GITHUB COPILOT & POLYAPI

Understanding their differences and how they can work together.

Darko Vukovic & Dan Fellin

PolyAPI

Summary

This paper outlines key considerations for utilizing Microsoft GitHub Copilot, particularly in comparison and conjunction with PolyAPI. The objective is to convey my understanding as accurately as possible, enabling the reader to swiftly discern the advantages and disadvantages of PolyAPI versus GitHub Copilot. This will aid in comprehending the roles of each system, their intersections, and synergies. My overarching aim is to elucidate why relying solely on GitHub Copilot is inadequate for realizing the full potential of exceptional API discovery, consumption, and orchestration.

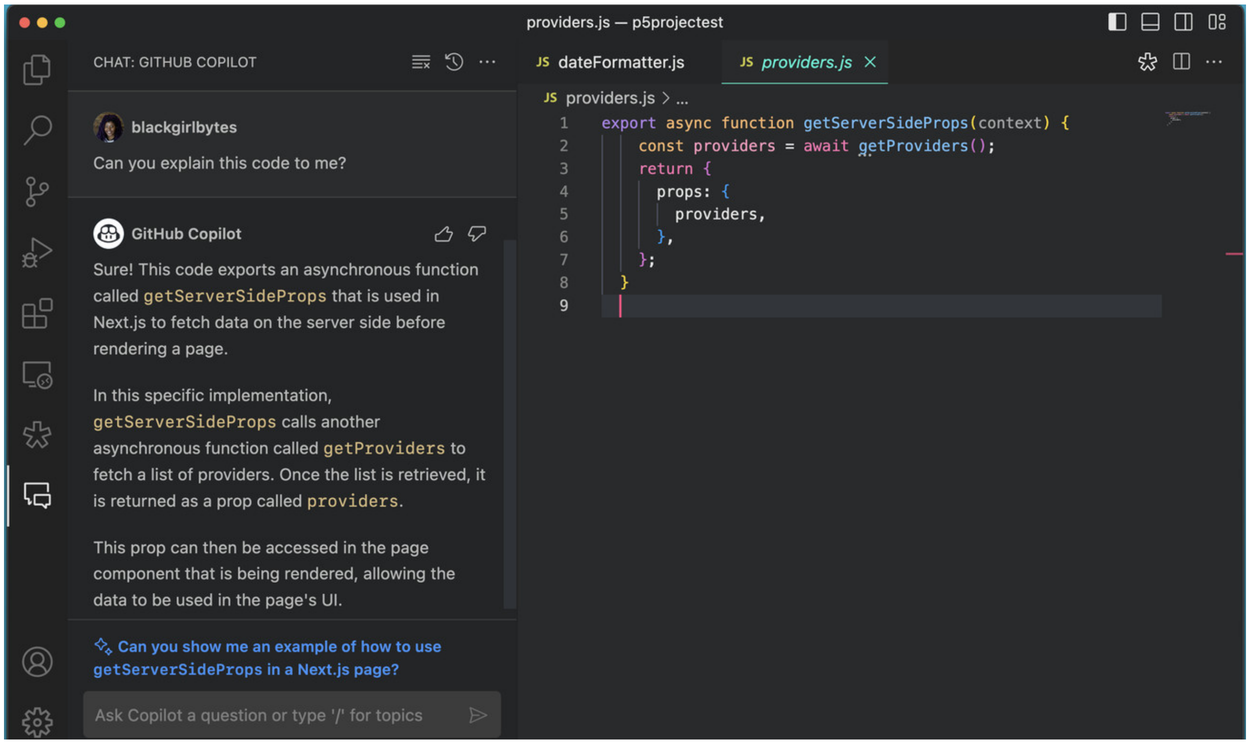
The paper primarily focuses on the architectural implications and how these will affect and constrain an enterprise's ability to leverage AI for accelerating internal development. It aims to illustrate the architectural decisions behind PolyAPI and GitHub Copilot, as understood by me, and the long-term consequences of these decisions. Additionally, it attempts to sketch a broader picture of software development using APIs and to pinpoint scenarios where a more holistic approach which leverages both systems is necessary, beyond mere AI-driven API discovery and consumption.

All the information and insights presented here are based on my decade long experiences in the realm of APIs and software development and are influenced by my own knowledge gaps and personal biases. While I strive to assert strong, well-founded opinions, I encourage the reader to maintain a healthy skepticism and conduct their own assessment to determine the validity of my viewpoints.

