHub Blog.

Martin Brandl

Martin (@martin_jib) works as the CTO and Managing Director at white duck GmbH. He has over a decade of experience with Microsoft Azure, with a passion for developing cloud-native applications. Due to his contributions to the Azure Community, he has already received several awards as a Microsoft MVP for Azure.

Tip 1:

Using GitHub Copilot

Slash Commands in GitHub Copilot

by Martin Brandl

GitHub Copilot's **Slash Commands** make interacting with the assistant even more efficient.
By simply tapping a slash ("/") in the chat a list of useful commands appears, allowing you to perform various actions directly, without much typing.

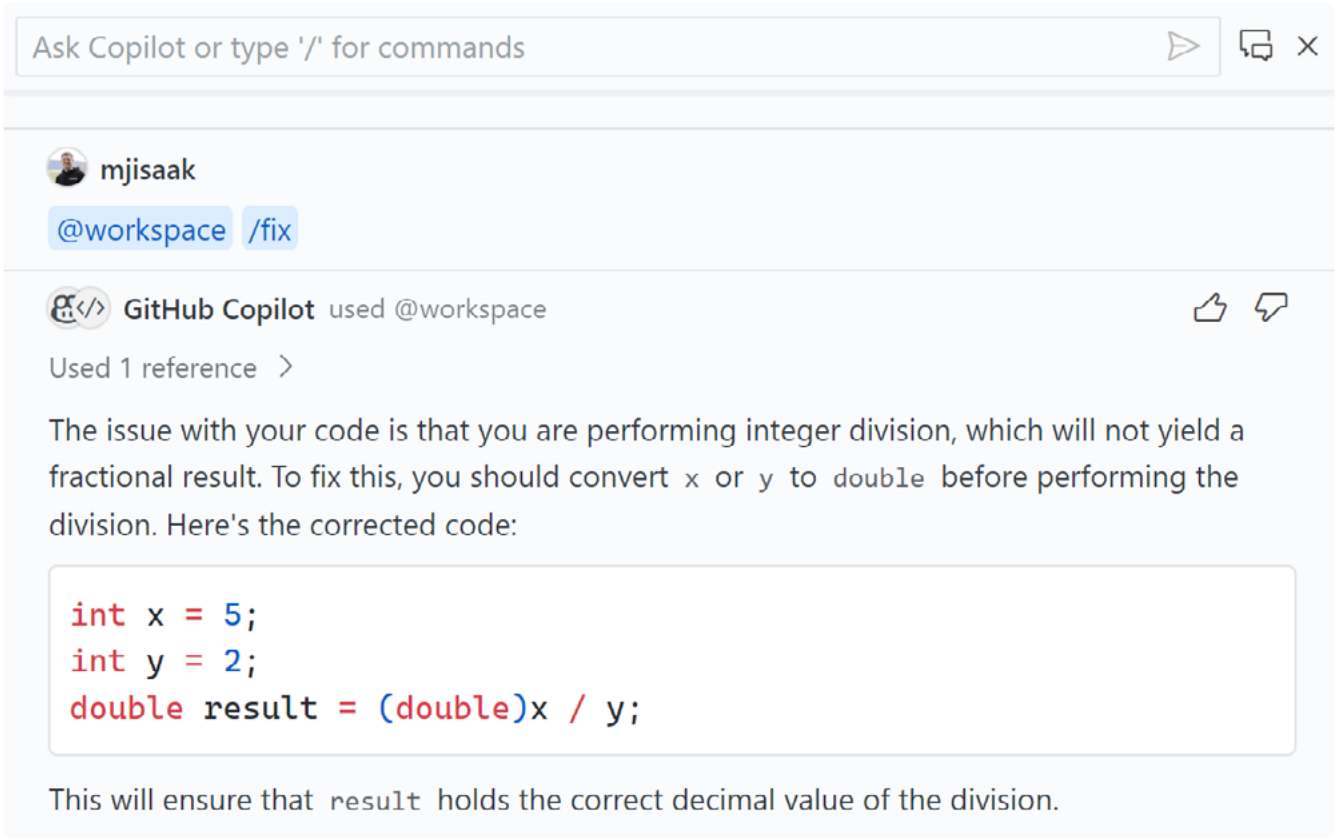
The commands are context-sensitive and always relate to a so-called agent.
They are indicated with the @ symbol. Currently, there are two:

- **@workspace** has context information about the code in your workspace and can help you navigate by finding relevant files or classes.
- **@vscode** knows commands and functions in the VS Code editor itself and can help you use them effectively.

Example: In the following code snippet, an integer division is performed, where the decimal places are truncated. The result is 2.0, instead of the expected 2.5:

```
int x = 5;
int y = 2;
double result = x / y;
```

However, this error can be easily fixed by running the **/fix** command in the GitHub Copilot chat. The @ workspace agent is automatically specified in this case:



In addition to the **/fix** command, you have access to a number of other useful commands that you can call up with **/help:**

- **@workspace /explain**: This command explains the selected code step by step.
- **@workspace /fix**: This command suggests a solution for any errors in the selected code.
- **@workspace /new**: This command allows you to create a new project based on a natural language description.
- **@workspace /newNotebook**: This command enables the creation of a new Jupyter Notebook based on your description.
- **@workspace /terminal**: This command provides explanations on how to use the integrated terminal for various tasks.
- **@workspace /tests**: This command automatically generates unit tests for the selected code.
- **@vscode /api**: If you have questions about developing VS Code extensions, this command is available to you.

#Developer #ReduceTechnicalDebt
#GitHubCopilotChat

David Losert

David is a Principal Solutions Engineer at GitHub, where he supports enterprise customers with his over 10 years of experience as a developer and cloud consultant in optimally establishing GitHub as an AI-driven DevSecOps platform.

Tip 2:

App migration between programming languages with GitHub Copilot Chat

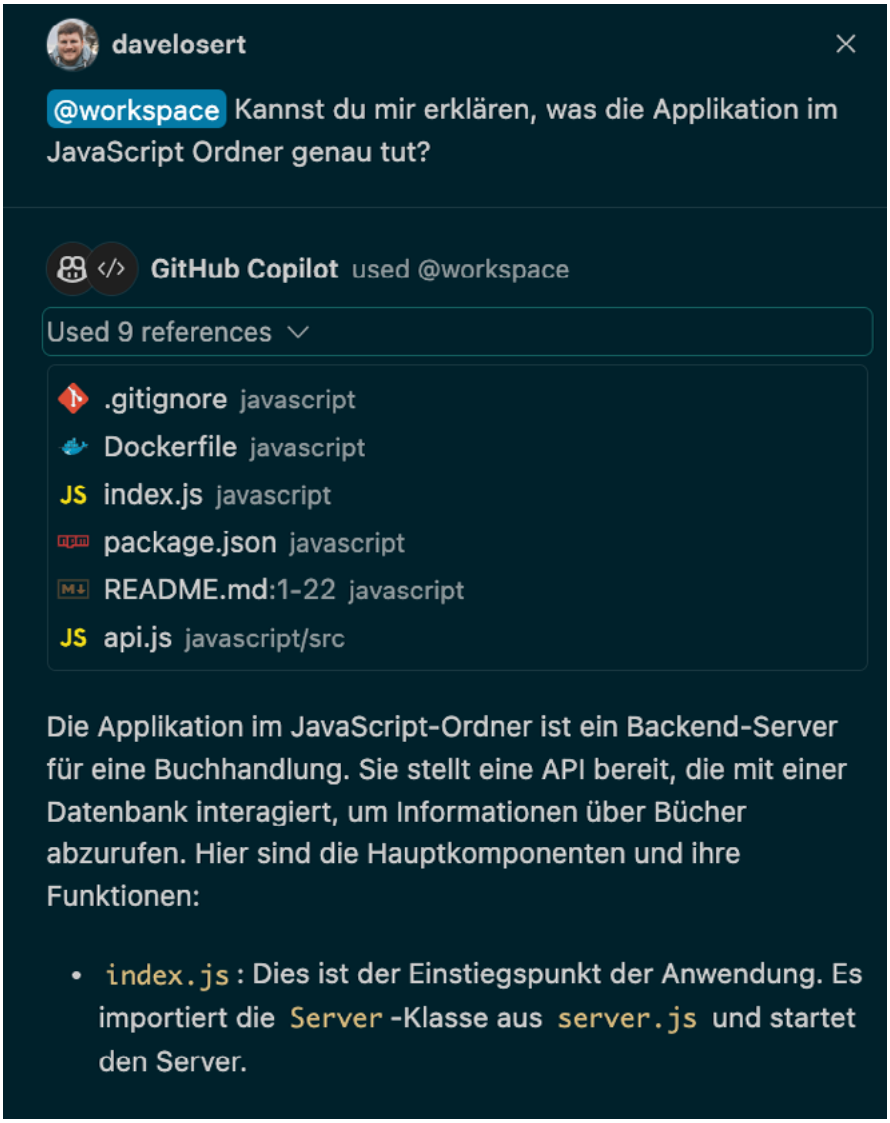
by David Losert

From standardization and modernization efforts over a lack of developer expertise to the desire for company-wide collaboration across team boundaries (keyword: innersourcing), there are many good reasons to translate an existing application or parts of it from one programming language to another. Especially in enterprises and after about a decade of technological diversification – driven not least by microservices, containerization, and the simplicity of technology-agnostic operations in the public cloud – we are now seeing a reverse movement and a desire for more standardization and uniformity. Fortunately, translation or migration between two programming languages has become much easier thanks to advances in artificial intelligence in software development, led by GitHub Copilot. Here's how it works.

Preparation

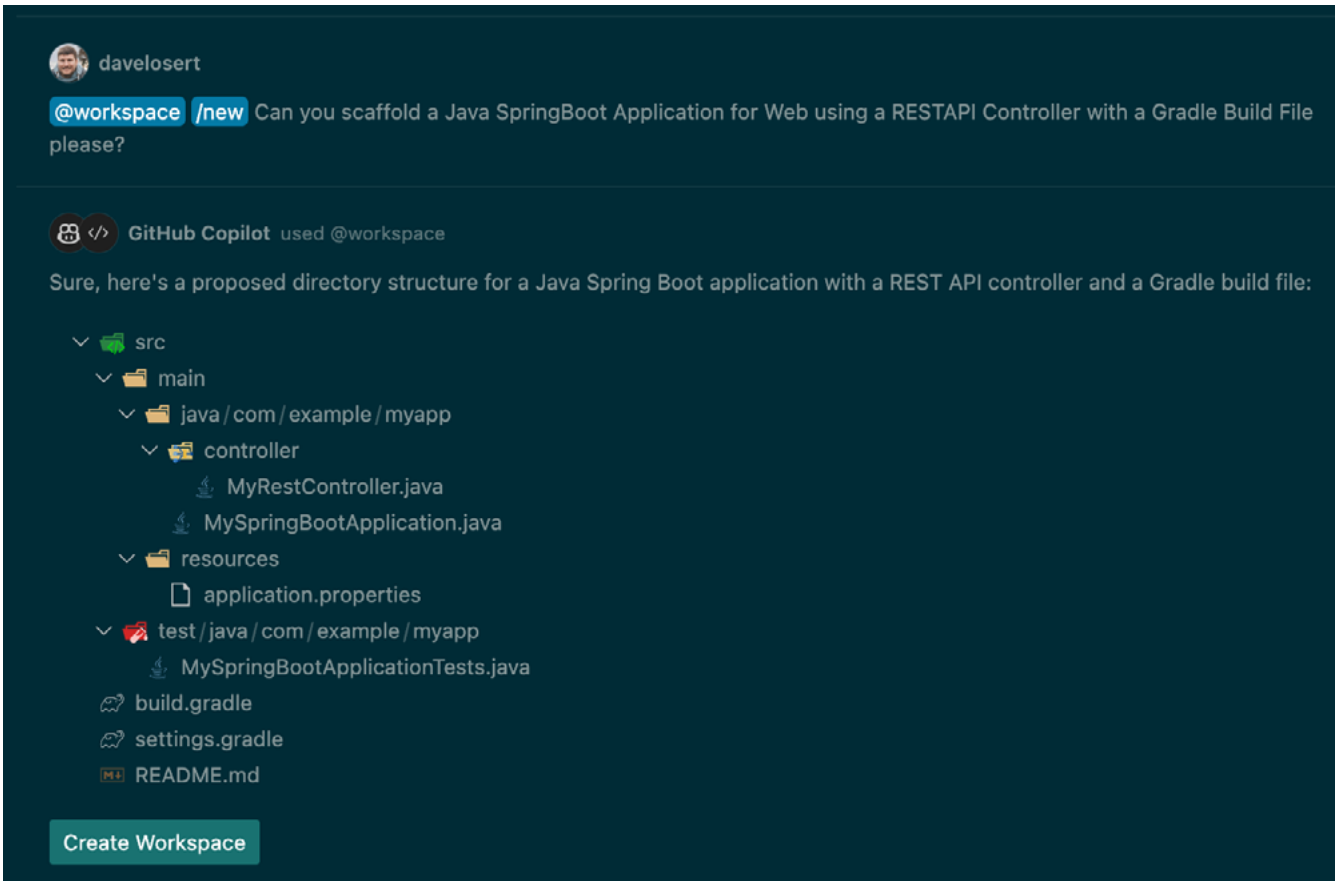
Before you can start translating, you need to define the target technology and the target architecture, because even with GitHub Copilot, the destination of the journey must be determined and the path well planned.

It is also advisable to gain a basic understanding of the source application to avoid migrating blindly. The GitHub Copilot Chat with the so-called @workspace agent can already be very helpful for this. It's an agent which automatically includes relevant files to the context processed by GitHub Copilot. This way, you benefit from explanations and a better understanding of a larger number of files.



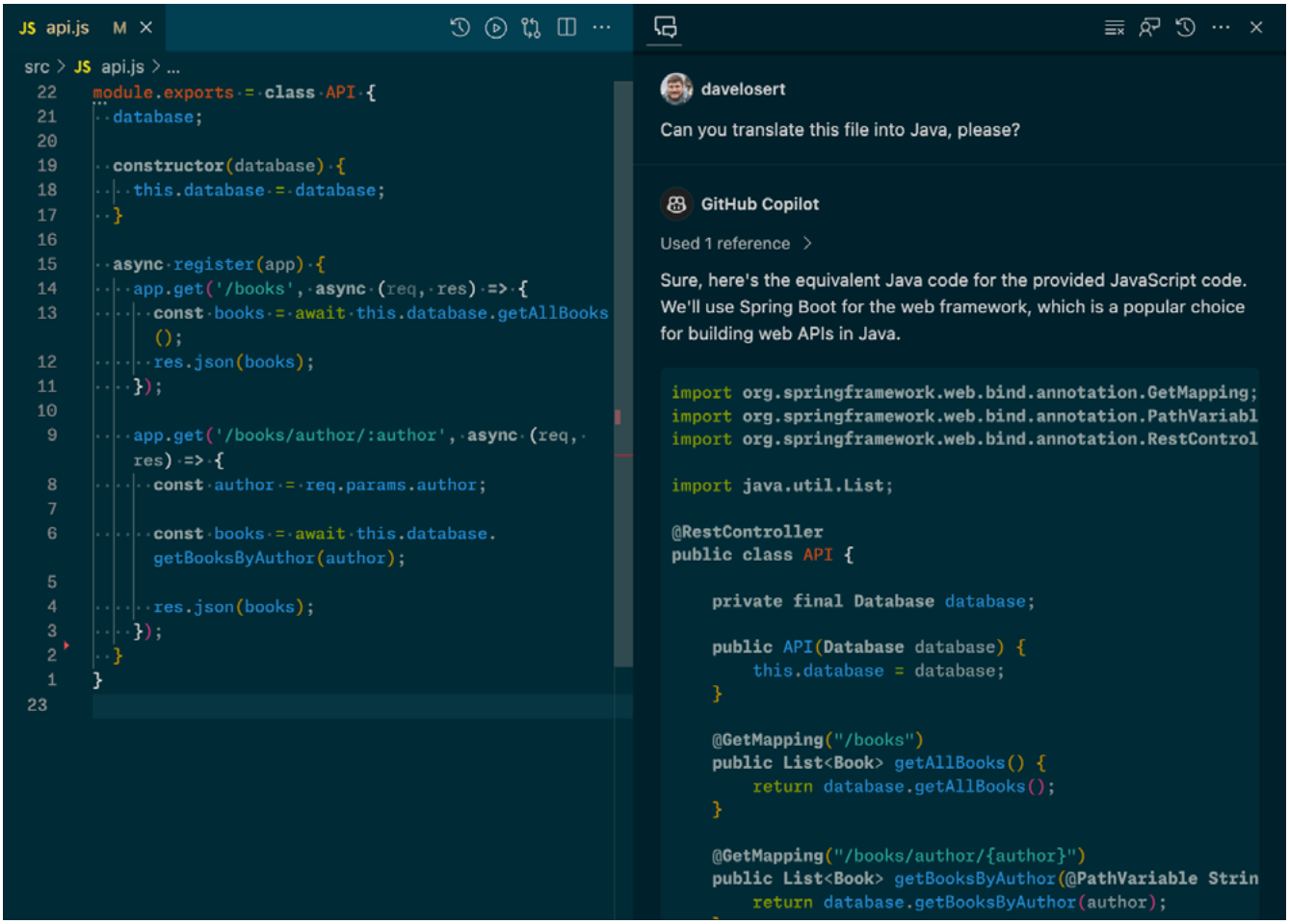
As the screenshot shows, you can also communicate with GitHub Copilot in German. Since the code is usually defined in English, and to make the whole thing consistent, the following screenshots were created with English texts (Note: This article was first published in german, hence the german example). The target stack can also be initialized with GitHub Copilot Chat and the slash command **/new.**

Migration



The actual migration happens directly in the IDE, executed by the developers and is very simple:

1. You open a file to be translated.
2. Within the GitHub Copilot chat window, you request the translation: "Translate this file into <Target-Programming-Language>, please!" (it doesn't hurt to be polite to AI 😉)



3. Copy and paste the generated code or insert it directly into the destination with the option "Insert into new File".

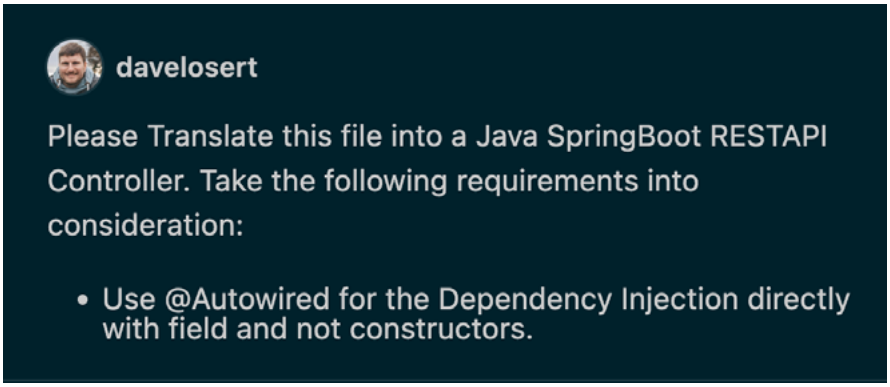
Repeat this process for all files in the source system to successfully complete the migration. IOf course, it's often not quite that simple and there are a few things to consider, which I will discuss in more detail below.

