PolyAPI

Integrations and Microservices

A Match Made in Heaven

This paper covers the key benefits of using serverless microservices when developing and running integrations. We start by defining and illustrating the fundamental primitives (building blocks) of integrations using Poly, then outline the key business and technical benefits of applying these approaches in production. After reading this paper, we hope you will better understand why we chose the approach we used in Poly’s architecture and why we strongly feel it will serve you better as a potential user than a traditional integration runtime model. We hope this paper is informative for even those not in the market for an integration platform, as these same concepts and principles can be adopted more broadly for the software development of integrated applications as well.

Enjoy!

AUTHOR: **DARKO VUKOVIC**
PUBLICATION DATE: **03/20/24**

Note to Competitors

If you are reading this from a Poly competitor, ask yourself why you are working at your company and reading about innovation in your competitors' whitepapers when you could instead be working at Poly and pushing innovation forward. If you are driven and want to be at the forefront, reach out at info@polyapi.io.

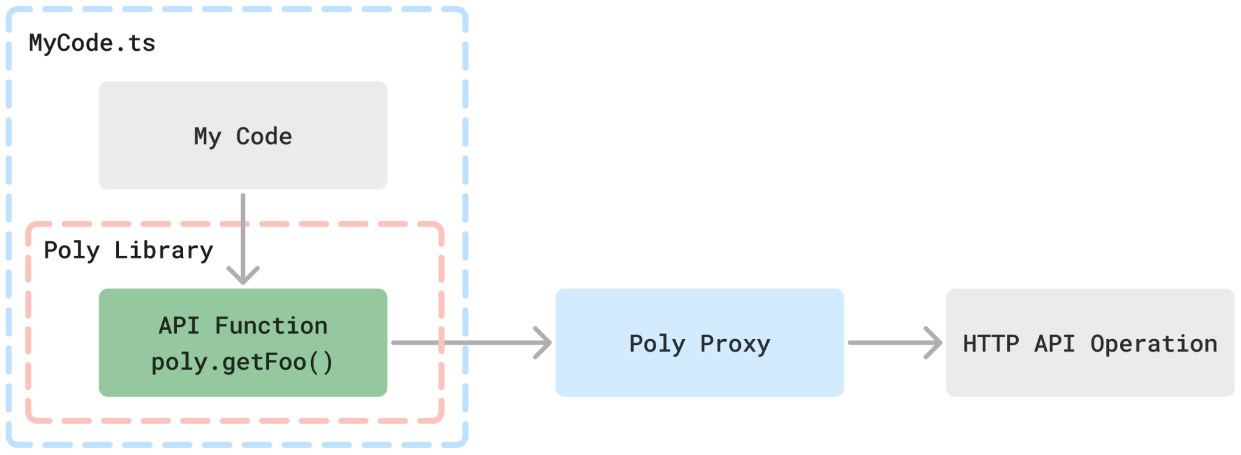
The Primitives

As many of these terms mean different things to different people, below is a list of core Poly primitives that play into developing integrations using microservices with a short description of what we mean when we use them. We then provide an illustration of how these primitives assemble together in a few different use cases. Note that all of this is written from the perspective of a developer building an integration using a "high code" IDE such as VS Code or IntelliJ.

API Functions (Operations)

In Poly, each API function is tied to a specific API operation. Think of a function as the most basic building block. Integrations are typically constructed by combining various functions. The key role of an API function is twofold: **firstly, it gathers the necessary data from the integration developer for the downstream API**. **Secondly, it handles the response, ensuring that it is delivered in a structured manner which is consistent and type-safe for programming.**

For instance, consider a function designed to fetch user details from an API operation. When this API function is executed, it performs the corresponding API operation – retrieving the user details – and then returns the actual response from the API. A critical aspect of Poly is to maintain a seamless alignment between the model (what's defined in the function) and the actual response, ensuring a 1:1 correspondence.

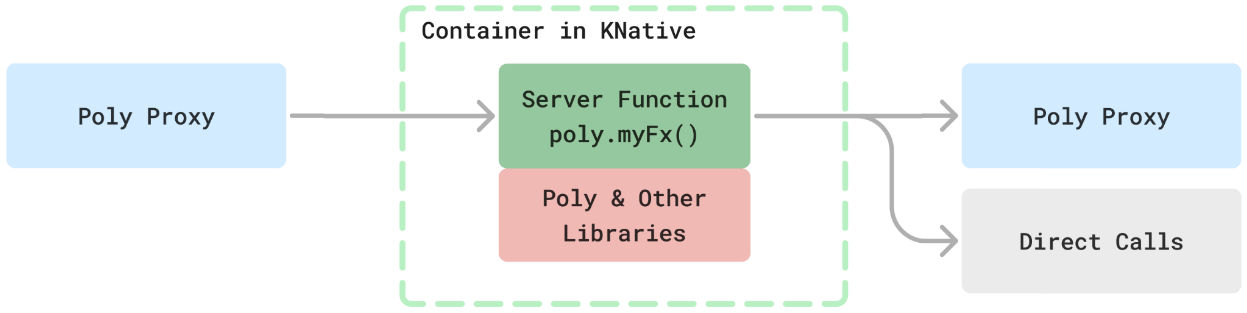


Server Functions (Serverless)