Table of Contents

Introduction

Upon comparing the user interface (UI) of the PolyAPI IDE extension with the newly announced GitHub Copilot Chat UI, it becomes readily apparent why one might find them confusingly similar.


providers.js — p5projectest
CHAT: GITHUB COPILOT
dateFormatter.js
providers.js
providers.js > ...
Ask Copilot a question or type '/' for topics

blackgirlbytes

Can you explain this code to me?

GitHub Copilot

Sure! This code exports an asynchronous function called `getServerSideProps` that is used in Next.js to fetch data on the server side before rendering a page.

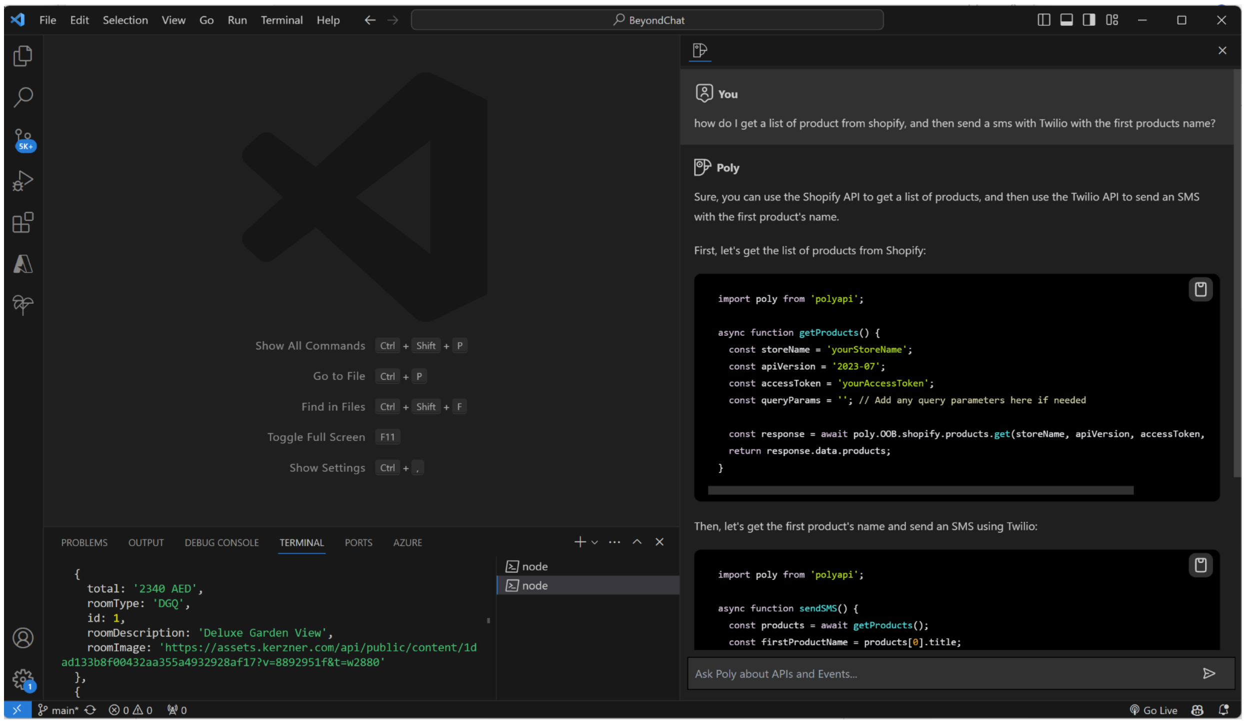
In this specific implementation, `getServerSideProps` calls another asynchronous function called `getProviders` to fetch a list of providers. Once the list is retrieved, it is returned as a prop called `providers`.

This prop can then be accessed in the page component that is being rendered, allowing the data to be used in the page's UI.

Can you show me an example of how to use `getServerSideProps` in a Next.js page?

```
export async function getServerSideProps(context) {
  const providers = await getProviders();
  return {
    props: {
      providers,
    },
  };
}
```

GitHub Copilot Chat in VS code (Credit theverge.com [Link](#))



```
  {
    total: '2340 AED',
    roomType: 'DGQ',
    id: 1,
    roomDescription: 'Deluxe Garden View',
    roomImage: 'https://assets.kerzner.com/api/public/content/1d
ad133b8f00432aa355a4932928af17?v=8892951f&t=w2880'
  },
  {
```

You

how do I get a list of product from shopify, and then send a sms with Twilio with the first products name?