Review & Refinements

Usually, after translating individual files, smaller or larger improvements are still required on the target file GitHub Copilot helps with about 80 to 90% of the initial translation, but especially in the beginning, the target code might not be perfect just yet.

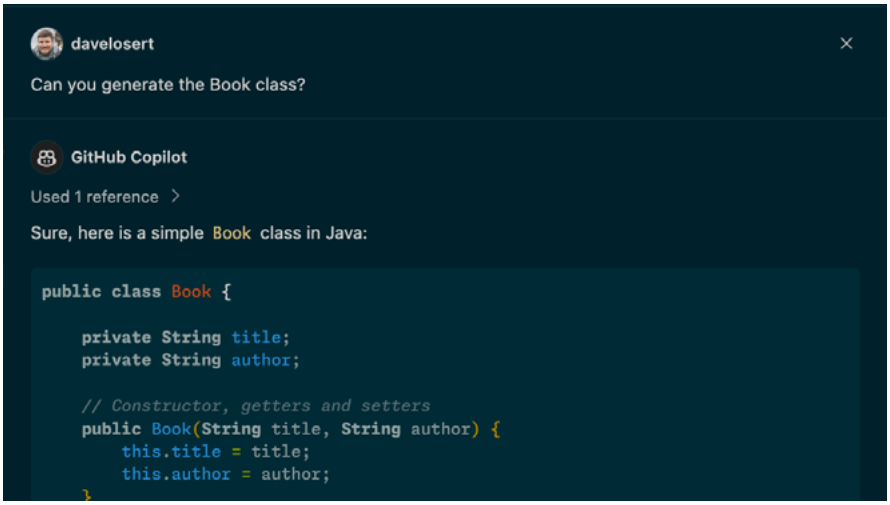
Consider the example above using Spring Boot with the @Autowired annotation. Actually, there is a nicer way to inject dependencies into the classes. In this case, use the chat to improve the target code:

To avoid having to perform this refinement with each new translation, you can refine the translation prompt directly for the next file to be translated:



Like that, you will get increasingly better translations from GitHub Copilot from the initial prompt over the course of the migration. Quick tip: With “Up Arrow,” you can bring up previous prompts for improvement into the input field like in a terminal.

If there are conceptual differences between programming languages, some more workmight be necessary. For instance, If you translate from a typeless JavaScript to Java,, you need to create a data class in Java for data structures that are defined inline in JS. You can also use GitHub Copilot Chat for this, especially to save a lot of typing work for larger data classes:



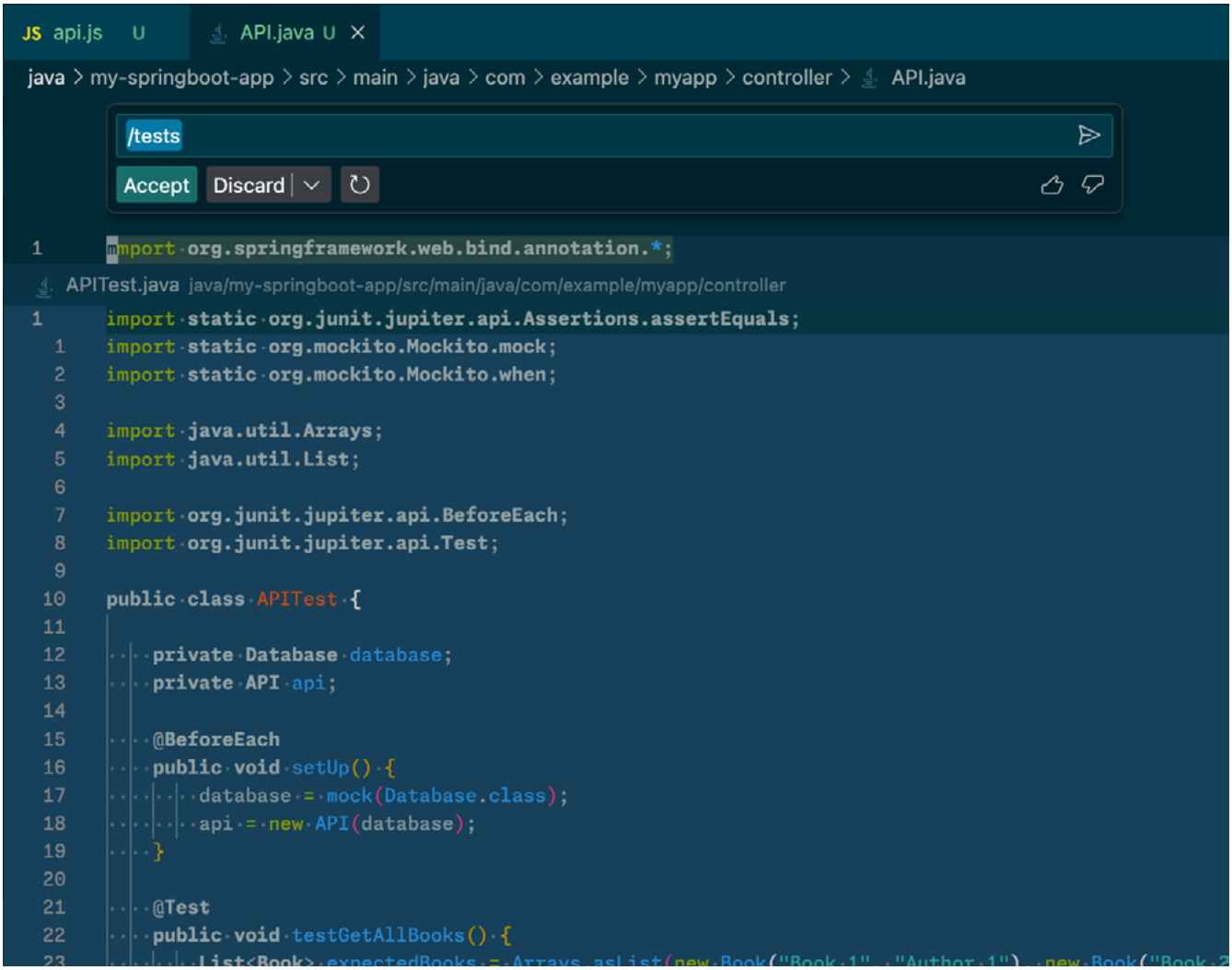
Migration Strategy – Divide & Conquer

In a migration, it is advisable to work from one compiled and functioning state to the next, committing in between to avoid context switching and being overwhelmed by too many changes and errors at once. This also keeps you focused.

Since a program often consists of a network of interdependent files, it is recommended to start from the bottom up, i.e., with files that have few or no dependencies, and then work your way "upwards." GitHub Copilot can also translate multiple files at once. In this case, however, refinement is more difficult, and you risk losing focus.

Tests

Are there unit tests in the source? Great! These can also be translated with GitHub Copilot.
Are there none? No problem. With the **/test** slash command from GitHub Copilot Chat, the problem is quickly solved:



A click on “Accept” generates the corresponding test file with some initial test cases that you can build upon. Technology-agnostic E2E or black box tests can also be an important addition to verify the general behavior of the application in both the old and new code and to ensure the core functionality on both sides.

That's it

With this approach, you can migrate applications between technologies faster than ever before using GitHub Copilot Chat. The code suggested by GitHub Copilot may not always be perfect at first, but with a little work and the right prompts on the target files , you can quickly make significant progress. GitHub Copilot Chat not only helps you better understand the source code but also saves you a lot of typing work without you having to think about it.

That in turn reduces cognitive load during the migration, allowing you to focus on the important parts of the project, such as ensuring the actual use cases and functionality of the application as well as the quality of the new code.

#Developer #AppInnovation

Daniel Meixner

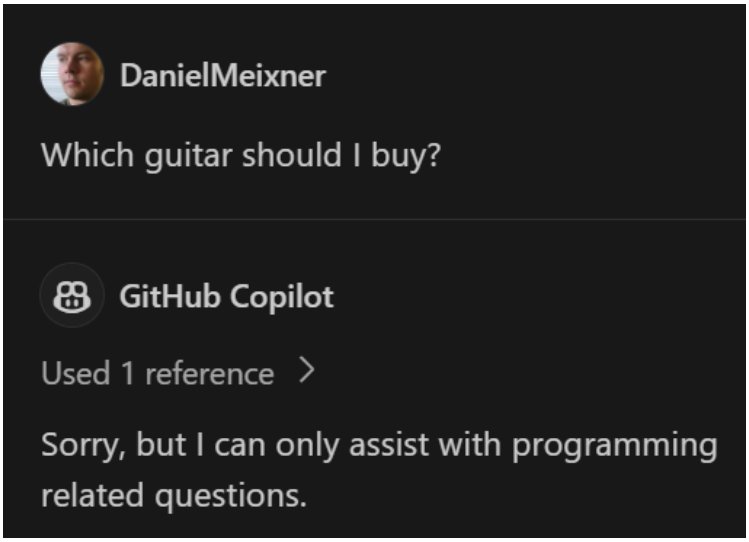
Daniel is an App Innovation Specialist at Microsoft, focusing on cloud architectures, development tools, as well as AI and intelligent apps. In various roles – from developers to evangelists – he has experienced the "good, bad, and ugly" of software development from a multitude of perspectives.

Tip 3:

No more Lorem Ipsum

by Daniel Meixner

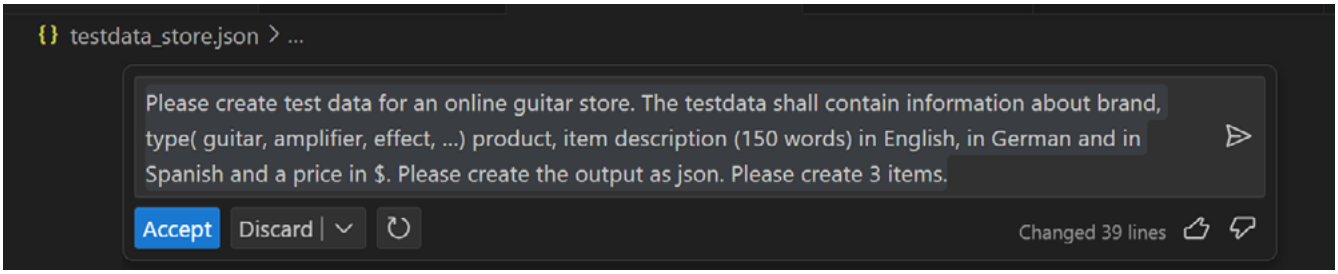
GitHub Copilot is a development tool—so it makes sense that the AI assistant focuses primarily on answers related to development. For example, if you ask GitHub Copilot for a buying recommendation for a specific product, the assistant will reply that it cannot answer that question.



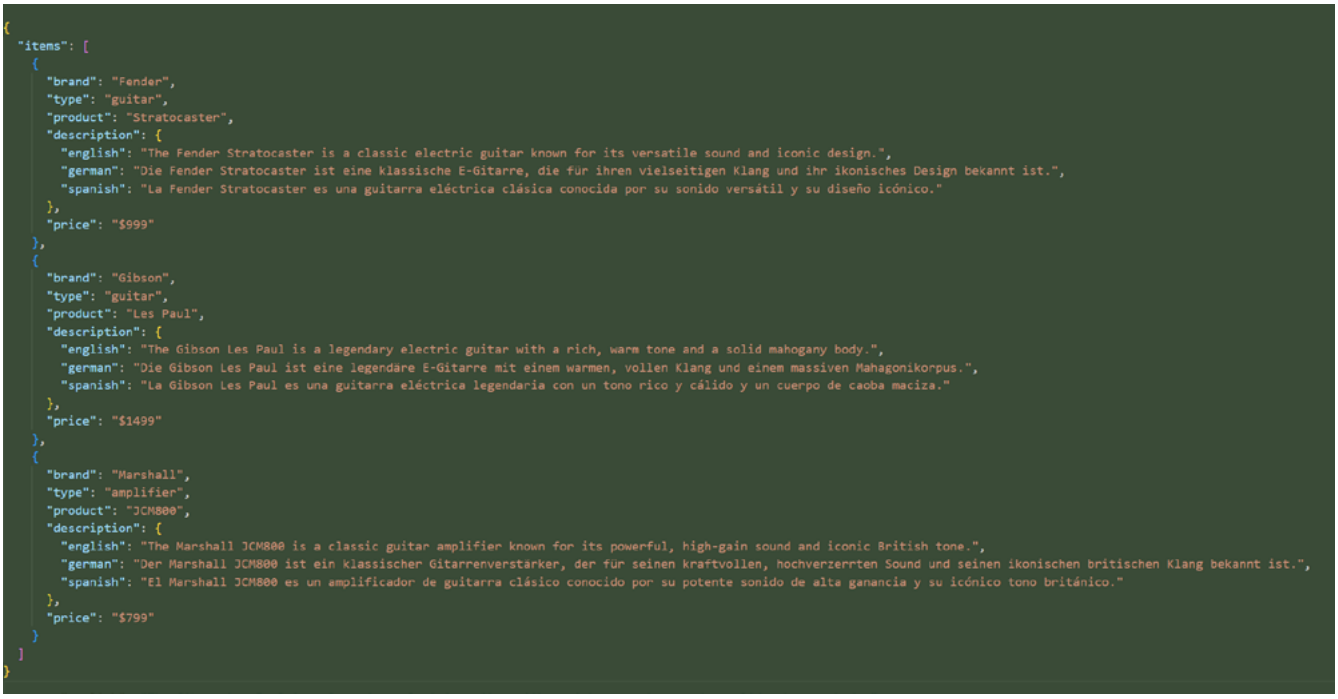
However, this isn't because GitHub Copilot can't also utilize general knowledge from domains outside of software development. You can use this knowledge, for example, to generate meaningful test data that technically fits an application and requires knowledge from a specific domain.

If you are programming a web shop, for example, you need test data to display the products. Of course, you can compile the test data from various sources. Often, this approach also yields useful data. Typically, however, this method is associated with some effort and is time-consuming. With

GitHub Copilot, you can simply have this data generated by the AI, optionally as JSON, so that you can process it sensibly. In this example, you create a file named *testdata.json* and request test data for your application inline in this file, for example with the Ctrl + I keyboard shortcut. In the following, we assume a web shop that sells guitars and accessories:



As you can see from the output, it is already quite realistic. GitHub Copilot seems to have very good knowledge of this area. The descriptions are genuine and also suitable for testing. This is much better than using Lorem Ipsum as meaningless filler text. By using multiple language variants, the application can also be developed in multiple languages. This allows you to start programming very quickly without dependencies on third parties and without losing time.



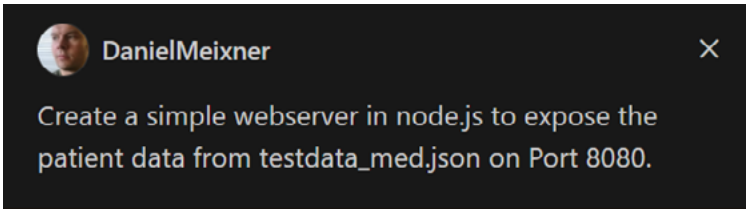
GitHub Copilot's knowledge is not limited to one area but can also be used for other use cases. Suppose you are working on a medical application and need test data for patients. GitHub Copilot can help you create fictitious patient data that you can use in your application:

```
Please create testdata for my medical application. I need
information on patients. Please provide  json containing
patientname, patientid (following the scheme of XX-1111XX11), dob,
date of last surgery, modality (like CT or MR etc), sex, hospital
and phsyscian and patient address (an address in Germany). Please
create 5 records.
```

Here is an excerpt of the result:


```
{
  "patients": [
    {
      "patientname": "John Doe",
      "patientid": "AB-1234CD56",
      "dob": "1980-05-15",
      "lastsurgerydate": "2021-10-10",
      "modality": "CT",
      "sex": "Male",
      "hospital": "ABC Hospital",
      "physician": "Dr. Smith",
      "address": "123 Main Street, Berlin, Germany"
    },
    {
      "patientname": "Jane Smith",
      "patientid": "EF-5678GH90",
      "dob": "1992-09-20",
      "lastsurgerydate": "2021-09-05",
      "modality": "MR",
      "sex": "Female",
      "hospital": "XYZ Hospital",
      "physician": "Dr. Johnson",
      "address": "456 Elm Street, Munich, Germany"
    },
```

It is also helpful here to specify formats in strings (patient ID)—without any regular expressions. Also, note how GitHub Copilot correctly interpreted "dob" as Date of Birth automatically. If you want to deliver the data, you can instruct GitHub Copilot to create a web server. This allows you to query the data via HTTP request.



#Developer #MVP

George Kosmidis

George is a Microsoft MVP and Senior Architect at Slalom Germany, specialising in Microsoft technologies. As founder and organiser of the Munich .NET Meetup, he has built a strong community of around 1,700 .NET and Azure professionals. His contributions to Azure, .NET and Azure DevOps have earned him the Microsoft MVP award. George is a passionate speaker who frequently shares his knowledge and experience with others, making a huge impact on the tech community.

Tip 4:

The GitHub Copilot Intelligent Coding

by George Kosmidis

Software development has dramatically evolved in recent times, with technological advances and changing market needs shifting from traditional, manual coding to a more agile and innovative approach. This change has been fueled by the integration of artificial intelligence (AI) and machine learning (ML), revolutionizing both developer workflows and outcomes. GitHub Copilot exemplifies this shift, offering AI-powered assistance in code writing, reviewing, and optimizing, thus fostering creativity and strategic thinking over routine coding.