Server functions in Poly are a type of serverless computing, meaning they don't require a dedicated server to run. Instead, they are managed by Poly and run on Poly's KNative platform, which handles their execution automatically. These functions are ideal for specific, standalone tasks or services that aren't directly provided by a system's API.

Think of server functions as specialized tools designed to create new services, orchestrate workflows, and handle complex integration logic. They're particularly useful for developing new features or services that augment existing systems. Additionally, these server functions have the flexibility to interface directly with other protocols or databases, offering a powerful means to interact with and manipulate a wide range of data sources

A vital factor in using server functions (instead of client functions) is considering the privacy and security of the code and the services they interact with. For example, consider a token minting/validation service or a database wrapper service. In these cases, it's crucial that the underlying mechanics of the server function are not exposed to the developers using them. This ensures that sensitive details about the function's operation are kept secure, and the users of the function cannot alter its code, preserving its integrity and security.

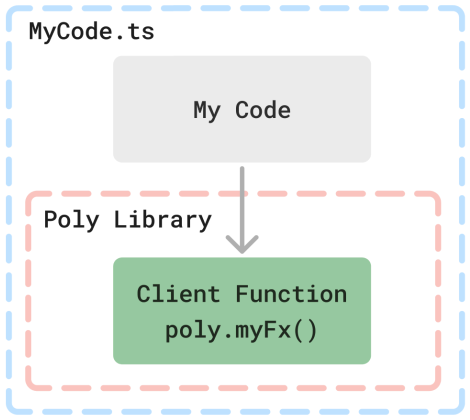


Client Functions (Utilities)

Client functions in Poly are similar to server functions, with a key difference: **client functions do not operate on a dedicated runtime. Instead, they execute within the client application that utilizes them.** This approach means that client functions are inherently language-specific and their source code is accessible to the developers who use them.

For instance, imagine a pre-built client function that handles data validation within a web application. Since it's executed on the client-side, it's designed specifically for the web application's programming language, and the developers can view and understand the source code. This visibility can be beneficial for reuse, learning, and customization.

Client functions excel in providing 'pre-built' code components, significantly enhancing code reusability. They offer a convenient way to boost productivity among developers, allowing them to leverage existing code without incurring additional infrastructure costs or operational overhead. This makes them an excellent tool for building more efficient and effective client-side applications.

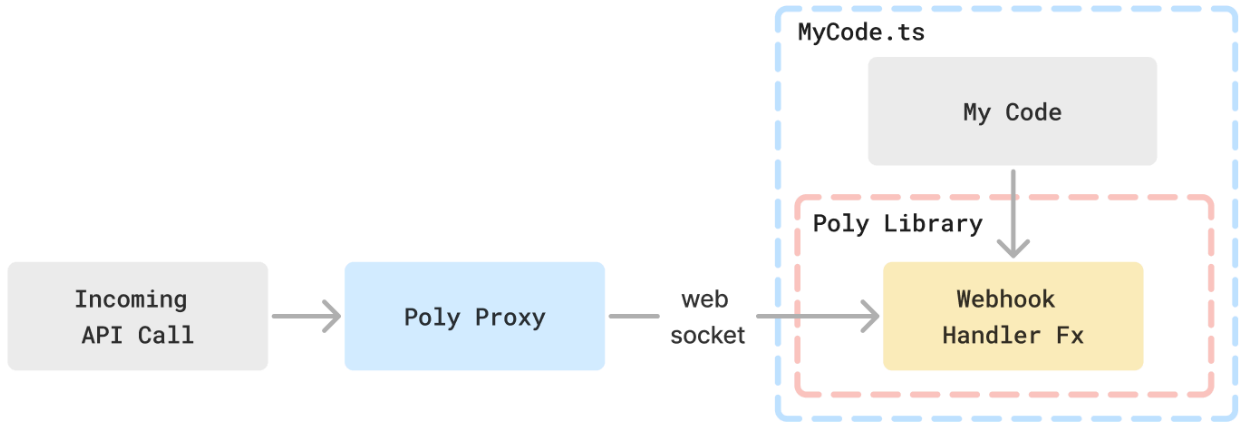


Webhooks Handlers (HTTP Listeners or APIs)

Webhooks form an essential part of Poly's core Gateway service. Simply put, **webhooks are configurations that dictate how incoming requests at specific endpoints should be processed**. They represent a set of expectations about the data that will be received. When we refer to Webhook Handlers in Poly, we are not only talking about processing these requests but also encompassing the mechanisms for triggering events within Poly and the functions that automatically subscribe to these events.