Poly

Sure, you can use the Shopify API to get a list of products, and then use the Twilio API to send an SMS with the first product's name.

First, let's get the list of products from Shopify:

```
import poly from 'polyapi';

async function getProducts() {
  const storeName = 'yourStoreName';
  const apiVersion = '2023-07';
  const accessToken = 'yourAccessToken';
  const queryParams = ''; // Add any query parameters here if needed

  const response = await poly.OOB.shopify.products.get(storeName, apiVersion, accessToken,
  return response.data.products;
}
```

Then, let's get the first product's name and send an SMS using Twilio:

```
import poly from 'polyapi';

async function sendSMS() {
  const products = await getProducts();
  const firstProductName = products[0].title;
```

PolyAPI Extension in VS Code

This similarity naturally leads to an important question:

Why is it necessary to use both? Is it possible to simply integrate my API and Event data into GitHub Copilot through its extensibility model, thus eliminating the need for PolyAPI?

The purpose of this paper is to guide the reader through the complex enterprise architecture related to this question. My aim is to demonstrate why combining the strengths of both systems results in a synergy that surpasses what either could achieve individually.

Training source data

A core principle of PolyAPI is to train directly on actual APIs rather than on approximate representations like specifications or documentation. Although this requires presenting the API to Poly via Postman, PolyAPI's vision and short-term roadmap include a feedback loop that learns in real-time, identifying discrepancies between the live API and its internal data. I strongly believe that API documentation will always be an approximation, and striving for 100% accuracy through a specification like OpenAPI is an asymptotically challenging and costly endeavor.

In pursuing the goal of perfect specifications, enterprises may encounter these issues:

- Missing specifications
- Incomplete specifications
- Incorrect specifications

These issues often arise from:

- APIs acquired through vendor applications or partner integrations
- APIs developed by teams with nascent API programs, possibly due to cultural differences or recent acquisitions
- APIs that are deprecated, overseen by reorganized teams, or maintained by overburdened developers

To mitigate these problems, enterprises may resort to costly methods such as:

- Imposing stringent release standards for APIs, which slows down development
- Utilizing tools to generate specifications from code, each with its own costs, privacy concerns, and limitations
- Continuously abstracting each partner/vendor API into a new, more controllable API

From my experience, I advocate for training directly on the API itself and treating training data as live, real-time information in a database. Continuous, automatic updating of training data is essential to avoid staleness. Similarly, any AI interfacing methods, like embeddings or RAG-based approaches, should use this live database to provide developers with accurate information.

Observations of developer behavior support this approach. Developers often distrust OpenAPI specifications and usually test API calls (commonly with Postman or curl) before accepting documentation as accurate. They prefer to verify an API's functionality firsthand before integrating it into their code. This skepticism towards API documentation has contributed to the success of tools like Postman and underscores why AI services relying solely on specifications are likely to fail.

Training on documentation poses a significant challenge: the AI's inability to self-correct due to a lack of introspection into the real API. This will translate to AI having a stubborn adherence to incorrect information. Developers, who typically have little tolerance for such errors, will quickly lose trust in the AI after a few mistakes, reverting to tools like Postman for API validation and hand crafting code. The unique complexities of enterprise systems, especially with custom objects, fields, and business rules, exacerbate the difficulty of accurate representation. This is why at Poly we set a hard principle to only train off real API calls.

Compiler & IDE Services

At Poly API, we transform API training data into a library that abstracts the developer's code from the API. We initially contemplated assisting users in generating direct API calls but ultimately decided against it for several reasons. Relevant to this paper's topic, we decided to leverage compiler assistance and coding services within IDEs to validate the use of our AI-generated code. Once an API is trained, Poly generates a library that represents the API as Poly perceives it (and soon, Poly will continuously monitor it). This library, when generated and included in a development project, provides API-specific context, including complete object models, enabling compilers and IDE services (like TypeScript Server and its Java/Python counterparts) to interpret the API's actual type definitions. Although the Poly AI assistant uses the same source data to generate code as the library, it may not always produce code that is type-safe or considers all possible failure scenarios and conditions. When developers incorporate this code into their projects, compilers and IDE services often identify areas where the code needs strengthening.