GitHub Copilot, developed by GitHub, OpenAI and Microsoft, is a pioneering AI code completion tool that significantly enhances the coding experience. It provides intelligent, context-aware suggestions from an extensive coding database. It also greatly speeds up the coding process while also improving accuracy and reducing errors, proving invaluable in business contexts where time is crucial. In addition to these features, GitHub Copilot includes an interactive feature known as GitHub Copilot Chat. This allows developers to ask coding-related questions and receive answers directly within supported Integrated Development Environments (IDEs) like Visual Studio Code. By facilitating this direct dialogue, Copilot Chat further enriches the coding environment, allowing for quicker problem-solving and knowledge acquisition. The tool also encourages the exploration of new coding methodologies, leading to innovative solutions and freeing up more time for creative and strategic thinking. As such, GitHub Copilot is more than just a coding tool; it's an essential asset for developers, enhancing both

productivity and the overall quality of their work.

In this evolving landscape, Slalom has been an early adopter of GitHub Copilot, gaining valuable experience with the tool. The following section will explore their insights and discuss the impact of Copilot on their development practices. Slalom's experience highlights Copilot's role as more than just a coding aid, but as a crucial asset in enhancing productivity and the quality of development work.

The GitHub Copilot Edge in a Slalom Customer Success Story

Slalom, renowned for its innovative solutions, partnered with a leading healthcare provider looking to leverage Azure AI services to create an advanced, Azure Native healthcare analytics platform. The goal was to integrate various Azure AI capabilities, including Azure Machine Learning, Azure Cognitive Services, and Azure Bot Service, to enhance patient care and operational efficiency.

The Challenge

The project's ambition lay in seamlessly integrating native features of Azure AI services with Azure ML models to be deployed as containers or directly consumed by the AI services. This integration presented a multifaceted challenge: it demanded profound expertise to fully leverage Azure's extensive capabilities, while simultaneously necessitating efficient cost management to maximize output. Orchestrating this balance – especially under stringent timelines – posed a significant test of Slalom’s technical acumen and resource management skills.

Solution with GitHub Copilot

GitHub Copilot was introduced into the development workflow to assist the team. Here's how it made a difference:

1. **Accelerating Development**

   While building the core analytics engine using Azure Machine Learning, the team utilized GitHub Copilot to suggest optimized code patterns and algorithms. This not only accelerated development but also ensured that the solutions were robust and efficient.

2. **Streamlining Integration**

   For the integration of Azure Cognitive Services, particularly in customizing AI models for natural language processing, GitHub Copilot offered real-time coding assistance. It provided code snippets and integration examples that significantly simplified the implementation process.

3. **Enhancing Bot Development**

   In developing a patient interaction bot with Azure Bot Service, Copilot's suggestions helped the team navigate through complex conversation flows, integrating them seamlessly with the Azure backend.

A deeper dive

To better understand the impact of GitHub Copilot in this project, let's delve into a specific example. One of the key efforts was integrating Azure Cognitive Services for natural language processing within an Azure function. GitHub Copilot played a crucial role here, not only in speeding up the coding process but also ensuring that the code adhered to best practices. The following Python snippet demonstrates how Copilot guided the development team in implementing sentiment analysis using Azure's Text Analytics client.

```
# Importing necessary Azure libraries
from azure.ai.textanalytics import TextAnalyticsClient
from azure.core.credentials import AzureKeyCredential

# Initializing the Text Analytics Client
# Github Copilot suggested the use of AzureKeyCredential
client = TextAnalyticsClient(endpoint="<endpoint>", credential=
AzureKeyCredential("<key>"))

# Analyze patient feedback using Azure Cognitive Services
# GitHub Copilot helped in crafting this function structure
def process_patient_feedback(feedback):

    # Using the sentiment analysis feature of Text Analytics
    # Copilot suggested the use of 'client.analyze_sentiment' for
processing natural language
    response = client.analyze_sentiment(documents=[feedback])
    if not response:
        return { "error": "No response from sentiment analysis!" }
    if response[0].is_error:
       return { "error": "Error in sentiment analysis!" }
    sentiments = response[0]

    # Additional insights such as key phrases can also be extracted
```

```
    # This suggestion by Copilot enhances the depth of analysis
    response = client.extract_key_phrases(documents=[feedback])
    if not response:
        return { "error": "No response from key phrases
extraction!" }
    if response[0].is_error:
        return { "error": "Error in key phrases extraction!" }

    return {"sentiment": sentiments, "key_phrases": response[0].
key_phrases}
```

In the above snippet, GitHub Copilot's contribution is evident in several aspects. It suggested using AzureKeyCredential for secure API key management, a best practice in Azure application development. Additionally, the structure and implementation of the sentiment analysis function were streamlined using Copilot's suggestions, from choosing the right methods to process natural language to enhancing the analysis with key phrase extraction. This example highlights how Copilot not only accelerates coding but also enriches it with advanced features and best practices, proving invaluable in complex Azure Native solution development.

The GitHub Copilot Advantage

The recent internal assessment at Slalom presents a compelling narrative about the transformative impact of GitHub Copilot in the realm of software development. Conducted across two distinct locations – London and Denver – with a total of four development pods, this study offered a unique insight into the efficacy of GitHub Copilot under controlled conditions.

Each pod, starting with identical code bases and project backlogs, embarked on a sprint with a crucial variable: half of the teams utilized GitHub Copilot, while the other half proceeded without this AI-driven tool. This approach provided a candid examination of how GitHub Copilot influences development, especially as all developers were relatively new to both the codebase and Copilot itself.

The results were nothing short of remarkable. Less experienced engineers using GitHub Copilot saw their productivity skyrocket by 88%. This statistic alone speaks volumes about Copilot's ability to level the playing field, offering less seasoned programmers a significant boost in efficiency and confidence.

Furthermore, GitHub Copilot's role in streamlining documentation was evident. Its context-aware suggestions, coupled with on-point code examples and grammar corrections, made the documentation process not just faster but also more intuitive. Developers could focus more on the creative aspects of coding rather than getting bogged down by the intricacies of documentation.

One of the most striking outcomes was that teams using Copilot observed up to a 50% increase in code production speed. This efficiency gain is a game-changer in a field where time is often the most critical resource. When it came to repetitive tasks, Copilot's impact was even more pronounced, with a staggering 96% improvement in speed, underscoring its ability to automate and optimize the more mundane aspects of coding.

Lastly, the 77% decrease in time spent searching for information or solutions highlights another key benefit of GitHub Copilot: its role as an on-demand knowledge repository. By providing instant access to information and coding insights, Copilot significantly reduces the time developers spend in the search phase of problem-solving.

Slalom's internal assessment unequivocally demonstrates that GitHub Copilot is not just a tool but a paradigm shift in software development. Its impact on productivity, learning curve, documentation quality, and overall coding efficiency marks a significant leap forward, especially for teams striving to stay agile and innovative in the fast-paced world of technology. As we continue to explore and embrace the capabilities of AI-assisted tools like GitHub Copilot, the future of software development looks more promising and exciting than ever.

#SolutionArchitecture #Demo
#POC #AzureCLI

Julia Kordick

Titles that have been given to Julia: Senior Software Engineer, IT Team Lead, Cloud Solution Architect, Technical Specialist. Things Julia likes: Clean code, distributed software architectures, solving complex problems, JavaScript, hackathons, feminism & American football.

Tip 5:

From idea to deployment in under 30 minutes

by Julia Kordick

"Hey Julia, can you show the technical contacts from the customer how to implement this on Azure?"

„Hi Julia, do you know if what the customer has come up with will work?"

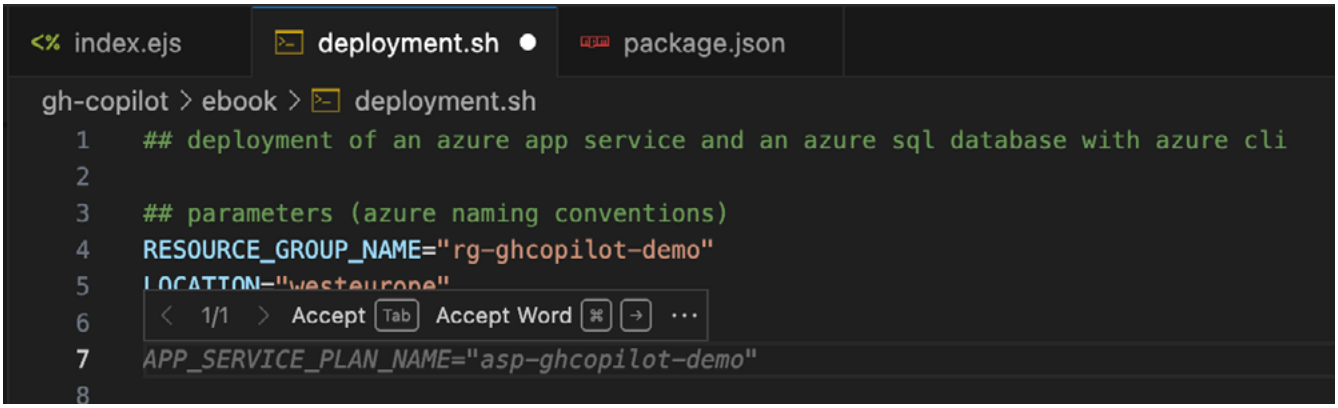
„Hi Julia, can you do a demo on a technical topic for the customer?"

I receive requests like these almost every day. Proofs of concept, explaining and demonstrating technical contexts, and demos dominate my workday. After some time, I had created a collection of various repositories that I can now use as a basis for different scenarios. In practice, every customer's topic is different – both from technical aspects and in terms of the specific use case. GitHub Copilot helps me to both adjust existing code and deployments and to quickly and effectively create new ones targeted at specific audiences.

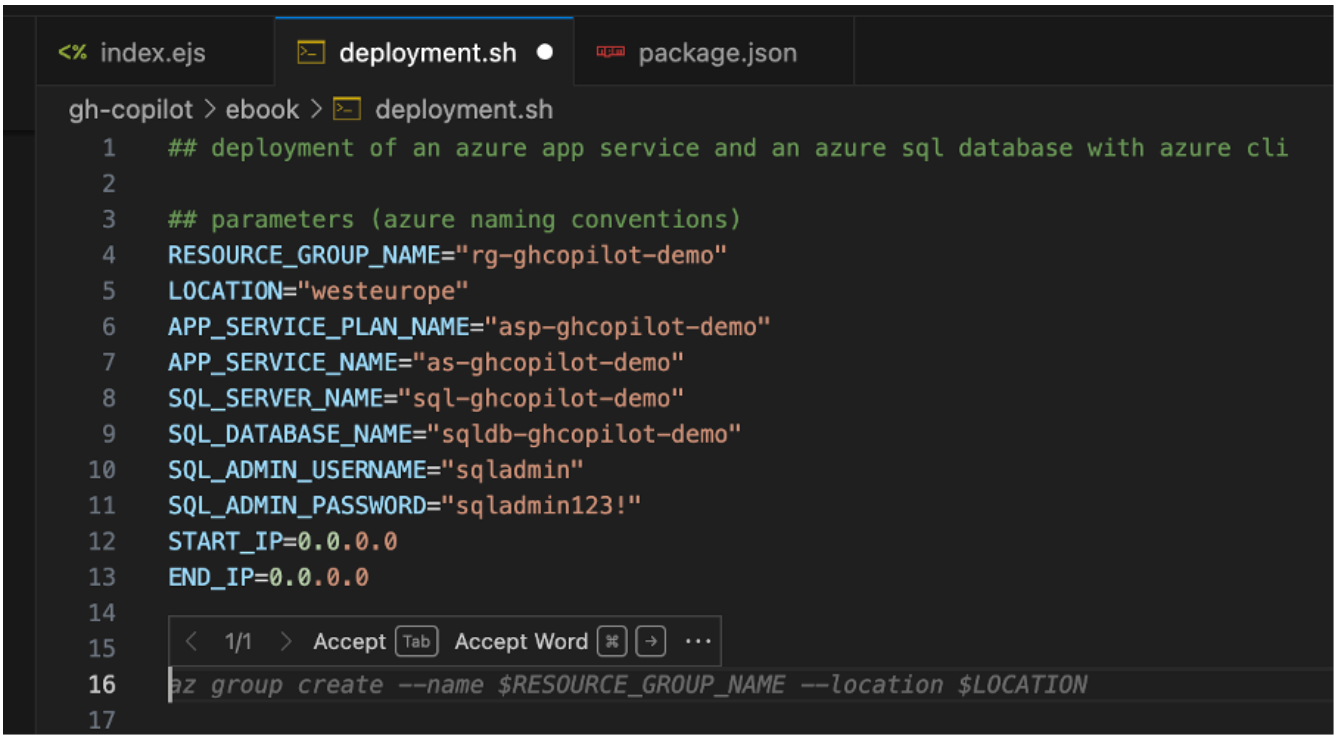
I am completely unbound in terms of Infrastructure-as-Code and programming languages – even though I have never worked professionally with .NET, my pro-code skills from other languages combined with the superpower of GitHub Copilot are completely sufficient to create runnable code in almost any language – and then deploy it quickly and easily via Azure CLI, which I want to focus on in this example.

Here's the scenario: The customer has a monolithic Node.js application that is not containerized and requires a (relational) database in the background. Therefore, we want to provision an Azure App Service and an Azure SQL database – the simplest way is through the Azure CLI.

First, I create a deployment.sh, briefly describe in the first comment what is supposed to happen, and then start setting up some parameters for my deployment. After I have defined the first parameters, GitHub Copilot knows my naming scheme and makes further suitable suggestions, which I can almost all simply accept.



Then, using the Azure CLI, I begin to provision the resources and GitHub Copilot makes suggestions that incorporate my parameters.



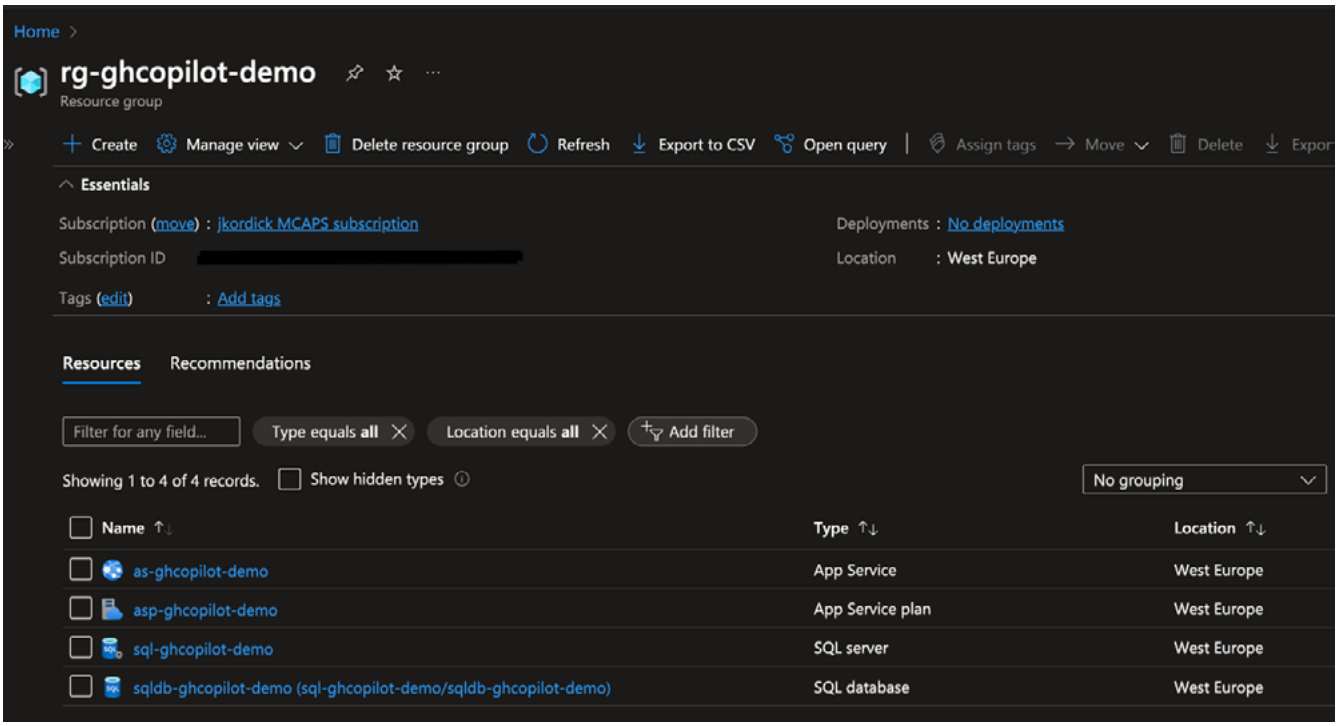
Step by step, GitHub Copilot suggests the necessary descriptive steps, including comments, to create an Azure App Service and an Azure SQL database. I just keep pressing *Enter* to receive a new suggestion from GitHub Copilot and *Tab* to accept it completely. For particularly important or security-relevant elements, like configuring firewall rules, I choose to not accept it completely but word by word using *Command & Tab* to review the suggestion from element to element carefully.

```
## configure sql server firewall
```

1/1 Accept Tab Accept Word ⌘ → ···

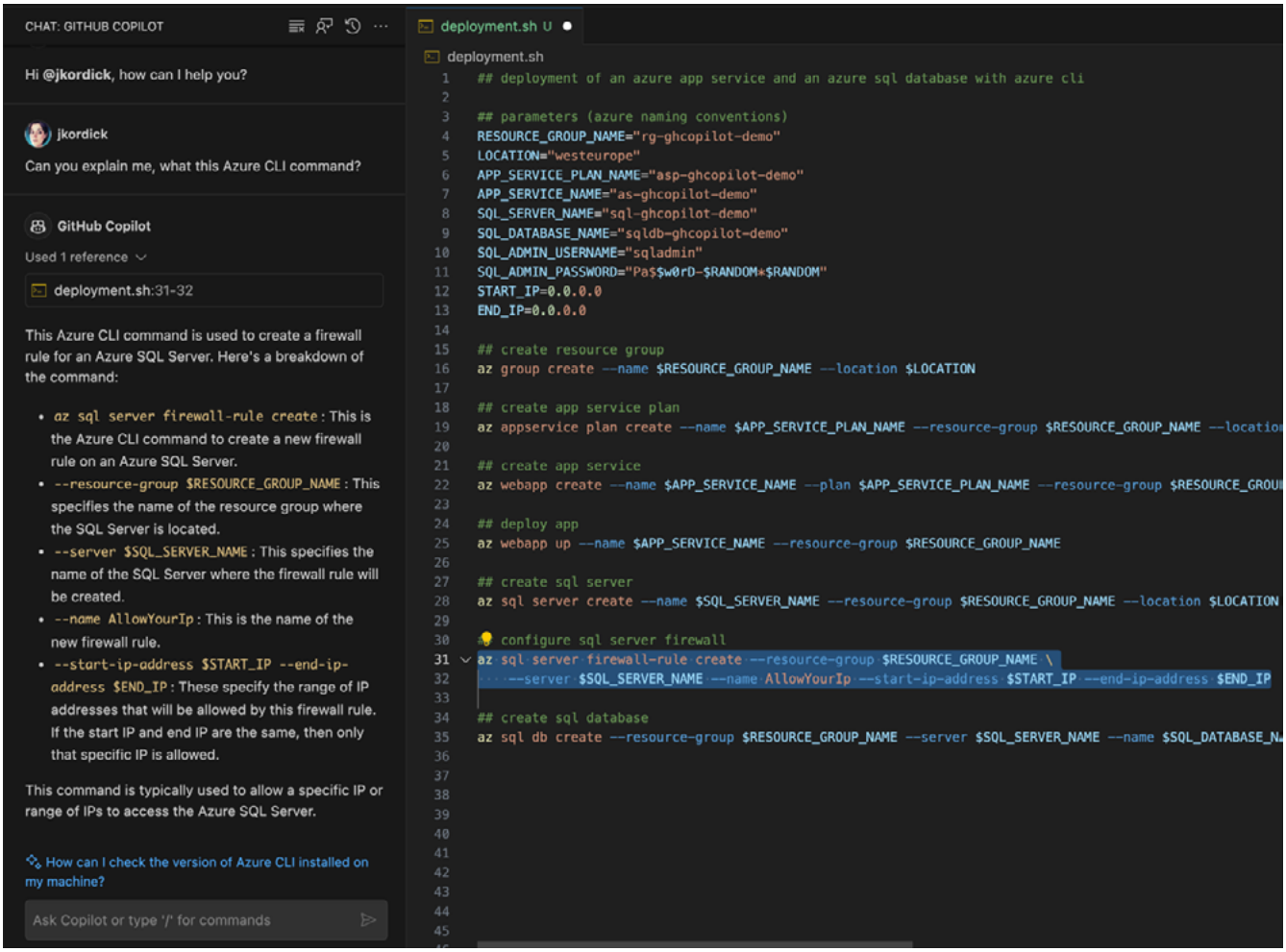
```
esource-group $RESOURCE_GROUP_NAME --server $SQL_SERVER_NAME \
  --name AllowYourIp --start-ip-address $START_IP --end-ip-address $END_IP
```

Describing the necessary resources only takes a few minutes for someone as familiar with Azure as I am. Afterward, I start the shell script I just created and can see the result a few minutes later both as an output on my console and in the Azure portal.



Despite my advanced knowledge of Azure, I naturally don't know every command of the Azure CLI by heart. Therefore, it often happens that GitHub Copilot makes suggestions that I have a rough idea of what will happen, but I want to know exactly.

That's where GitHub Copilot Chat helps me. I highlight the part in the file on the right side that I want explained and ask about it in the chat window on the left side.


CHAT: GITHUB COPILOT
Hi @jkordick, how can I help you?
jkordick
Can you explain me, what this Azure CLI command?
GitHub Copilot
Used 1 reference
deployment.sh:31-32
This Azure CLI command is used to create a firewall rule for an Azure SQL Server. Here's a breakdown of the command:
az sql server firewall-rule create : This is the Azure CLI command to create a new firewall rule on an Azure SQL Server.
--resource-group $RESOURCE_GROUP_NAME : This specifies the name of the resource group where the SQL Server is located.
--server $SQL_SERVER_NAME : This specifies the name of the SQL Server where the firewall rule will be created.
--name AllowYourIp : This is the name of the new firewall rule.
--start-ip-address $START_IP --end-ip-address $END_IP : These specify the range of IP addresses that will be allowed by this firewall rule. If the start IP and end IP are the same, then only that specific IP is allowed.
This command is typically used to allow a specific IP or range of IPs to access the Azure SQL Server.
How can I check the version of Azure CLI installed on my machine?
Ask Copilot or type '/' for commands
deployment.sh U
deployment.sh
deployment of an azure app service and an azure sql database with azure cli
parameters (azure naming conventions)
RESOURCE_GROUP_NAME="rg-ghcopilot-demo"
LOCATION="westeurope"
APP_SERVICE_PLAN_NAME="asp-ghcopilot-demo"
APP_SERVICE_NAME="as-ghcopilot-demo"
SQL_SERVER_NAME="sql-ghcopilot-demo"
SQL_DATABASE_NAME="sqldb-ghcopilot-demo"
SQL_ADMIN_USERNAME="sqladmin"
SQL_ADMIN_PASSWORD="Pa$$w0rD-$RANDOM*$RANDOM"
START_IP=0.0.0.0
END_IP=0.0.0.0
create resource group
az group create --name $RESOURCE_GROUP_NAME --location $LOCATION
create app service plan
az appservice plan create --name $APP_SERVICE_PLAN_NAME --resource-group $RESOURCE_GROUP_NAME --locatio
create app service
az webapp create --name $APP_SERVICE_NAME --plan $APP_SERVICE_PLAN_NAME --resource-group $RESOURCE_GROU
deploy app
az webapp up --name $APP_SERVICE_NAME --resource-group $RESOURCE_GROUP_NAME
create sql server
az sql server create --name $SQL_SERVER_NAME --resource-group $RESOURCE_GROUP_NAME --location $LOCATION
configure sql server firewall
az sql server firewall-rule create --resource-group $RESOURCE_GROUP_NAME \
--server $SQL_SERVER_NAME --name AllowYourIp --start-ip-address $START_IP --end-ip-address $END_IP
create sql database
az sql db create --resource-group $RESOURCE_GROUP_NAME --server $SQL_SERVER_NAME --name $SQL_DATABASE_N

#Developer #UnitTests #TypeScript

Thomas Pentenrieder

Thomas (@th_p) is a Senior Software Engineer at Medialesson, working primarily in the field of cloud and web development. He is also co-organizer of the Azure Dev Meetup in Munich. In 2023 he was honored by Microsoft as MVP in the Azure area.

Tip 6:

Writing tests with GitHub Copilot Chat

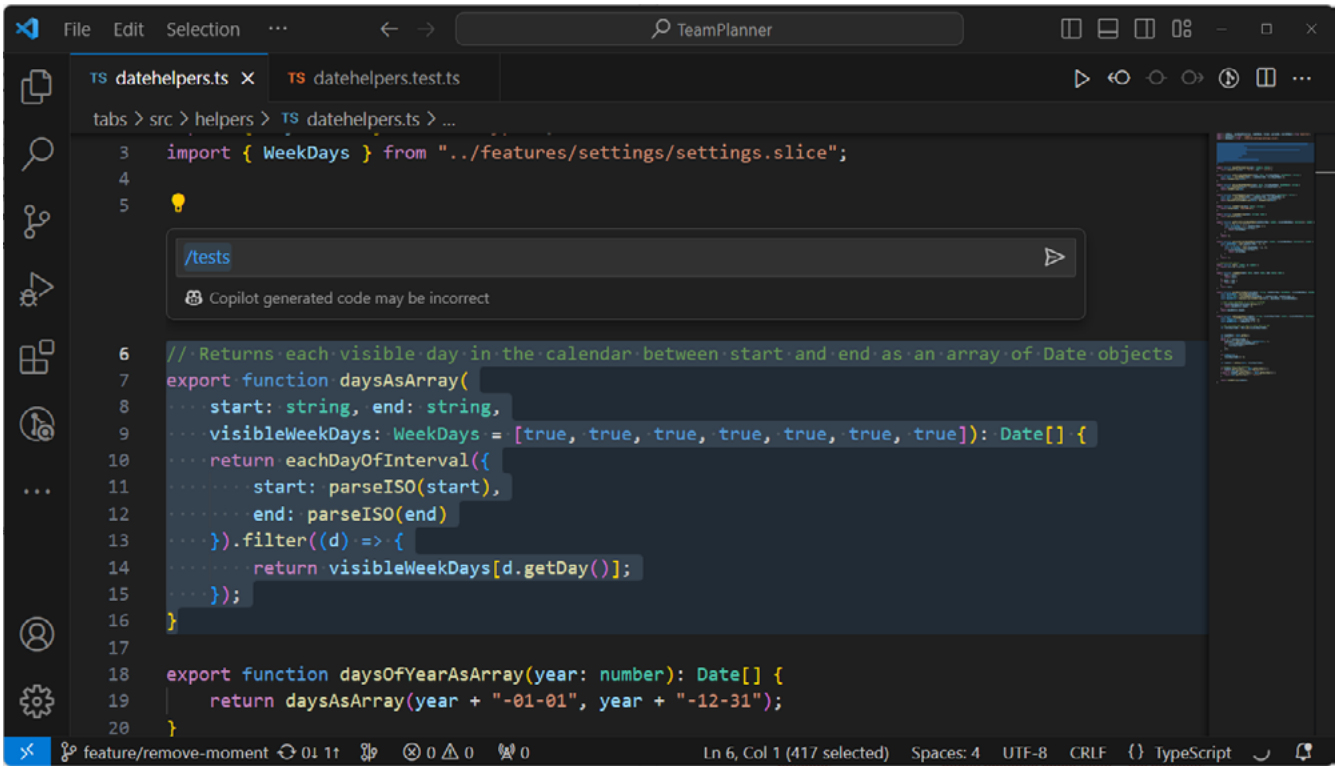
by Thomas Pentenrieder

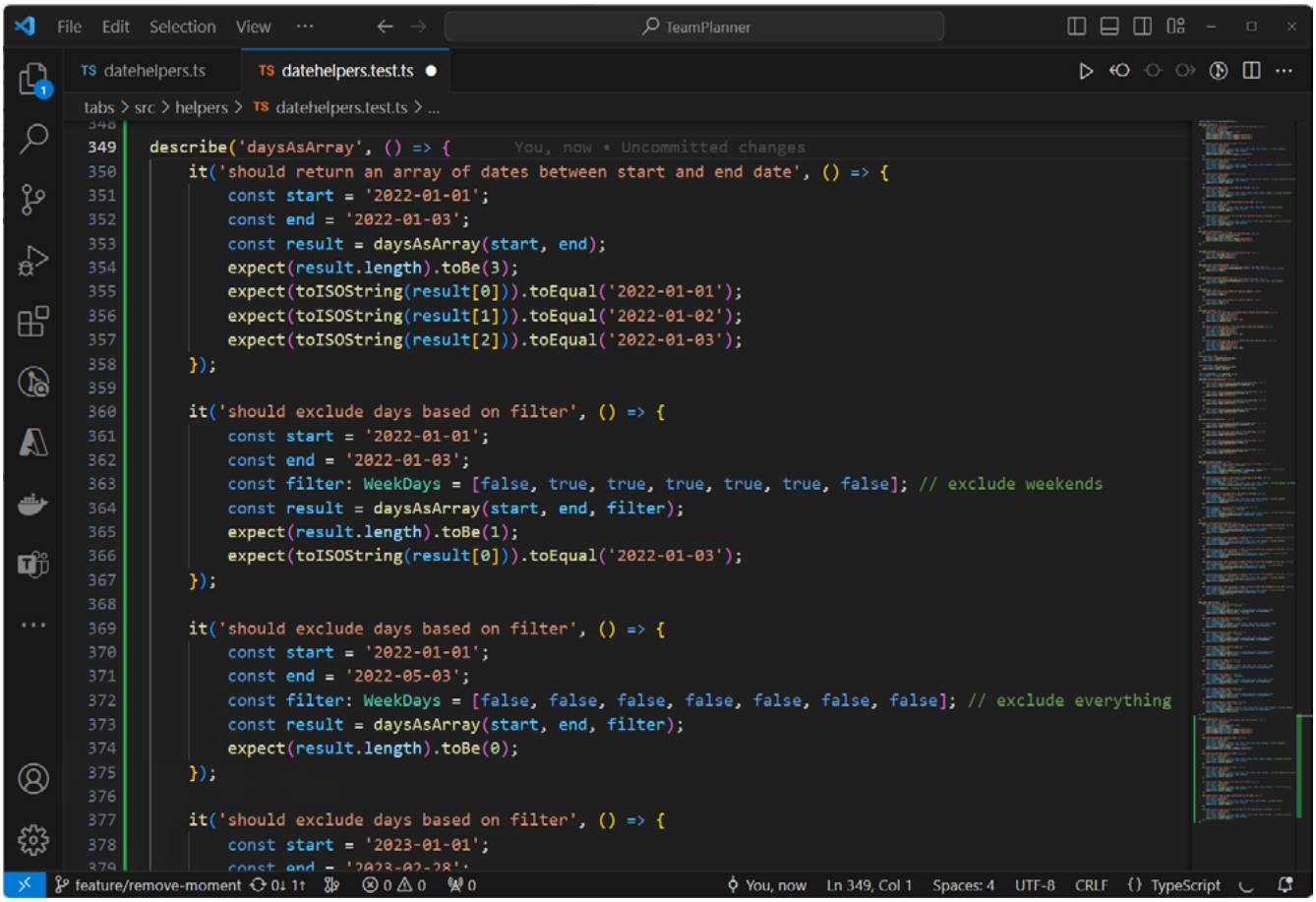
GitHub Copilot can generate large parts of program logic from comments or prompts as needed. However, from my perspective, the tool provides the greatest benefit to developers when it simplifies more tedious (though important) tasks, such as writing tests.

Especially when methods have already been fully implemented, GitHub Copilot can generate suitable unit tests and corresponding test data for various cases with very high accuracy from the existing context. This allows you not only to fundamentally review large parts of the program logic in a very short time but also to consider edge cases that you might not have thought of on your own.

In the following example, we use GitHub Copilot to extensively test *date*-specific helper methods. This is particularly critical in browsers, as the region and time zone depend on the users and are not under the control of the developers.

The method to be tested, *daysAsArray*, returns all visible days of a calendar, for example, only week-days between a start and end date formatted according to the ISO standard. By selecting the desired method and calling the */tests* command in the GitHub Copilot Chat, you receive seven different test cases within seconds. These are automatically added to the already existing test file, without you having to specify this additionally.





You should definitely correct the tests afterward. Errors in the implementation can lead to GitHub Copilot having different expectations for the results than you, and accordingly, it may formulate incorrect assumptions in the tests.

#DevOpsEngineer #PowerShell #EnterpriseMobility
#DeviceManagement #Analytics #Remediation

Jannik Reinhard

Jannik (@jannik_reinhard) is a Senior System Engineer and Microsoft MVP in the Enterprise Mobility category. He works as a Technical Lead in the AIOps division at the largest chemical company in the World. Outside of work, he blogs and speaks at events.

Tip 7:

Proactive troubleshooting on Windows devices

Deployment of Intune remediation scripts with GitHub Copilot

by Jannik Reinhard

Microsoft Intune is a unified endpoint management and security platform that enables companies to manage and protect their mobile devices such as smartphones, desktops, but also devices like HoloLens and many others. A key component of Intune are proactive remediation scripts. These are used to check devices for issues using a detection script, identify these issues, and depending on the return value, execute a remediation script that fixes the problem.

Scripts are mostly written in PowerShell and build upon each other. Here, GitHub Copilot is a big help. It only needs to be described that it is a remediation script and which error the script should check for.

Every day, new problems are reported by users or noticed through the logs. The goal is to move in a proactive direction and develop scripts for these errors that automatically detect and correct problems when they recur or occur in users. This not only reduces the effort in the service desk but also the number of tickets. Moreover, this approach offers users a better experience in their daily work.

Since the scripts ideally need to be created promptly and quickly, GitHub Copilot is a huge help here, saving a lot of time. Not every command, function, and line of code needs to be painstakingly researched. In many cases, it is enough to create a detailed description of what I want to achieve with the script. After that, GitHub Copilot creates the script for me.

Let's look at an example where it is checked if there are events with ID *1000* or *1001* in the Windows event log. These are application stability issues.

First, a comment is written for this example on what should be checked:

```
# Intune proactive remediation script to detect if events 1000 and 1001
are present in the last 24 hours.
```

All that needs to be done now is to press "Tab" a few times. GitHub Copilot then generates the script for me. I can focus on what the command should execute to read the event logs, or how to filter for the last 24 hours.

```powershell
# Intune proactive remediation script to detect if 1000 and 1001 events are present in the last 24 hours

# Get the current date/time
$now = Get-Date

# Get the last 24 hours
$last24hours = $now.AddHours(-24)

# Get the 1000 and 1001 events from the last 24 hours
$events = Get-WinEvent -FilterHashtable @{LogName='Application';ID=1000,1001;StartTime=$last24hours}

# If there are 1000 and 1001 events in the last 24 hours, return a non-zero exit code
if ($events.Count -gt 0) {
    exit 1
}
```

The whole thing should now be adjusted to filter only for problems with the Teams client. To do this, add a new comment to filter for Teams:

```powershell
# Intune proactive remediation script to detect if 1000 and 1001 events are present in the last 24 hours

# Get the current date/time
$now = Get-Date

# Get the last 24 hours
$last24hours = $now.AddHours(-24)

# Get the 1000 and 1001 events from the last 24 hours
$events = Get-WinEvent -FilterHashtable @{LogName='Application';ID=1000,1001;StartTime=$last24hours}

# Filter only events where Microsoft Teams crashed
$events = $events | Where-Object {$_.Message -like "*Microsoft Teams*crashed*"}

# If there are 1000 and 1001 events in the last 24 hours, return a non-zero exit code
if ($events.Count -gt 0) {
    exit 1
}
```

Here the suggestion is not quite correct. But the comment was also not precise enough. Let me correct that:

```powershell
# Intune proactive remediation script to detect if 1000 and 1001 events are present in the last 24 hours

# Get the current date/time
$now = Get-Date

# Get the last 24 hours
$last24hours = $now.AddHours(-24)

# Get the 1000 and 1001 events from the last 24 hours
$events = Get-WinEvent -FilterHashtable @{LogName='Application';ID=1000,1001;StartTime=$last24hours}

# Filter only events where Microsoft Teams crashed (Microsoft Teams)
$events = $events | Where-Object {$_.Message -like "*Microsoft Teams*"}

# If there are 1000 and 1001 events in the last 24 hours, return a non-zero exit code
if ($events.Count -gt 0) {
    Write-Host "Events found"
    exit 1
}
Write-Host "No events found"
exit 0
```

Now that the detection script is ready, a remediation script is needed that clears the Teams cache in such a case. This script is also created by GitHub Copilot. The procedure is the same. I'll write a comment about what I want to achieve. The comment reads:

Intune proactive remediation script to remediate team crashes with cleaning the team cache.