Moreover, having pre-modeled type definitions allows developers to incorporate them into their code beyond the AI generated code, benefiting from type safety across their entire project. While AI-powered discovery and code generation are important facets of the Poly platform, I argue that the crux of our solution lies in generating the library that enumerates functions, interfaces, and object models. These generated libraries form the basis of everything else Poly offers.

Referring back to the previous section about developers' zero tolerance for errors, I believe that code failing during testing or runtime will lead to significant skepticism towards AI-generated code. Developers might feel betrayed if such code fails in production, with them bearing the responsibility. Therefore, it's vital for the AI agent to validate its assumptions about underlying APIs through real-time API calls and by harnessing compiler and IDE services to identify oversights in type-specific and conditional logic is crucial.

Embeddings, Fine Tunings, Prompting

Several approaches exist for AI-powered discovery experiences like those offered by Poly or GitHub Copilot Chat, each with its own set of advantages and disadvantages. At Poly, we've utilized both embeddings and prompt engineering, and although we considered fine-tuned models, we ultimately decided against them for various reasons.

Prompting, especially with a Retrieval-Augmented Generation (RAG) approach, requires a system for generating prompts. This can become complex with multi-step prompts, where any step might fail. For instance, Poly currently employs a three-level deep prompt sequence and plans to expand it to four levels. This design aims to ensure infinite catalog scalability, support multiple AI providers, reduce hallucinations, and provide highly accurate API details.

The main benefit of prompt engineering is the control it offers at each step—identifying APIs, selecting winners, generating code, and validating results. This control allows us to fine-tune the experience and make adjustments at various stages in our process.

From my best understanding of GitHub Copilot's extensibility model for training on user-provided data, it involves a process where an enterprise supplies its data via a GitHub repo, and GitHub Copilot then constructs an embeddings model for runtime use. This is a generic extension model, which can be used for training on APIs by hosting API documentation within one or more repositories. The strength of embeddings lies in their versatility; they can handle data of any form, structured or unstructured, including metadata or free text. A simple conceptualization of this approach is to imagine “talking to your documentation” where in the case the documentation is API documentation.

There are significant downsides to an embeddings-based system:

- **False Positives and Lack of Control Mechanisms:** With embeddings, there's a single AI call to select content, leaving no room for enterprises to influence the selection algorithm. High-scoring but irrelevant content might be chosen, used without validation for correctness.

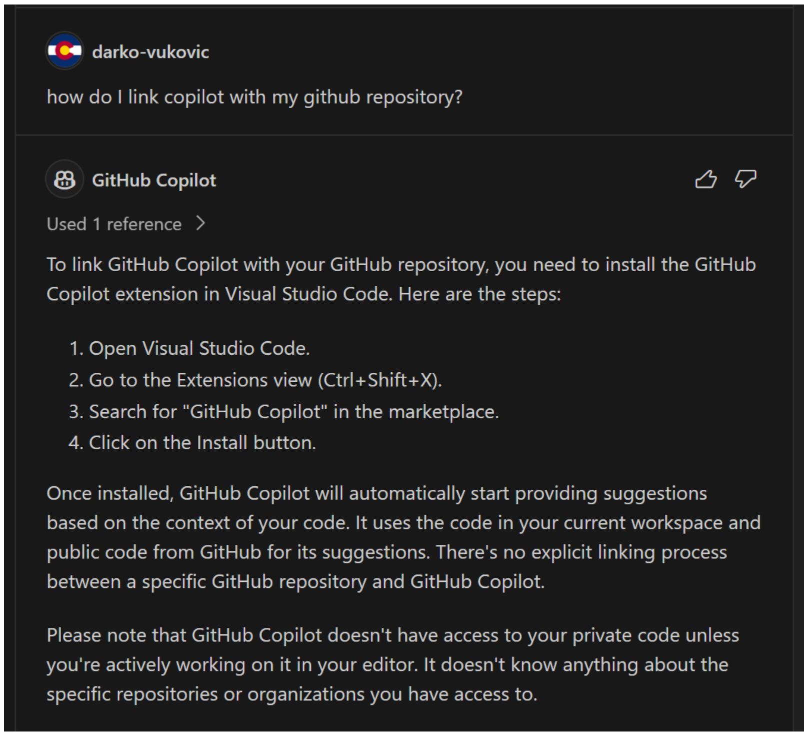
- **Lack of Change Management and Control**: General-purpose systems by large vendors tend to be generic, leaving much work for the customer. For API discovery, this means preparing and submitting content for embeddings generation, a challenging task for enterprises with thousands of endpoints. Specifically, customers will have to address these considerations:
    - **Optimization for High Recall**: The inability to influence the AI's selection process poses a challenge compared to full control with prompt engineering.
    - **Delay in Usage:** Regular training is essential, but the frequency can impact the cost and cause frustration when revisioning the content and testing with the AI in a realistic full catalog setting.
    - **Cost Implications**: Deciding between retraining on all content or just updated parts, and identifying those changes, affects costs.
    - **Segmentation Challenges:** Properly partitioning API documentation into blocks and managing size limitations is crucial especially with OpenAPI specifications which are large or reference common data models.
    - **Large Prompts Issue**: Selecting how many blocks should be injected in the prompt can form large prompts can be expensive and error-prone.