Again, it recognizes what is to be achieved and all I have to do is to confirm. Unfortunately, the result is still not satisfactory. I got the same result as with the detection script and need to rework it.

```powershell
# Intune proactive remediation script to remediate teams crashes with cleaning the teams cache

# Get the current date/time
$now = Get-Date

# Get the last 24 hours
$last24hours = $now.AddHours(-24)

# Get the 1000 and 1001 events from the last 24 hours
$events = Get-WinEvent -FilterHashtable @{LogName='Application';ID=1000,1001;StartTime=$last24hours}

# Filter only events where Microsoft Teams crashed
$crashes = $events | Where-Object {$_.Message -like "*Microsoft Teams*stopped working*"}

# If there are 1000 and 1001 events in the last 24 hours, return a non-zero exit code
```

Let's also apply a trick here. What I have done now is to point out to GitHub Copilot what it should do here. Now I can also simply "tab through" and have a perfect script.

```powershell
# Intune proactive remediation script to remediate teams crashes with cleaning the teams cache

# Cleare teams cache
Remove-Item -Path "$env:LOCALAPPDATA\Microsoft\Teams\*" -Recurse -Force -ErrorAction SilentlyContinue

# Restart teams
Stop-Process -Name Teams -Force -ErrorAction SilentlyContinue
Start-Process -FilePath "$env:LOCALAPPDATA\Microsoft\Teams\current\Teams.exe" -ErrorAction SilentlyContinue

# Wait 10 seconds
Start-Sleep -Seconds 10

# Check if teams is running
$teamsRunning = Get-Process -Name Teams -ErrorAction SilentlyContinue

# If teams is not running, return a non-zero exit code
if ($teamsRunning -eq $null) {
    exit 1
}
```

#Developer #CrossLanguage

Holger Sirtl

Holger has been working as a Cloud Solution Architect at Microsoft since 2006. He supports enterprise customers in building cloud-based application architectures. As a Quantum Ambassador, he is passionate about quantum computing.

Tip 8:

What was that code again?

by Holger Sirtl

Programming is my passion. Since my childhood, I've been fascinated by the world of codes and algorithms. It amazes me how I can solve complex problems and develop innovative solutions with just a few lines of code. As a Cloud Solution Architect at Microsoft, it's my job to bring this passion to customer and partner projects and to program tailored solutions.

In my customer projects, I often face the challenge of creating small code examples, whether to demonstrate new features or to illustrate best practices. However, I face two major challenges: Firstly, my work as a Solution Architect with frequently changing customer scenarios often leaves me little time to code myself. Secondly, my projects require that I develop in many different programming languages. This means that I must be proficient not in one, but in several languages.

How do I iterate over an array in Python? How do I create a tree structure in C# or Java? How do I output the content of an environment variable in PowerShell? How do I do it in Bash under Linux? There are many other examples, and it's nearly impossible to memorize every programming construct in every desired programming language. This is where GitHub Copilot comes into play. The tool allows me to formulate the desired code example via comments. This means that I don't have to keep every detail of the programming language in mind but can focus on the logic and the algorithm instead.

Using GitHub Copilot, it's very easy to create small demos and code examples. Basically, I just need to know how to write comments in the desired language. I then simply write the desired algorithm as a comment, in German or English, depending on the customer's preference. GitHub Copilot then

automatically translates my requirement into the corresponding code.

Here's a cheat sheet for comments in the most important scripting and programming languages for me:

Language	Single-line comments	Multi-line comments
Bash	# comment	: ' comment '
C#	// comment	/* comment */
C++	// comment	/* comment */
CSS	/* comment */	/* comment */
HTML	<!-- comment -->	<!-- comment -->
Java	// comment	/* comment */
JavaScript	// comment	/* comment */
PHP	// comment	/* comment */
PowerShell	# comment	<# comment #>
Python	# comment	""" comment """

Of course, this approach has its limits. To create larger software solutions, it is essential to know the structure and constructs of the desired programming language. But especially for smaller projects, quick solutions, and additions to existing software, GitHub Copilot is a huge help. GitHub Copilot saves time and facilitates the work since I don't have to keep every detail of the programming language in mind. Overall, I am excited about the possibilities that GitHub Copilot offers. It is a powerful tool that helps me implement my passion for programming in customer projects.

I recommend everyone try out the tool and see for themselves its efficiency and simplicity. At the end of the day, programming is not just about writing code, but about solving problems and implementing ideas. And that is exactly where GitHub Copilot helps me.

#Automation #Chat #Developer
#Transformation

Robin-Manuel Thiel

During the day, Robin-Manuel (@robinmanuelt) works as a Global Black Belt for Cloud-Native Architectures and AI applications at Microsoft. By night, he is a tinkerer and mechanic with nerdy stuff and anything that has a cable. Podcast: http://todocast.io

Tip 9:

Automating tedious development tasks with GitHub Copilot Chat

by Robin-Manuel Thiel

Similar to ChatGPT, GitHub Copilot offers a text interface through which one can communicate with the AI in natural language via so-called prompts. Additionally, single, or multiple relevant source code files or sections can be referenced to answer the question. As with most natural language instructions to an AI, the structure of the entered information (known as a prompt) plays an important role in GitHub Copilot Chat as well. The discipline of optimally preparing a prompt for the AI is known as prompt engineering. Repetitive tasks often follow a certain pattern. The clearer we communicate the instructions for processing these tasks via the prompt to the AI, the higher our chances of success.

Some aspects that help us structure a prompt so that it can be better processed and more accurately answered by the AI include:

- Include examples.
- Structure prompts with separators and separate requests for context information.
- Code blocks with Markdown instructions should be marked as such.

Example: Transforming source code to cloud infrastructure

Some time ago, in a customer project, I was tasked with creating a new table in an SQL database for each value of an extensive Enum with sentiment evaluations.

```
enum Sentiment {
  Happy = 'happy',
  Excited = 'excited',
  Content = 'content',
  Joyful = 'joyful',
  Optimistic = 'optimistic',
  Amused = 'amused',
  Pleased = 'pleased',
  Relieved = 'relieved',
  Grateful = 'grateful',
  Inspired = 'inspired',
  Proud = 'proud',
  Playful = 'playful',
  Satisfied = 'satisfied',
  Hopeful = 'hopeful',
  Serene = 'serene',
  Jubilant = 'jubilant',
  Ecstatic = 'ecstatic',
  Elated = 'elated',
  Blissful = 'blissful',
  Radiant = 'radiant',
  Merry = 'merry',
  Enthusiastic = 'enthusiastic',
  Cheerful = 'cheerful',
  Whimsical = 'whimsical',
  Eager = 'eager'
}

export default Sentiment;
```

What would have been a tedious task years ago can now be automated in seconds with GitHub Copilot Chat. So, I highlighted the entire Enum code and initiated GitHub Copilot Chat with the following first prompt:

```
Create an SQL script that creates a new table for each value of the enum. The table name should be the enum value itself. The table consists of the following columns: ID (primary key), Text, Score.
```

The result was quite sobering:

```
Here is the SQL script that creates a new table for each value of the enum.

CREATE TABLE happy (
  ID INT PRIMARY KEY,
  Text VARCHAR(255),
  Score INT
);
```

I received the SQL command that created a table according to my specifications from a single Enum value. However, the table name that began with a lowercase letter (which did not comply with our guidelines) and was just for a single value.

At this point, I could have refined the initial prompt by specifying the capitalization of the table name. Additionally, I could have instructed in the text to create only one single script with an SQL command to create a table per each Enum value.

Refining the prompts in several iterations would probably have taken as much time as manually writing the SQL script with the information already received. A much more effective method to steer the result in the desired direction with prompt engineering is the addition of concrete examples. Therefore, I used the insights from the first response to construct a second prompt enriched with examples:

````
Create an SQL script that creates a new table for each value of the enum.
The table name should be the enum value itself. The table consists of the
following columns: ID (primary key), Text, Score.

---

Example:

```
CREATE TABLE Happy (
  ID INT PRIMARY KEY,
  Text VARCHAR(255),
  Score INT
);

CREATE TABLE Excited (
  ID INT PRIMARY KEY,
  Text VARCHAR(255),
  Score INT
);
```
````

The result was impressive and perfectly complemented the example script from the prompt with the remaining values of the Enum. Even the capitalization of the table name, which I did not explicitly mention but only listed in the example, was considered by GitHub Copilot this time. With a few tricks, such as adding an example from the first response and clearly separating instruction and example with dashes, the prompt could be directed in the right direction without rephrasing.

Conclusion

The true strength of GitHub Copilot Chat lies in its ability to understand the existing code. When converting Enums into SQL commands, GitHub Copilot must understand the context of the Enums and generate code that is not only syntactically correct but also semantically meaningful. For repetitive

tasks and semantic conversions from one technology to another, GitHub Copilot Chat is a game-changer for developers, especially for tasks that require an understanding of the existing code and need to be adapted to another technology. Here, GitHub Copilot Chat plays to its strengths.

Interacting with an AI-powered, chat-based pair programmer not only saves time but also minimizes human errors and boosts the productivity of developers, allowing them to focus on the interesting, challenging, and important tasks of software development. We'll leave the boring part to AI from now on.

Christian Wenz

Christian (@chwenz) is a web pioneer, technology specialist and entrepreneur. Since 1999, he has written numerous books about web technologies and related topics, which have been translated into eleven languages. In his main profession, he advises companies on digitalization and Industry 4.0. He has been a Microsoft MVP for Developer Technologies since 2004.

Tip 10:

Better prompts for better code – tips & tricks for developers

by Christian Wenz

Product demos and conference presentations of GitHub Copilot focus on how easy it is to generate working code. The selected test scenarios are similar and well-prepared. In practice, working with GitHub Copilot isn't always as straightforward, as trivial tasks and complex requirements alternate. GitHub Copilot is not universally suitable for every use case.

In cases where it is applicable, code generation with GitHub Copilot doesn't always work as smoothly as shown under ideal conditions. The saying "garbage in, garbage out" applies here—or to put it positively: The quality of the prompt greatly influences the quality of the generated code. For this reason, the specialized discipline of "Prompt Engineering" has been established—a corresponding Wikipedia entry was created back in October 2021. Optimized prompts lead to better results. With GitHub Copilot, this involves both direct input prompts and the comments in the code that GitHub Copilot can process.

But as always: "It depends"; there is no one-size-fits-all solution for every application. However, adhering to some guidelines can contribute to a good result:

1. Stay clear and specific

If you task a development team with creating an app for a to-do list, you might not get the result you expect. For example, information about the target system, desired functionality, UX, and much more

might be missing. Precision is key here:

The more precise the requirements, the more accurately the results can match the requirements. This is all the truer for a software tool like GitHub Copilot. Specify exactly what you want and omit irrelevant details (with exceptions, see point 2).

The phrasing should also be clear and actionable, that is, you should avoid negations and provide clear instructions and requirements.

2. Context is King (or Queen)

A prompt is best started like a user story: the context is explained. The most important question is: What should the code do? A canonical example would be an application for a to-do list, so start the prompt with exactly that statement. This allows the system to consider existing solutions in this area and to define a concept for the application to be created. Then follow with a description of the functionalities to be implemented.

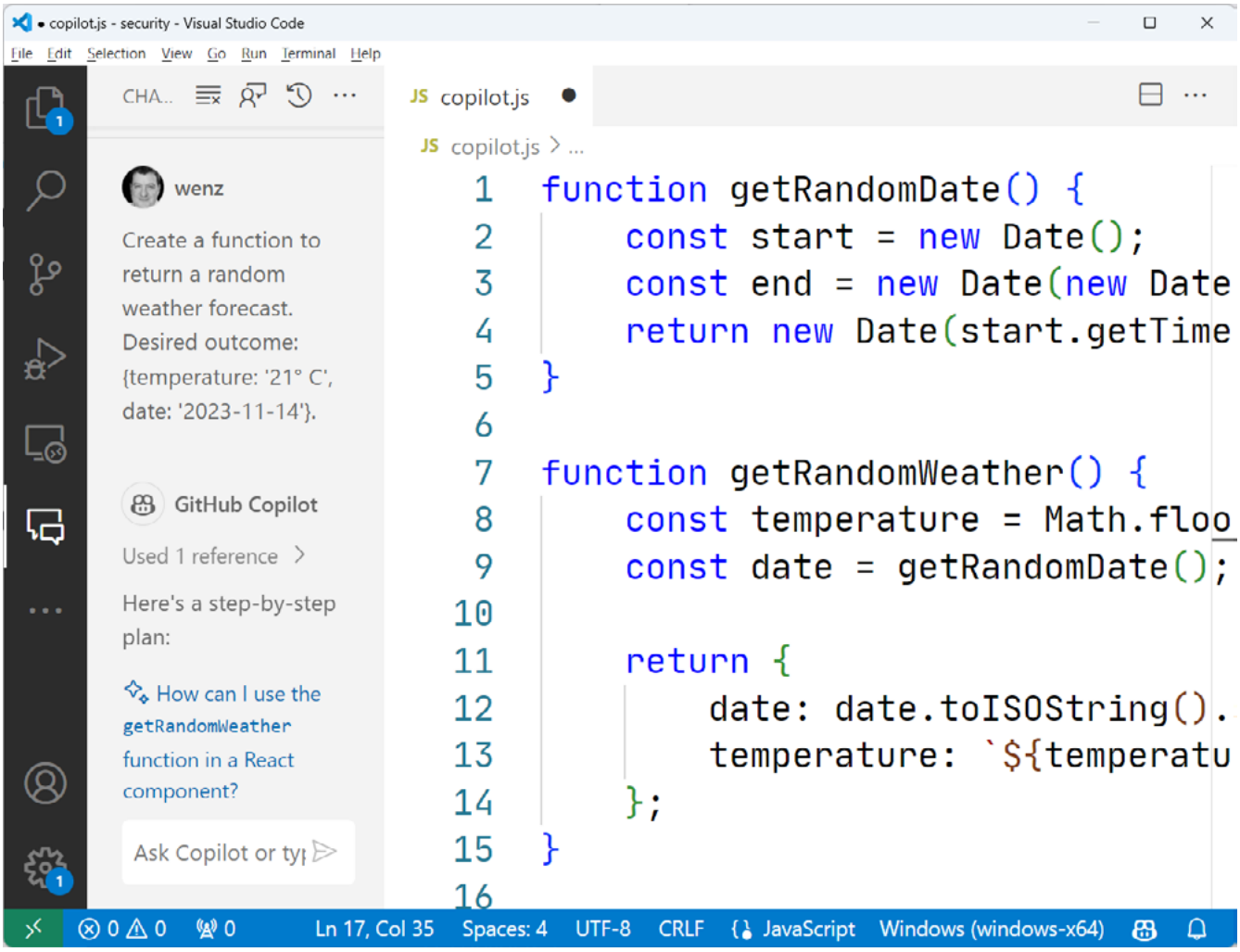
3. Step by step to success

Like programming itself, code support through GitHub Copilot is an iterative process. Rarely does the first draft already correspond to the desired and final result. There is always room for improvement, either through GitHub Copilot or through your own intervention in the code. For this reason, it makes sense to specify the individual required tasks in the code separately. Not all at once, but in successive steps. For example, when creating an API, first generate the data model and then implement the endpoints.

Of course, it's also possible to do everything at once, for example by providing the context (see point 2) and then a bulleted list of relevant features. In this case, however, interventions can only be made once the entire code has been generated. With an iterative approach, corrections can be made earlier.

4. Proceed with examples

What may seem obvious to you can raise open questions for others. In prompt engineering, it is often helpful to specify the output of a functionality with examples. This also sets the data structure as it should be processed later.



This provision of examples is important for aspects of code hygiene: naming schemas for identifiers and an existing code style can be considered by GitHub Copilot. In the end, the code will look (almost) as if it had come from you.

5. Say please and thank you

No, really! It's not about the popular cartoon in IT circles where robots take over the world but then spare a person who always said "please" and "thank you" when communicating with an AI system. However, there are apparently indications that polite interaction with a computer system both improves one's own mood and personal handling. And, let's be honest—properly applied, we save the extra typing effort for politeness phrases through the quick results generated, right?

Suad Wolgram

Suad works as a Junior Software and Cloud Developer at white duck GmbH and is studying computer science at TH-Rosenheim.

Tip 11:

Meaningful function names & variables

by Suad Wolgram

A crucial factor for the effective use of GitHub Copilot is the use of meaningful function names. Therefore, I want to show you how you can help GitHub Copilot generate more accurate code suggestions by using better function names.

Let's consider the scenario with the following code example. In the function fetch(), we want to retrieve all Star Wars characters via the Star Wars API:

```
export class ApiService {
  private starWarsUrl = environment.baseUrl;

  constructor(private httpClient: HttpClient) {}

  fetch() {

  }
}
```

In this context, it becomes clear that GitHub Copilot does not understand what goal we aim to achieve with the fetch() method. Therefore, we cannot receive suitable suggestions at this point. One solution is to replace fetch() with a meaningful function name that precisely describes what the method is supposed to do. Once the context is clearer, GitHub Copilot immediately provides suggestions for achieving our desired goal:

```
getAllStarWarsCharacters() {
  return this.httpClient.get(`${this.starWarsUrl}/people`);
}
```

In this case, GitHub Copilot even recognized our variable *starWarsUrl* and integrated it into the solution.

As a second example, let's assume we want to calculate the number of inhabitants of a planet in the Star Wars universe and return the gathered information in an Observable Map.

As before, we choose a meaningful method name and specify exactly what we want to return.
In addition, we define a variable named *planetCitizenCount:*

```
calculateCitizenCountForEachPlanet(): Observable<Map<string, number>> {
  let planetCitizenCount = new Map<string, number>();

  return this.getAllStarWarsCharacters().pipe(
    map((characters: any) => {
      characters.forEach((character: any) => {
        let planetName = character.homeworld;
        if (planetCitizenCount.has(planetName)) {
          planetCitizenCount.set(
            planetName,
            planetCitizenCount.get(planetName) + 1
          );
        } else {
          planetCitizenCount.set(planetName, 1);
        }
      });
      return planetCitizenCount;
    })
  );
}
```

As soon as we begin to insert the logic, GitHub Copilot recognizes the context due to the chosen method name and appropriate variable names and suggests the corresponding implementation.
We can also see that GitHub Copilot uses the variable correctly due to the clear context.

#CSharp #UnitTest #Developer #XUnit

Joël Zimmerli

As a Full-Stack Software Developer with experience in various programming languages such as Java, C#, and TypeScript, Joël Zimmerli focuses on test-driven development (TDD).

Tip 12:

Support in test case creation with GitHub Copilot

by Joël Zimmerli

In test-driven development (TDD), writing unit tests is one of the most important but often time-consuming tasks. In this article, I will show how I use GitHub Copilot to write my tests efficiently and comprehensively, leaving more time for actual development.

As a specific example, I use a Dotnet application where real estate properties with their sale prices are stored. The search function allows filtering by price range. To implement this filter, I wrote a factory that creates a comparison function based on the minimum and maximum prices. An excerpt of this function is shown in the following illustration:

```
static Expression<Func<HouseDocument, bool>> PriceIsInRangeFilter(
    int? minPrice, int? maxPrice){
    var expressionBuilder = PredicateBuilder.New<HouseDocument>(true);
    if (minPrice.HasValue)
        expressionBuilder.And(PriceHigherThanFilter(minPrice.Value));
    if (maxPrice.HasValue)
        expressionBuilder.And(PriceLowerThanFilter(maxPrice.Value));

    return expressionBuilder;
}
```

To use GitHub Copilot for automatic test generation, I start by creating the test name. This name describes which object is being tested, which function is called, and what result is expected. This naming structure is widely used in the world of testing. This allows GitHub Copilot to automatically generate the entire test, as shown in this illustration.

```
[Fact]
public void HouseQueryToExpression_ShouldEvaluateCorrectly(){
    // Arrange
    var query = new HouseQuery()
    {
        UpperPriceLimit = 100,
        LowerPriceLimit = 0
    };
    var house = new HouseDocument()
    {
        Id = "house",
        Name = "house",
        Price = 50
    };

    // Act
    var expression = query.ToExpression();
    var result = expression.Invoke(house);

    // Assert
    result.Should().BeTrue();
}
```

This test should then be performed with different property prices. To do this, the test can be parameterized, as shown in the following illustration.

```
[Theory]
public void HouseQueryToExpression_ShouldEvaluateCorrectly(int?
upperLimit, int? lowerLimit, int housePrice, bool expected){
    // Arrange
    var query = new HouseQuery()
    {
        UpperPriceLimit = upperLimit,
        LowerPriceLimit = lowerLimit
    };
    var house = new HouseDocument()
    {
        Id = "house",
        Name = "house",
        Price = housePrice
    };

    // Act
    var expression = query.ToExpression();
    var result = expression.Invoke(house);

    // Assert
    result.Should().Be(expected);
}
```

GitHub Copilot can also generate the test parameters, as depicted in this illustration.

```
[Theory]
[InlineData(100, 0, 50, true)]
public void HouseQueryToExpression_ShouldEvaluateCorrectly(int?
upperLimit, int? lowerLimit, int housePrice, bool expected){
```

By generating the parameters multiple times, a comprehensive list of possibilities is obtained, as shown in the next illustration. However, it is advisable to carefully review and reconsider this list.

```
[Theory]
[InlineData(100, 0, 50, true)]
[InlineData(100, 0, 99, true)]
[InlineData(100, 0, 1, true)]
[InlineData(100, 0, 150, false)]
[InlineData(100, 50, 150, false)]
[InlineData(100, 150, 200, false)]
[InlineData(100, 0, 0, false)]
[InlineData(100, 100, 100, false)]
[InlineData(100, 100, 0, false)]
[InlineData(null, null, 0, true)]
[InlineData(100, null, 0, true)]
[InlineData(100, null, 101, false)]
[InlineData(null, 100, 101, true)]
[InlineData(null, 100, 99, false)]
public void HouseQueryToExpression_ShouldEvaluateCorrectly(int?
upperLimit, int? lowerLimit, int housePrice, bool expected){
```

With these steps, GitHub Copilot can assist you in efficiently creating tests in various scenarios.

#Evaluation #Adaption

Tobias Deekens

Tobias Deekens is a retired basketball player and lousy guitarist, as well as a developer, avid teacher and spontaneous speaker with strong experience in frontend development and architecture. He feels great joy in mentoring while working with diverse teams in agile environments.

Tip 13:

How we evaluated the impact of GitHub Copilot for 3 months

by Tobias Deekens

Last February, GitHub announced its launch of GitHub Copilot for Business. This announcement immediately caught our attention and interest, and engineers across the organization shared their desire to use this product. After aligning internally on an adoption strategy, we decided to evaluate GitHub Copilot for three months to learn how it could help us be more productive. This article describes our path to evaluating and adopting GitHub Copilot.

Here at commercetools, we use various programming languages and tools to build our different products — from Scala to TypeScript and PHP to Go and Rust. We prioritize making educated choices about technology decisions so we can select tools that enhance our productivity. Furthermore, as our company grows, we want to retain our collaborative mindset, which is ingrained into our company values. We see a huge impact toward collaboration in software engineering by the breadth of tools built with generative AI and are keen to integrate them into our daily routines.

Why evaluate first and not just adopt?

You may be wondering why we evaluated an omnipresent and successful product such as GitHub Copilot for three months instead of just adopting it for all engineers. That's because, as a company policy, commercetools believes in a pragmatic approach to the adoption of AI. AI is a widely supported initiative, but its usage and determination is bottom-up. We want our teams to evaluate and decide how AI can enable productivity and functionality. In this case, the engineering department investigated

GitHub Copilot just as we would any other new tool. In doing so, we involved those who would actually use and be affected by the tool in order to get their pragmatic opinions.

Moreover, the incoming flux of new and enhanced products backed by generative AI increases the importance of thoroughly evaluating each tool's impact individually, as well as how they can be used together. For instance, is it beneficial to use Replit Ghostwriter, Codeium, CodeComplete, and GitHub Copilot in tandem, or should we consider complementing one of these with a different tool such as Mintlify or Wrap? This is a question one can only answer after exploring such tools in practice and not only through scanning their marketing websites.

To perform an informed adoption, we wanted to clearly understand the expected and actual impact of GitHub Copilot across our engineering organization. This included Frontend Engineers, Backend Engineers, Site Reliability Engineers, Test Automation Engineers and those working on documentation.

How we evaluated GitHub Copilot

Having decided to conduct a controlled evaluation, we first determined a meaningful duration for it: Three months, spanning two quarters, felt ideal. Throughout this time, we hoped to get a comprehensive snapshot of the engineering cycle, including an end of quarter where teams often roll out new functionality across our products.

After having settled on a duration, we needed a sample size. Evaluating with only five to ten out of 150 engineers could easily yield skewed results. As a result, we wanted to aim for 30 to 35 engineers to join the team of evaluators, in turn, a representation of 20 to 25%. Lastly, we wanted to involve as many disciplines as possible to form a heterogeneous group using different tools and languages.

We were now ready to share our plans through an internal blog post. We covered the process and linked a Google form for those interested in signing up. After a week, 34 people across the organization took part. This roughly matched our desired sample size, and fortunately, we didn't need to retroactively adjust our pool of evaluators. Everybody was then added to an email list and Slack channel to share updates. To grant access, all members were added to a dedicated team on GitHub, giving them access to GitHub Copilot.

With everything set up, we stepped back and allowed everybody to proceed with their work, incorporating GitHub Copilot into their workflow. Only after a week did we briefly check in to ensure that

everybody had successfully installed and integrated GitHub Copilot into their editor of choice. For the coming weeks, people shared their impressions and code samples in Slack or on Pull Requests while we remained in the background preparing a larger final survey.

Throughout the duration of our evaluation, we remained in touch with GitHub in the background. The evaluators shared interesting statistics with us, such as the average code acceptance rate. Additionally, we obtained GitHub Copilot Chat for the final two weeks of our evaluation, allowing us to peek into the future of GitHub Copilot as a more collaborative tool. We are excited to see where the future of GitHub Copilot goes and where its different offerings take us.

Results and outcome

We anticipated GitHub Copilot to integrate seamlessly into daily workflows and be easy to use. We hoped our suggestions would be useful across programming languages and not get in people's way. Throughout our evaluation, we were not disappointed in any of these expectations, but we also discovered room for improvement and noticed that the quality of suggestions varied significantly depending on the type of work being performed.

In more detail, our main survey turned out to be 15 questions long and focused around three key areas:

1. Was GitHub Copilot used continuously?
2. Does GitHub Copilot make us more productive?
3. Does GitHub Copilot pose any major risks or downsides to us?

> ***"It writes release notes for me! This is the best thing ever!"***
>
> AN EVALUATOR AFTER GETTING AN EARLY-MORNING COFFEE

Based on these three key areas, we drilled deeper with questions such as:

1. How often did you use GitHub Copilot during our trial?
2. Did your usage of GitHub Copilot change during the three months?
3. How often did you have to amend suggestions made by GitHub Copilot?
4. Where did you see your biggest productivity gains?
5. Should we evaluate other tools using generative AI this year to improve our productivity?

> ***"There was a daily wrestle between GitHub Copilot vs. regular IntelliSense."***
>
> AN EVALUATOR WATCHING AN EVERYDAY STRUGGLE

Having asked all these questions, these were the main takeaways:

- 57% used GitHub Copilot every day; everybody else used it every other day.
- 95% stated that GitHub Copilot made them more productive.
- 63% claimed that their usage increased over time.
- 67% stated that the suggestions were helpful.
- 82% stated that the suggestions were rarely problematic.
- 60% claimed that GitHub Copilot was sufficient as an AI coding assistant.
- 80% did not expect other tools to be significantly better.
- 100% would like to continue using GitHub Copilot.

> ***"At times GitHub Copilot seems asleep with many VS Code windows open. Then it yells 50 lines of code at you."***
>
> AN EVALUATOR BEING HIT BY A RAPID SUGGESTION BURST

In addition to these numbers, we also managed to gather more qualitative insights on where GitHub Copilot excelled and areas where it fell short:

- Succeeds at writing tests (72%).
- Helps in refactoring code (42%).
- Shines in autocompletion, boilerplate and scaffolding (~60%).
- Struggles with complex business logic (82%).
- Is not powerful when code context matters (43%).
- Should be considered carefully with performance or security-related topics (27%).
- Is not helpful with highly specialist or modern frameworks (14%).

> ***"GitHub Copilot is exactly smart enough to at times be dangerous too."***
>
> AN EVALUATOR AFTER GITHUBCOPILOT SUGGESTED TO LOAD 40,000 ENTITIES FROM A DATABASE ONE-BY-ONE

As we evaluated GitHub Copilot for a longer period, we also saw areas for improvement:

- Currently, you cannot provide feedback on its suggestions.
- It can't be configured to be inactive in specific folders or situations.
- It doesn't work very well across file boundaries.
- It struggles with homogeneous refactoring across larger code bases.

Those are a lot of numbers, but they certainly helped us understand the usefulness of GitHub Copilot across our organization and helped us make the decision to adopt the tool. Once enabled, GitHub Copilot was used continuously, and its use even increased. Suggestions were often accepted and of good quality. Users were able to embed it easily into their existing work environments and experienced significant productivity gains — all of which means that we will continue to roll it out across our organization in the coming weeks.

This article was originally published at [https://commercetools.com/blog/how-we-evaluated-the-impact-of-github-copilot-for-3-months.](https://commercetools.com/blog/how-we-evaluated-the-impact-of-github-copilot-for-3-months) Thanks to Tobias and commercetools for allowing us to republish in this eBook.

#Accessibility #WCAG #PromptEngineering

Dennis Gassen

Dennis Gassen is the Go to Market Manager for Digital & Application Innovation at Microsoft Germany. When he's not deeply immersed in business (whatever that may mean), he supports the Developer Community. In his free time, he is most likely to be found cycling somewhere in the south of Munich.

Tip 14:

GitHub Copilot as an AI assistant for accessible software

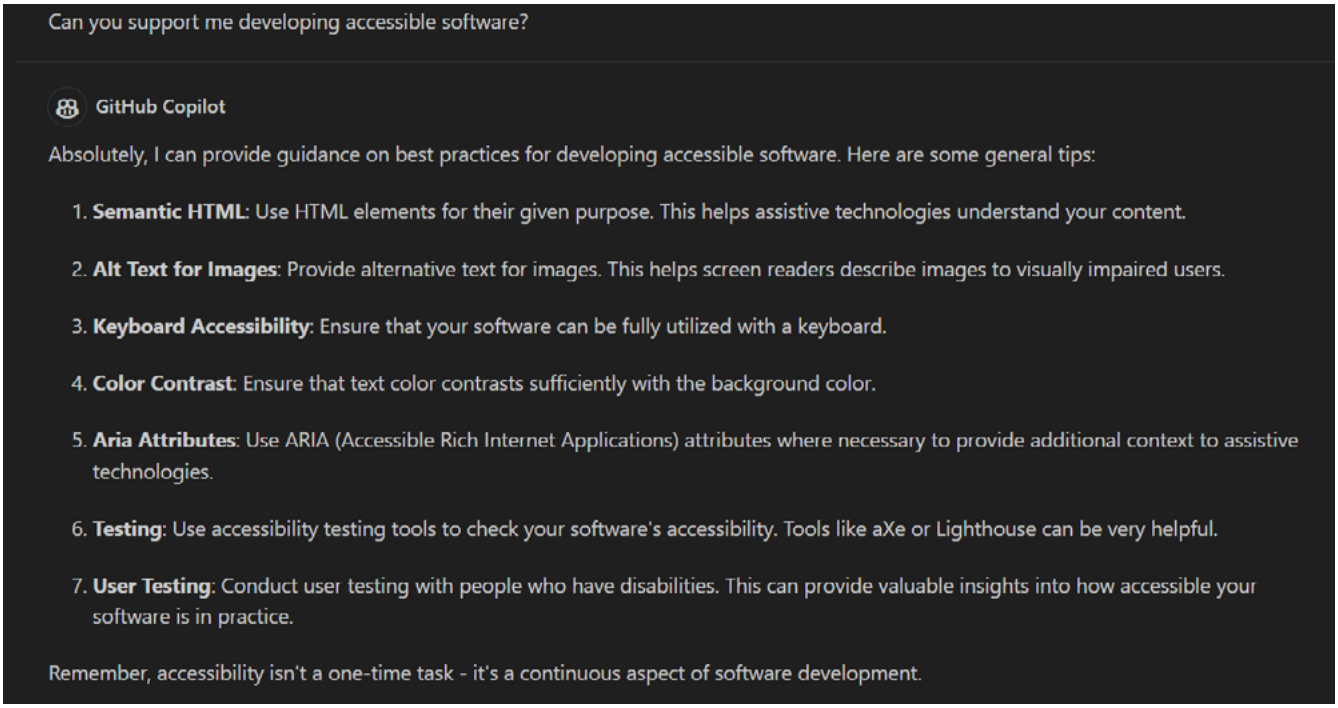
by Dennis Gassen

This article does not offer a comprehensive view of how accessible software can or should be developed – the scope here is too limited for that. Instead, I want to show that GitHub Copilot can already consider and comply with some fundamental standards that make software-based products more accessible and usable for many people with disabilities.

Approximately 1.3 billion people worldwide live with a disability, accounting for about 16 percent of the global population. A large portion of these individuals reside in low-income countries with poor health care and education systems and are poorer than many people without disabilities. For this group, access to information is critically important.

The Web Content Accessibility Guidelines (WCAG) are the international standard for accessible web offerings. In the European Union, they have been mandatory for public institutions since September 2019 for new websites, since September 23, 2020, for existing websites, and since June 23, 2021, for mobile applications. The World Wide Web Consortium (W3C) first published the WCAG as Version 1.0 in 1999. Since December 2008, Version 2.0 has been recognized as an ISO standard. Currently, Version 2.2 is recommended by the W3C. Although the W3C recommendations do not have general legal validity, they nevertheless have high binding force for the development of accessible websites and applications.

When we now use GitHub Copilot Chat and simply ask how GitHub Copilot can support us in developing accessible applications, we receive the following answer:



These initial recommendations are not incorrect, but they should definitely be understood as "general tips." Using **Prompt Engineering**, however, we have the opportunity to give GitHub Copilot more context about our specific requirements to get more detailed answers.

For this, the following prompt can be used:

```
I want to learn more about accessibility standards and need to write
code that confirms with WCAG 2.2 defined at https://www.w3.org/TR/
WCAG22/. Please act as my accessibility coach to make sure I stick to
the most common accessibility standards and guidelines. When you answer
accessibility related questions, please also use sources like w3.org
and webaim.org. Please also provide links and references in your ans-
wers whenever possible. When you suggest code, use semantic HTML and
follow the ARIA Authoring Practices Guide and related design patterns.
```

The response from GitHub Copilot then looks like this:

GitHub Copilot

Absolutely, I can guide you through the process of developing software that adheres to WCAG 2.2 standards and follows the ARIA Authoring Practices Guide. Here are some key principles and coding practices to keep in mind:

1. **Perceivable**: Information and user interface components must be presentable to users in ways they can perceive. This means that users must be able to perceive the information being presented. For example, providing alt text for images, transcripts for audio, and captions for videos.

```
<img src="image.jpg" alt="Description of image">
```

2. **Operable**: User interface components and navigation must be operable. This means that users must be able to operate the interface. For example, ensuring that all functionality is available from a keyboard.

```
<button onclick="myFunction()">Click me</button>
```

3. **Understandable**: Information and the operation of user interface must be understandable. This means that users must be able to understand the information as well as the operation of the user interface. For example, using clear labels for form inputs.

```
<label for="fname">First name:</label><br>
<input type="text" id="fname" name="fname"><br>
```

4. **Robust**: Content must be robust enough that it can be interpreted reliably by a wide variety of user agents, including assistive technologies. This means that users must be able to access the content as technologies advance. For example, using semantic HTML elements like `<nav>`, `<header>`, `<main>`, and `<footer>` to structure your content.

```
<header>
  <h1>Website Title</h1>
</header>

<nav>
  <a href="#content1">Content 1</a>
  <a href="#content2">Content 2</a>
</nav>

<main id="content1">
  <h2>Content 1</h2>
  <p>This is some content.</p>
</main>

<footer>
  <p>Copyright 2022</p>
</footer>
```

You can find more detailed information on each principle, guideline, and success criteria in the WCAG 2.2 documentation.

For more practical advice on implementing these standards, WebAIM is a great resource. They provide a wide range of articles, tutorials, and checklists to help you make your software more accessible.

The ARIA Authoring Practices Guide is a great resource for understanding how to use ARIA to improve accessibility, especially for complex widgets and interaction patterns.

Remember, accessibility is not just about adhering to standards - it's about ensuring that as many people as possible can use and benefit from your software.

The prompt serves as a basis and contains the following points:

- Extended context information to answer queries.
- Definition of the role GitHub Copilot has and what feedback we expect.
- Reference to reputable sources to be used in answering our questions.
- Reference to additional resources and sources.

We can also ask GitHub Copilot for specific examples where accessibility standards are not being met:

Please create a sample website that is not conform with the WCAG 2.2 standards and illustrate on this website what are the most common mistakes when it comes to accessibility.