From my experience, attempting to repurpose Copilot for API discovery and consumption through an embeddings-centric approach is likely to encounter significant limitations unless a heavy lift in engineering investment is made to build a Poly-like service internally.

Access Controls

PolyAPI is specifically designed with enterprise-level access controls in mind. In a typical large enterprise, there are numerous development teams, partners, contractors, and suppliers, all working within a shared catalog of services. In such an environment, where collaborative efforts span across various organizational segments, it's essential for the API discovery and consumption experience to acknowledge that not every user has access to all APIs. To address this, PolyAPI implements three levels of access controls: instance level, tenant level, and environment level. These tiers allow enterprises to manage who has access to specific API information, tailoring visibility based on the chosen publication level.

However, the integration of these access controls with AI models, especially those building embeddings, presents a major challenge. At time of this writing, it's unclear how GitHub repo-based approach can accommodate individualized access controls during training. In contrast, if Copilot adopts a prompt-based RAG model (hopefully based on OpenAI Plugins), PolyAPI can inherently align the APIs/functions it presents with the user's permissions.


darko-vukovic

how do I link copilot with my github repository?

GitHub Copilot

Used 1 reference

To link GitHub Copilot with your GitHub repository, you need to install the GitHub Copilot extension in Visual Studio Code. Here are the steps:

1. Open Visual Studio Code.
2. Go to the Extensions view (Ctrl+Shift+X).
3. Search for "GitHub Copilot" in the marketplace.
4. Click on the Install button.

Once installed, GitHub Copilot will automatically start providing suggestions based on the context of your code. It uses the code in your current workspace and public code from GitHub for its suggestions. There's no explicit linking process between a specific GitHub repository and GitHub Copilot.

Please note that GitHub Copilot doesn't have access to your private code unless you're actively working on it in your editor. It doesn't know anything about the specific repositories or organizations you have access to.

In the absence of PolyAPI, if my Assumptions about a repo-based learning model, where Copilot inherits a users credentials, enterprises are likely to face one or more of the following issues:

- Discovery of Inaccessible APIs: Users might discover APIs they don't have permission to use. This problem is compounded if a usable API is available but ranks lower in the search results, thus not being recommended in the UI by the AI. This would happen if a single central repo was used to represent the whole enterprise catalog.
- Overlapping Catalogs: Beyond the issues mentioned above, there may be a need to develop additional logic to create multiple catalogs (repos) that approximate permissions for different user segments. This would mean creating audience specific OpenAPI specifications and publishing them to the appropriate GitHub Repos. Copilot hopefully would be intelligent enough to answer questions based on the higher capability repo if prompted by a developer with access to both.

- **Exclusion of Developer Groups:** Certain groups, such as partners or contractors, might be inadvertently excluded from accessing the enterprise API catalog all together if there is no easy way to easily organize catalogs by repository.

These challenges underscore the importance of treating APIs as data subject to role-based access control (RBAC). They also highlight why I believe a prompt engineering approach, where data selection is based on the identity of the user seeking assistance, is more suitable for managing API accessibility in an enterprise setting.

Exclusive Poly Features

This section highlights unique features offered by Poly that are not a part of GitHub Copilot and are unlikely to be included in the foreseeable future. These features significantly contribute to my belief that GitHub Copilot alone, even with implemented solutions to previously mentioned challenges, cannot provide a comprehensive developer experience for building applications and integrations with APIs.