GitHub Copilot

Sure, here's a simple example of a website that does not conform to WCAG 2.2 standards:

```
<!DOCTYPE html>
<html>
<head>
    <title>Non-accessible Website</title>
    <style>
        .low-contrast {
            color: #888;
            background-color: #aaa;
        }
    </style>
</head>
<body>
    <img src="logo.png">
    <div class="low-contrast">This is a low contrast text.</div>
    <a href="#">Click here</a>
    <input type="text">
    <button onclick="alert('Button clicked!')">Submit</button>
</body>
</html>
```

Here are the most common mistakes in this website:

1. **Missing alternative text**: The image tag `<img src="logo.png">` does not have an `alt` attribute. Screen readers use the `alt` attribute to describe the image to visually impaired users.
2. **Low contrast**: The text inside the `div` with class `low-contrast` has low contrast, making it hard for visually impaired users to read.
3. **Non-descriptive link text**: The link text "Click here" does not provide context about what the link does or where it goes.
4. **Missing form labels**: The input field `<input type="text">` does not have a corresponding label, making it hard for screen reader users to understand its purpose.
5. **Non-accessible JavaScript**: The button uses an `onclick` event handler, which is not accessible to keyboard-only users. It should use a more accessible event like `onkeypress`.

Remember, these are just a few examples of common accessibility mistakes. There are many other aspects of accessibility to consider when developing a website.

Even better, GitHub Copilot also immediately suggests the appropriate solutions:

Here's how you can fix these issues:

```
<!DOCTYPE html>
<html>
<head>
    <title>Accessible Website</title>
    <style>
        .high-contrast {
            color: #333;
            background-color: #fff;
        }
    </style>
</head>
<body>
    <img src="logo.png" alt="Company Logo">
    <div class="high-contrast">This is a high contrast text.</div>
    <a href="#">More information</a>
    <label for="inputField">Input Field</label>
    <input id="inputField" type="text">
    <button type="button" onclick="alert('Button clicked!')">Submit</button>
</body>
</html>
```

Of course, the examples shown here are very simple. However, as an illustration, they very effectively demonstrate the possibilities GitHub Copilot offers to adhere to Accessibility Standards, making your own software as accessible as possible to as many people as possible.

Here are some more prompts that can help:

Prompt: What are best practices to incorporate accessibility testing in the whole application lifecycle?

GitHub Copilot's answer

```
Incorporating accessibility testing throughout the application lifecyc-
le is crucial for creating and maintaining accessible software. Here
are some best practices:

 1. Integrate Accessibility from the Start: Accessibility should be a
consideration from the very beginning of the project, not an afterт-
hought. This includes during the design phase, where color contrast,
font sizes, and navigation should be considered.
 2. Automated Testing: Use automated testing tools like axe, Lighthouse,
or WAVE to catch common accessibility issues. These tools can be
```

integrated into your CI/CD pipeline to ensure accessibility is checked with every build.

3. Manual Testing: Automated testing can't catch everything. Manual testing, including using the software with a screen reader, navigating with a keyboard only, and testing with real users with disabilities, is crucial.

4. Incorporate Accessibility in Code Reviews: Make accessibility a part of your code review process. This can help catch potential issues before they become a part of the codebase.

5. Training and Awareness: Ensure that all members of your team understand the importance of accessibility and how to implement it. This includes developers, designers, content creators, and testers.

6. Regular Audits: Regularly audit your application for accessibility. This can help catch any new issues that may have been introduced.

7. Use WCAG as a Guide: The Web Content Accessibility Guidelines (WCAG) provide a wide range of recommendations for making Web content more accessible. Use these guidelines as a reference throughout your project.

Remember, accessibility is not a one-time task, but a continuous effort throughout the entire application lifecycle.

For more detailed information, you can refer to the WCAG guidelines and WebAIM, which provides a wide range of resources to help you make your software more accessible.

Prompt: How can I test if my application is accessible for a screen reader?

GitHub Copilot's answer

```
Testing your application for screen reader accessibility involves both
automated and manual testing methods. Here are some steps you can follow:

 1. Automated Testing: Use automated testing tools like axe, Lighthouse,
or WAVE to catch common accessibility issues. These tools can help
identify issues like missing alt text, incorrect semantic HTML, and
missing form labels.
 2. Manual Testing with a Screen Reader: To truly understand the expe-
rience of a screen reader user, you should test your application with a
screen reader. There are several screen readers available, inclu-
ding NVDA and JAWS for Windows, VoiceOver for macOS and iOS, and Talk-
Back for Android. As you navigate your application with the screen
reader, pay attention to whether all content and functionality is
accessible and understandable.
 3. Keyboard Navigation: Screen reader users often navigate using the
keyboard, so ensure that all interactive elements are keyboard accessi-
ble (can be focused and activated using the keyboard) and that the tab
order is logical.
 4. Semantic HTML: Screen readers rely on semantic HTML to understand
the content and structure of the webpage. Ensure that you're using the
correct HTML elements for their intended purpose and that you're using
ARIA roles and properties where appropriate.
 5. Descriptive Text: Ensure that all images have alt text, that form
inputs have associated labels, and that links and buttons have descrip-
tive text.
Remember, while automated testing can catch many issues, there's no
substitute for manual testing with a screen reader. It's also a good
idea to involve users with disabilities in your testing process, as
they can provide valuable insights based on their lived experiences.
For more detailed information, you can refer to the WCAG guidelines
```

and WebAIM, which provides a wide range of resources to help you make your software more accessible.

Prompt: What are best practices for creating accessible tables in HTML?

GitHub Copilot's answer

Creating accessible tables in HTML involves ensuring that the table structure is clear and understandable, both visually and for assistive technologies. Here are some best practices:

1. **Use <th> for Headers:** Use the <th> element for table headers, and the <td> element for table data. This helps screen readers understand the structure of the table.

```html
<table>
  <tr>
    <th>Header 1</th>
    <th>Header 2</th>
  </tr>
  <tr>
    <td>Data 1</td>
    <td>Data 2</td>
  </tr>
</table>
```

2. **Use scope Attribute:** The scope attribute can be used on <th> elements to specify whether it's a header for a column (scope="col") or a row (scope="row"). This helps screen readers understand the relationship between headers and cells.

```html
<table>
  <tr>
    <th scope="col">Header 1</th>
    <th scope="col">Header 2</th>
  </tr>
  <tr>
    <th scope="row">Row Header</th>
    <td>Data</td>
  </tr>
</table>
```

3. **Use <caption> for Table Description:** The <caption> element provides a description of the table. This can be useful for providing context to screen reader users.

```
<table>
  <caption>A description of the table</caption>
  <!-- rest of table -->
</table>
```

4. **Use <thead>, <tbody>, and <tfoot>:** These elements can be used to group the header, body, and footer of the table. This can make the table more understandable for screen reader users.

```
<table>
  <thead>
    <!-- header rows -->
  </thead>
  <tbody>
    <!-- body rows -->
  </tbody>
  <tfoot>
    <!-- footer rows -->
  </tfoot>
</table>
```

5. **Avoid Complex Layouts:** Complex table layouts can be confusing for screen reader users. If possible, stick to simple layouts with clear headers for each row and column.
6. **Ensure High Contrast:** Ensure that the text and background colors of your table have sufficient contrast to be easily readable.

For more detailed information, you can refer to the WCAG guidelines and WebAIM, which provides a wide range of resources to help you make your tables more accessible.

Summary

With Prompt Engineering, you have the opportunity to significantly improve the results of GitHub Copilot Chat regarding accessibility. Of course, no one should expect perfect results, as it is "only" a "Co-Pilot." However, you can use GitHub Copilot Chat very effectively to expand your own knowledge and understanding of accessibility. Also, always remember that you can ask GitHub Copilot further questions about the answers or ask for additional references and examples. It is always the developers' responsibility to review the suggestions.

Syrine Chelly

Syrine Chelly is a Cloud Solution Architect Developer Advocate at Microsoft where she has been working in the intelligent cloud customer success unit for over 3 years. Prior to that, she worked in the robotics automation and AI in the automotive industry.

Tobias Wittenburg

Tobias Wittenburg is a Cloud Solution Architect Developer Advocate at Microsoft. He has worked in the software industry since 2005 and has experience in large scale software development on various technologies.

Tip 15:

From requirement to component

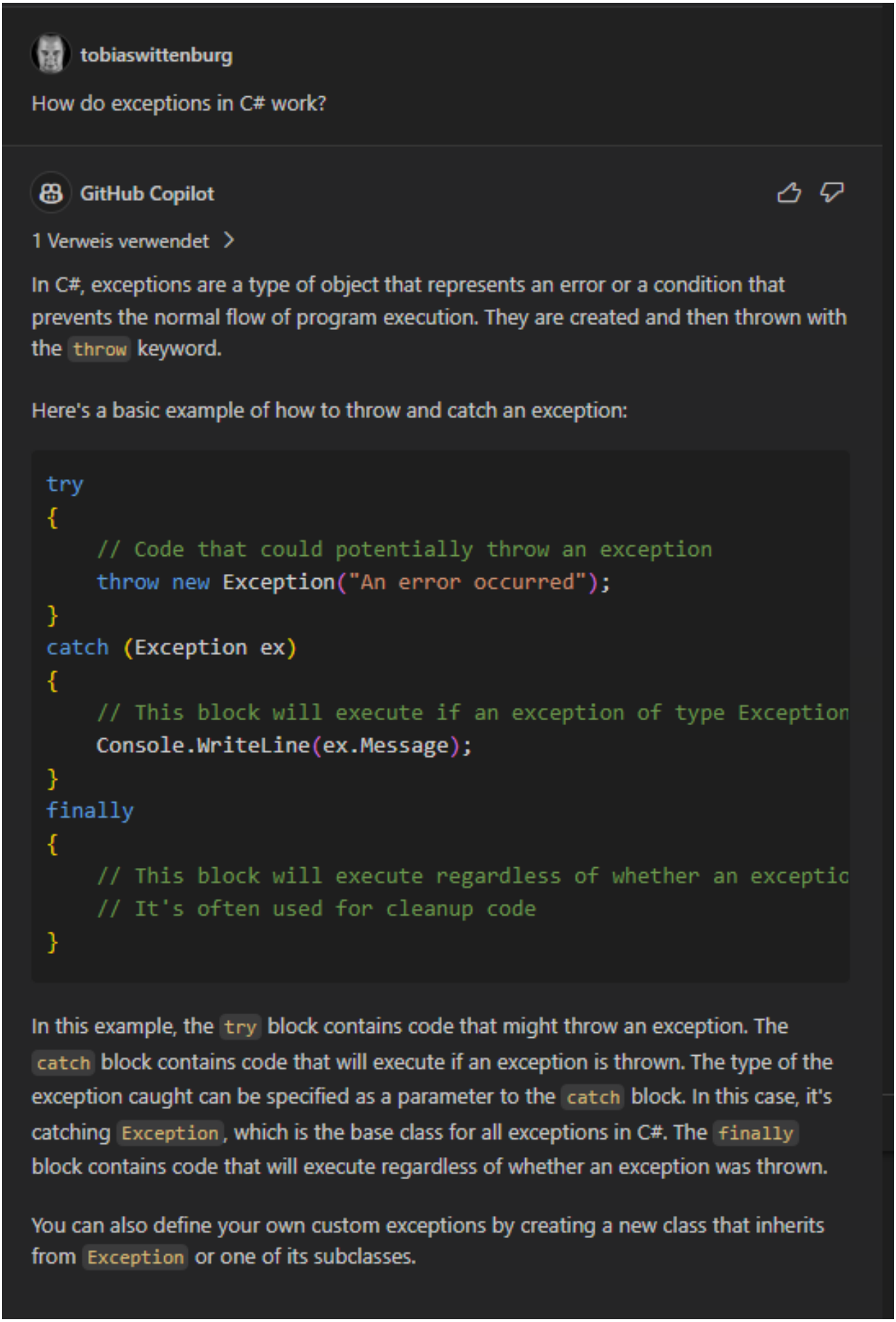
by Syrine Chelly and Tobias Wittenburg

Using natural language to produce code has been the software industry's dream for a long time, dating back to the 1950s with the introduction of COBOL. With the usage of large language models, we are coming closer and closer to this vision. GitHub Copilot leverages this capability to produce code.

When starting out with GitHub Copilot and GitHub Copilot Chat we used it to produce inline code from comments and ask questions about code, languages, procedures or frameworks in Copilot Chat. Here is an example in C# with the suggestion in grey.

```
//create a method to calculate the simple interest
public double Calculate(double principal, double rate, double time)
{
    if (principal <= 0)
    {
        throw new ArgumentException("Principal must be greater than zero");
    }

    if (rate <= 0)
    {
        throw new ArgumentException("Rate must be greater than zero");
    }

    if (time <= 0)
    {
        throw new ArgumentException("Time must be greater than zero");
    }

    Result = principal * rate * time / 100;
    return Result;
}
```

Here is an example how you would initially use GitHub Copilot Chat:



GitHub Copilot always takes context into consideration. Context is the currently open file, metadata (e.g. programming language) as well as neighboring tabs.

However, we can also enhance the context by including additional information to our prompts. One idea here is to tell Copilot about your intention and give Copilot specifications about what you want to achieve.

Here is an example prompt for Copilot

```
I am going to build a website for trainers that can be used during IT
Workshops. This website should have a couple of smaller tools that you
can pull up in between the sessions.

The website is being built in React with JavaScript.

The first component that I want to build is a timer for a coffee break.
Please help me building that timer. Here are the features:
 - Customizable duration for a break
 - Some buttons for a default break duration (5 min, 10 min, 15 min, 45
min, 1 hour)
 - A Button "Timer Start"
 - Upon clicking "Timer Start" the timer should start and count down.
 - When the timer is up an alarm should go off.

Can you build that component for me?
```

In this case we are telling Copilot what we want to do ("Build a website for trainers to be used during IT Workshops") and the technology that we are using ("This website is being built in React with JavaScript"). After that we give Copilot our requirements and we end up with a fully functional component in React that we can add to our page.

However, there might be one feature missing in the component: The toggling of the button to be a stop button when the timer is running and vice versa.

With an additional prompt we can ask Copilot for this implementation:

```
Please toggle the button to "Timer Stop" after the timer has started.
Also change the JavaScript code to stop the timer once the "timer Stop"
Button has been pushed.
```

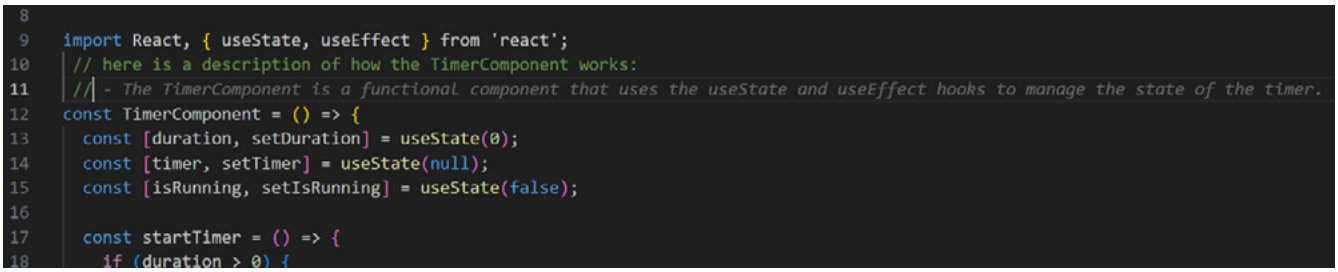
Another powerful way to utilize GitHub Copilot is by describing the desired functionality directly within the comments of your code. This approach allows you to embed detailed instructions within the codebase itself, making it easier for Copilot to generate the relevant code snippets.

Here is an example:

```
1   // Create a Timer component in React
2   // Features:
3   // - Customizable duration for a break
4   // - Default break durations: 5 min, 10 min, 15 min, 45 min, 1 hour
5   // - "Timer Start" button to initiate countdown
6   // - Countdown triggers an alarm when time is up
7   // - Toggle button to "Timer Stop" when countdown starts and vice versa
8
9   import React, { useState, useEffect } from 'react';
10
11  const TimerComponent = () => {
      const [duration, setDuration] = useState(0);
      const [timer, setTimer] = useState(null);
      const [isRunning, setIsRunning] = useState(false);

      const startTimer = () => {
        if (duration > 0) {
          setTimer(setInterval(() => {
            setDuration(prevDuration => prevDuration - 1);
          }, 1000));
          setIsRunning(true);
        }
      };

      const stopTimer = () => {
        clearInterval(timer);
        setIsRunning(false);
      };

      useEffect(() => {
        if (duration === 0) {
          clearInterval(timer);
          setIsRunning(false);
          alert('Time is up!');
        }
      }, [duration]);

      return (
        <div>
```

In this example, the comments are used to describe the entire functionality of the Timer component. When GitHub Copilot reads these comments, it can generate a complete implementation that matches the described features.

Beyond code generation, GitHub Copilot can also be used to generate documentation comments and provide explanations on how to use libraries. This can be particularly helpful for understanding complex code or unfamiliar libraries. In the example below, we trigger the copilot to describe the function by starting the description and letting the copilot get the job done.



By leveraging comments, you can guide Copilot to not only generate code but also create comprehensive documentation and explanations. This enhances code readability, maintainability, and helps in understanding and using various libraries more effectively.

#CopilotChat #ChatSyntax #Developer #Components

Maxim Salnikov

Maxim Salnikov is a tech and cloud community enthusiast based in Oslo. With over two decades of experience as a web developer, he shares his extensive knowledge of the web platform, cloud computing, and AI by speaking at and providing training for developer events worldwide. By day, Maxim plays a crucial role in supporting the development of cloud and AI solutions within European companies, serving as the leader of developer productivity business at Microsoft. In the evenings, he can be found running events for Norway's largest web and cloud development communities. Maxim is passionate about exploring and experimenting with Generative AI possibilities, including AI-assisted development.

Tip 16:

Leveraging Chat syntax: chat participants, chat variables, slash commands

by Maxim Salnikov

GitHub Copilot Chat is an incredibly powerful and useful feature that allows you to chat with or about your code. Even though it's 100% natural language-friendly (i.e., you can send your messages without using any specific syntax), leveraging certain special chat capabilities can unlock new AI-assisted development scenarios and significantly boost your productivity.

These powerful features, which you can use by applying special syntax, include chat participants, slash commands, and context variables. Note that the described features are available in VS Code and might not be fully supported in other IDEs where GitHub Copilot Chat is available.