Event Handler Functions

Poly enables the registration of event providers as functions within its catalog of discoverable functions. These event handler functions are more advanced than standard API functions, as they automatically establish a websocket connection to the Poly server and seamlessly subscribe to events. This capability simplifies event subscription for developers, regardless of the event protocol or type. Usually, event providers are not listed in API catalogs and are not described by OpenAPI specifications. They will introduce a second data type and source for training Copilot.

Client Side Functions

Included within Poly's library, these functions exist outside of endpoints and are not typically registered in API catalogs, as they are not APIs themselves. Instead, they are helpful functions designed to facilitate API usage, offering optimizations for specific use cases, reusable patterns, API testing, log collection, alert generation, and more. API providers can offer these custom functions as part of client libraries, providing interfaces and capabilities beyond standard API services. It is improbable that these functions could be trained into Copilot unless they are generated as libraries, whereas Poly automatically incorporates them into its catalog and uses RAG to present them when appropriate. Again, this could be a third data type and data source for training Copilot.

Server Side Functions

Poly offers a runtime environment for newly developed functions, deployable and hostable within an enterprise's Kubernetes instance. These services are easy to maintain and become discoverable upon deployment. They inherit the same access controls as other functions, enabling enterprises to set up new APIs and manage access efficiently. Each function is assigned an ID, which I will reference later.

Credentials and Configurations

Utilizing Hashicorp vault for secure storage of secrets and credentials within an enterprise's Kubernetes instance, Poly simplifies the process of obtaining, using, and updating these credentials and configurations. Our roadmap includes adding Azure Key Store and eventually more secrets vault providers. Poly AI assistant aids users in discovering credentials like URLs and keys. A significant advantage of Poly's library is its ability to handle credentials at runtime through .inject() statements, ensuring that neither developers nor their applications directly access sensitive information, streamlining their development experience compared to traditional manual vault access.

AI Agents as a Service

Poly users can create AI Agents by referencing function IDs through the Poly API Plugin generation service. Currently, we support OpenAI Plugins, with plans to expand to other major AI providers. These agents can be published to marketplaces or embedded into enterprise applications. By addressing functions by ID, changes in the functions' interfaces do not affect the AI Plugin, relieving developers of maintenance work. This ensures that services depending on the AI-powered service remain unaffected as the AI Agent evolves. Additionally, Poly offers a history service for maintaining and analyzing conversation data for performance enhancement and prompting.

GitHub Copilot and PolyAPI Working Together

In concluding, I'd like to outline our vision for how PolyAPI and GitHub Copilot can effectively collaborate. We view these two services as highly complementary and aim to clarify their distinct functionalities. GitHub Copilot excels as a general-purpose development aid, with API consumption being one of its many capabilities. Areas where Copilot will excel are ramping up new developers on a code base, hardening code, writing unit tests, and much more. It's important for readers not to underestimate the complexity of extending Copilot via a repository based approach; integrating an entire API catalog may seem straightforward, but it involves significant challenges.

Our vision positions PolyAPI as a vital component within an enterprise's own Kubernetes (K8) cluster. PolyAPI is dedicated to collecting real-time API metadata, constructing an accurate and dynamically updated model of these APIs, and forming a comprehensive database that encapsulates this institutional knowledge. Through prompt engineering, PolyAPI could provide Copilot with up-to-the-second information on APIs, events, and custom functions **if Copilot offers the ability to implement skills in a similar fashion to how OpenAI plugins work**. This would be achieved via a discovery service that incorporates a multi-step process combining RAG and conventional services. This process not only considers the identity of the user submitting the prompt but also tailors the responses accordingly.

We envision GitHub Copilot as a preferred interface for many developers. Within this framework, PolyAPI acts as an extension or 'skill,' enabling Copilot to access real-time enterprise API data, enriching its capacity to deliver precise and contextual information to developers. Additionally, PolyAPI offers various services, as previously detailed, to simplify the development of integrated applications.

While it's currently unclear if GitHub Copilot will support real-time RAG, we are confident that it will eventually adopt this architecture, as it is the most logical solution for this and potentially other programming challenges and since it has already been developed so well by OpenAI with the Plugins Model.

Should any part of this discussion seem to misrepresent GitHub Copilot, the intricacies of these issues, or the involved technologies, we would greatly value your feedback at feedback@polyapi.io.

Darko Vukovic
CEO & Founder of PolyAPI