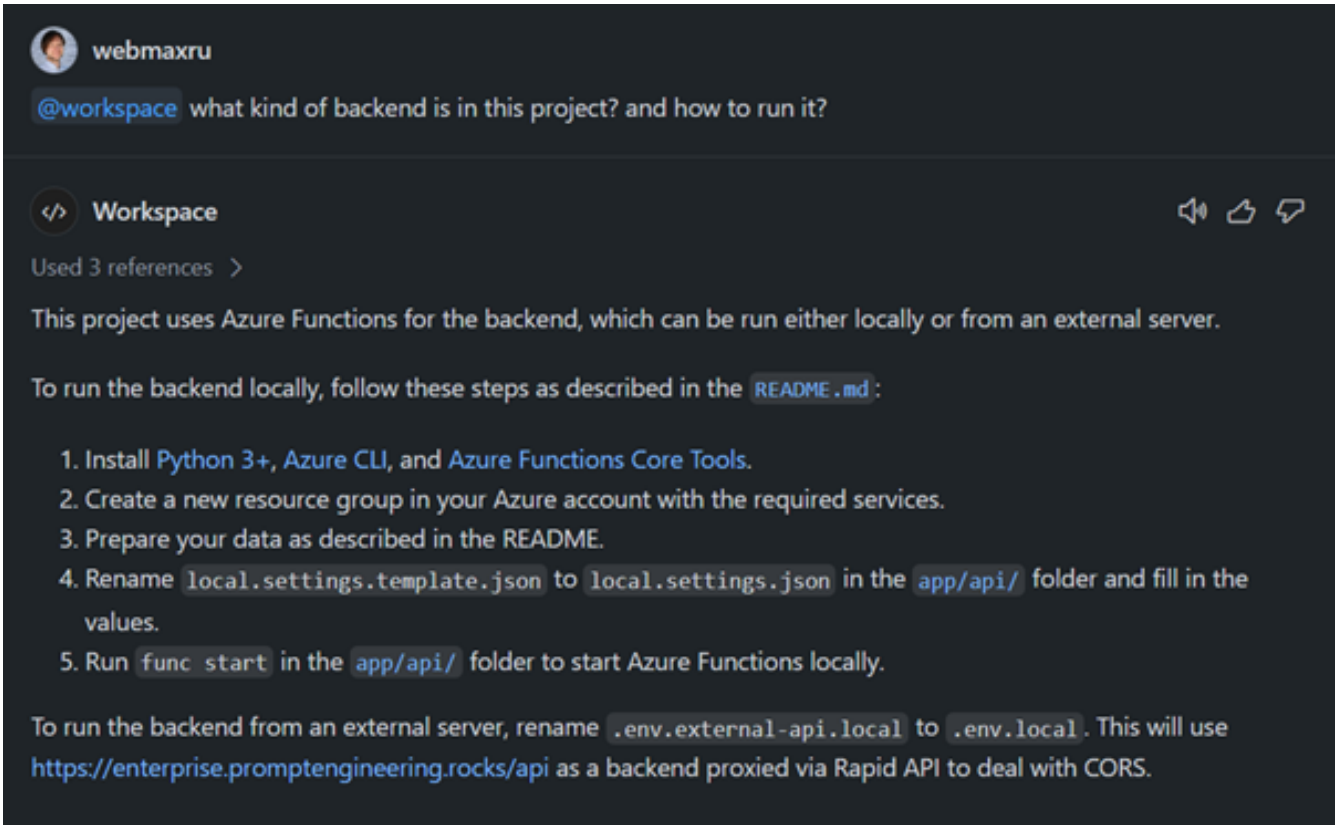
Target your question or request by messaging one of the available chat participants

In GitHub Copilot Chat, you can reference one of the AI-powered "domain experts" using conventional chat syntax—by prefixing @ to the participant name. The currently available chat participants are:

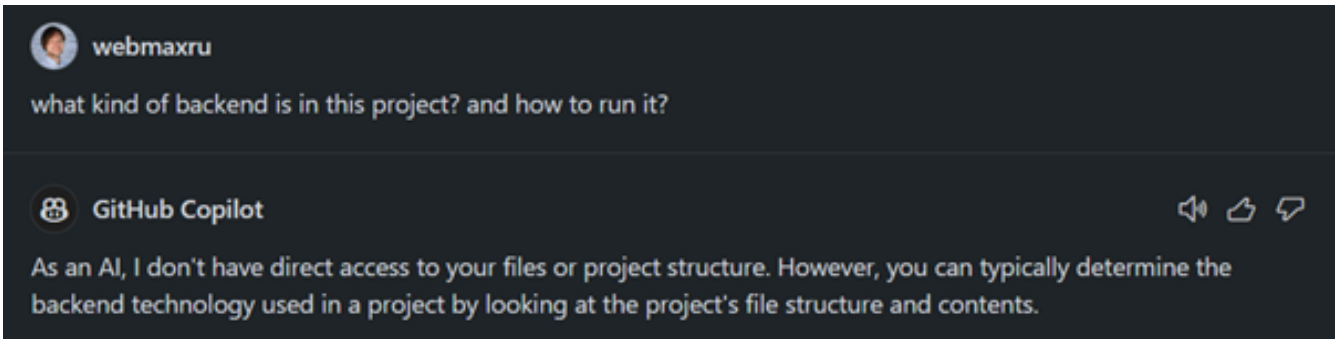
- **@workspace:** Knows everything about the code in your currently open workspace. This is the chat participant with whom you will probably communicate with most frequently.

- **@terminal:** Knows all about the integrated terminal shell, its contents, and its buffer.
- **@vscode:** Knows about the VS Code editor, its commands, and features.

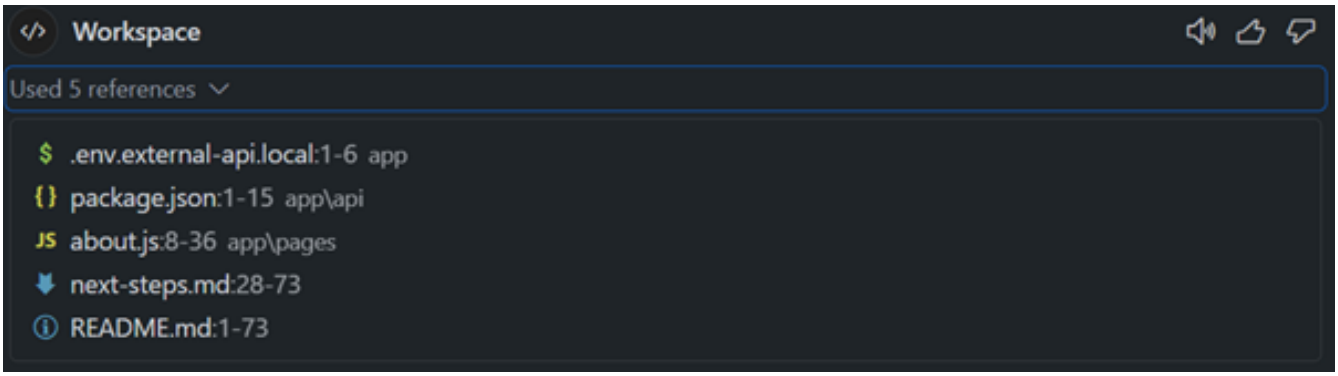
Example: Let's get information about the backend part of the project we've just been assigned to by asking the @workspace chat participant right after we open the project folder in VS Code.



In this particular case, you don't even need to have files open in your editor. Compare this with the response you get without tagging @workspace:



The @workspace chat participant is instrumental for all solution-wide queries in which you want all code to be considered for the chat response. However, this doesn't mean that all code will be used and sent as part of the prompt. The GitHub Copilot Chat extension in VS Code does its best to determine relevant files and parts of these files using local knowledge and intelligence first. You can check which files and code lines were used for the prompt by expanding the "Used references" line:



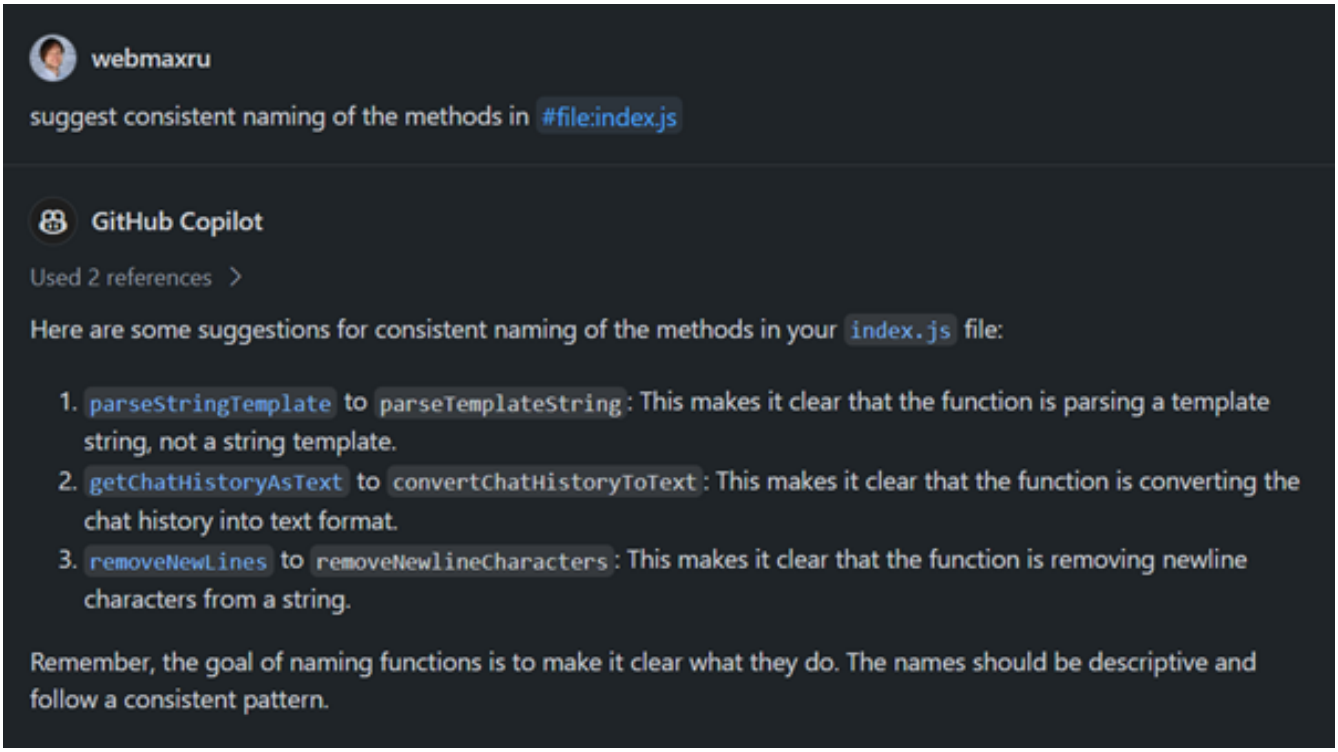
Productivity hint: Use Ctrl-Enter (Cmd-Enter) instead of just Enter after typing your message, and the @workspace string will be inserted into your message automatically before sending.

Be precise in setting the context using chat variables

In many cases, considering the full solution as the context for your question or request (by using @workspace) is overkill. You might want to point to specific files or even parts of the files in your message. Chat variables can help! Use # to call one from this list:

- **#file:** Points to a specific file in your workspace.
- **#codebase:** All content of the open workspace. It's similar to using @workspace and might be useful when you chat with another agent (like @terminal) but still want to reference the full solution.
- **#editor:** Source code in the editor's viewport (visible part).
- **#git:** Current git repository: branch, remotes, path, etc.
- **#selection:** The currently selected code.
- **#terminalLastCommand:** Last run command in the editor's terminal.
- **#terminalSelection:** Selection in the editor's terminal.

Example: Let's get help on improving method names in a specific file (and we want to ensure that the whole content of the file is taken into consideration).



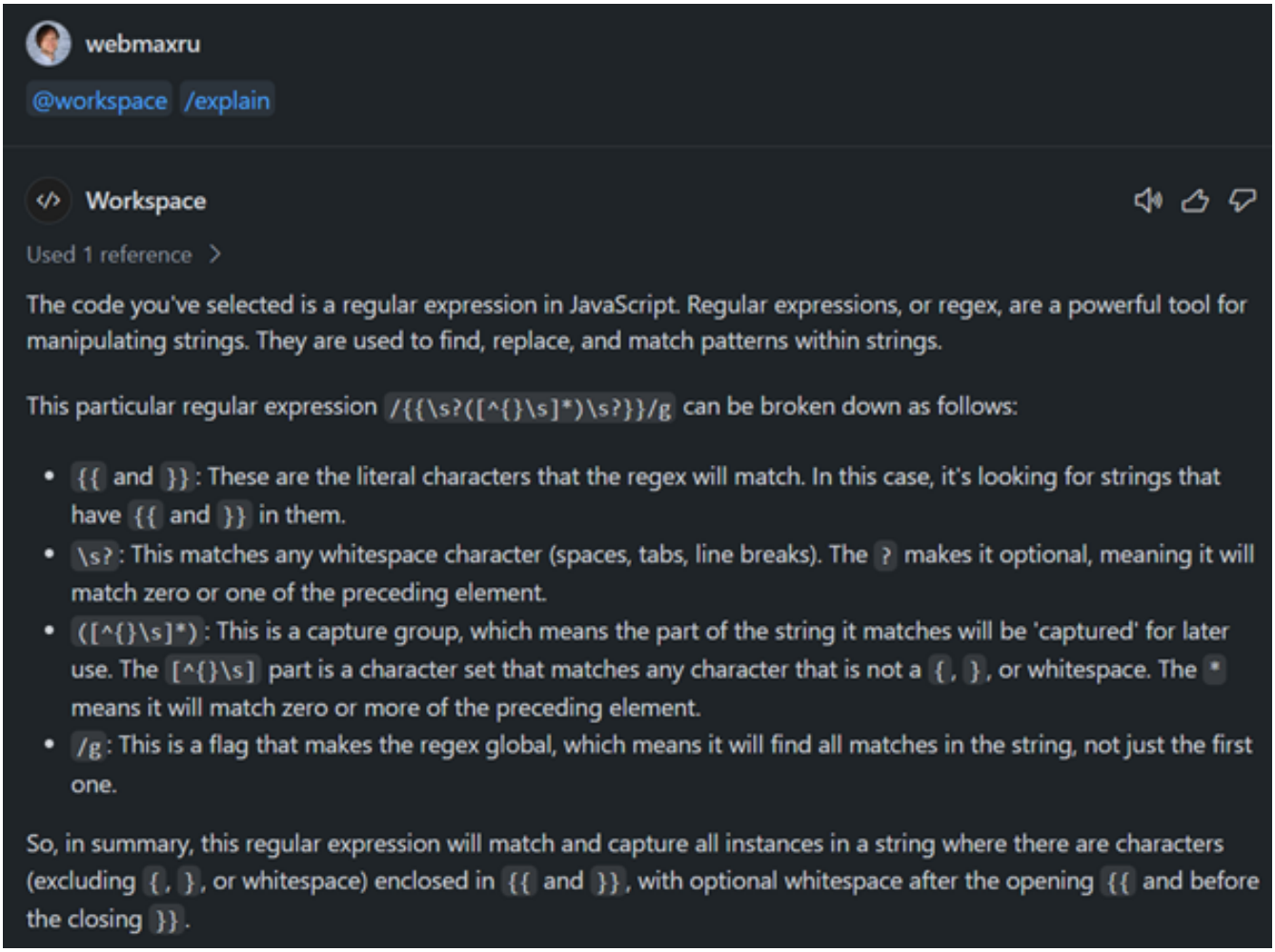
Productivity hint: Use the up and down keyboard arrows to pick the chat variable you need after typing #. In the case of #file, use keyboard navigation again to pick one of the suggested files.

Call up the most frequently used actions quickly with slash commands

Chatting with your code using natural language is fun, but having the option to call up often-used actions using handy shortcuts is even better. Compare typing the full message "Explain how selected code works" versus typing "/", then using keyboard arrows to pick "/explain" from the popup overlay. Another benefit of using the predefined syntax for commands is the confidence that GitHub Copilot understands our intent 100% correctly (natural language might have some ambiguity). There are a bunch of slash commands available. You can use them in conjunction with referencing the chat participant to provide the desired scope. Some of the commands are:

- **/help:** Help on available slash commands, chat participants, chat variables, and more.
- **/doc:** Generate documentation for the code.
- **/explain:** Explain how the code works (or get help with terminal commands if you prepend @terminal).
- **/fix:** Optimize and/or fix issues in the code.
- **/tests:** Create unit tests for the code.
- **/new:** Scaffold a new workspace.

Example: Let's get an explanation for one of the regular expressions in our code. Select the code line and use the slash command "/explain".



Productivity hint: Try GitHub Copilot Chat in inline mode instead of having the chat always open in the side pane. Press Ctrl-I (Cmd-I) and type your message in the small overlay dialog that appears right above the line where your cursor is in the code window.

Summary

Use chat participants, chat variables, and slash commands to maintain full control over the conversation context, ensure correct and consistent understanding of your intentions, and ultimately chat and code faster!

Use GitHub Copilot for your projects

These 16 tips from our experts are a great introduction to the fascinating possibilities of GitHub Copilot - but the examples presented are just a small preview of how this innovative tool can revolutionise your development processes.

The best way to experience the full power of GitHub Copilot is to try out the AI pair programmer for yourself. Integrating it into your workflow could make a big difference, whether it's speeding up routine tasks or inspiring new approaches to development.

Of course, we want to hear about your experiences and tips for using GitHub Copilot. We invite you to share your own insights and recommendations. Your valuable insights could find a place in the next edition of our whitepaper and help other developers get the most out of GitHub Copilot. Feel free to contact us at techwiese@microsoft.com.

Innovation is at the heart of software development, and GitHub Copilot is undoubtedly a game-changing tool. Take this opportunity to improve your coding experience and let's shape the future of software development together.

Get started today

If you want to learn more about the Azure cloud platform quickly, get started today:

Contact your Microsoft Account Manager
Contact Azure Sales | Microsoft Azure

Find an Expert Microsoft Partner
Find an experienced Azure partner | Microsoft Azure

Learn more about developer productivity
Azure-Developer-Productivity | Microsoft Azure

Additional resources

- Features of GitHub Copilot
- GitHub Copilot documentation
- Quickstart for GitHub Copliot
- GitHub Copilot in VS Code
- GitHub Copilot Labs
- GitHub Copilot Fundamentals: Understand the AI Pair programmer
- GitHub Copilot Trust Center
- GitHub Blog