From a developer's perspective, these handlers are versatile. They can be used to create and expose new APIs to others or to listen for incoming webhook calls. The key distinction here lies in who defines the interaction interface. When exposing an API, it's the producer who sets the interface. In contrast, when setting up a webhook listener, it's the upstream client that dictates the interface, and the integration developer adapts to it.

Think of Webhook Handlers as the broader category of 'Event Handlers.' While currently focused on HTTP events, in the future, we envision expanding or adapting this mechanism to include other types of event streams, such as Kafka Topics. For the purpose of this whitepaper, consider Webhook Handlers as a universal tool for handling various event-driven interactions.

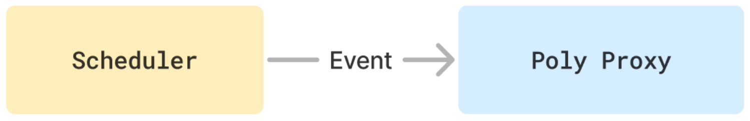


Schedulers (Cron Jobs)

In Poly, a scheduler acts as an initiator of events based on specific time configurations, similar to traditional cron jobs in computing. It is designed to automatically generate event requests at predetermined times and trigger the execution of one or more server or API functions, either in parallel or sequentially.

For instance, imagine a scheduler set to initiate a daily batch synchronization of records. At the scheduled time, it activates the necessary functions to initiate and manage the backup process.

Schedulers are incredibly versatile and can be employed for a variety of purposes. They are commonly used for triggering batch processes, conducting regular health and state checks of systems, automating tests, and handling routine maintenance tasks. Essentially, they bring automation and reliability to tasks that need to occur at regular intervals, enhancing efficiency and consistency in operations.

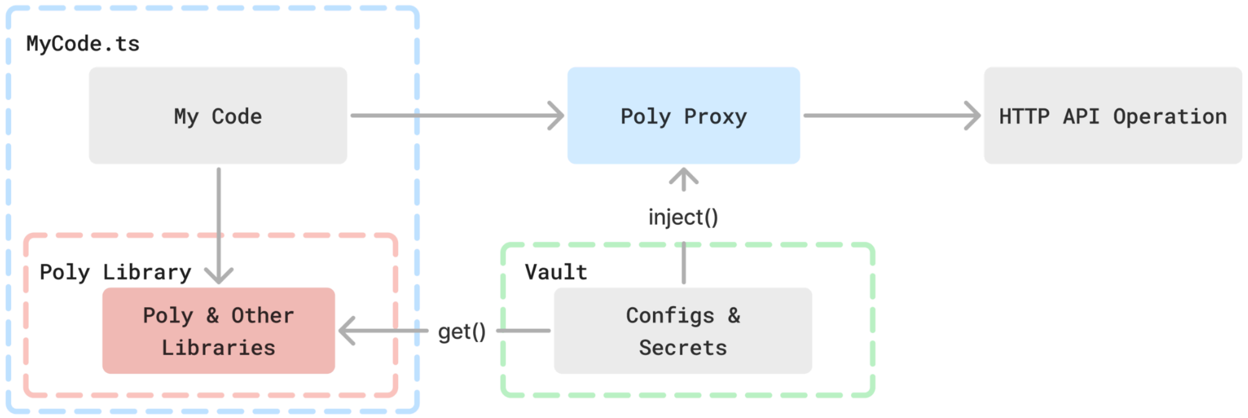


Variables (Secrets and Configs)

Variables in Poly are essential pieces of information used across integrations, orchestrations, and backend development. They might contain sensitive data or general information. **The primary characteristic of these variables is their dynamic nature: they can be updated or managed without the need to re-deploy the entire code.** This flexibility is crucial for maintaining efficiency and responsiveness in a dynamic development environment.

Variables play a vital role in various applications, such as accessing configurations, storing credentials, managing mapping files, serving as feature flags, or even holding example data for testing. They are stored securely in a 'vault' – either the default Poly Vault or, in the near future, a customer’s own cloud vault. This vault system ensures that sensitive information is kept secure and only accessible to authorized entities.

To use these variables, they can be retrieved from the vault for local use (in cases where they aren't secrets) or passed by reference and resolved securely through the Poly proxy for sensitive data. This approach ensures that variables, especially those containing secrets, are handled with the utmost security, preventing potential vulnerabilities in the system.



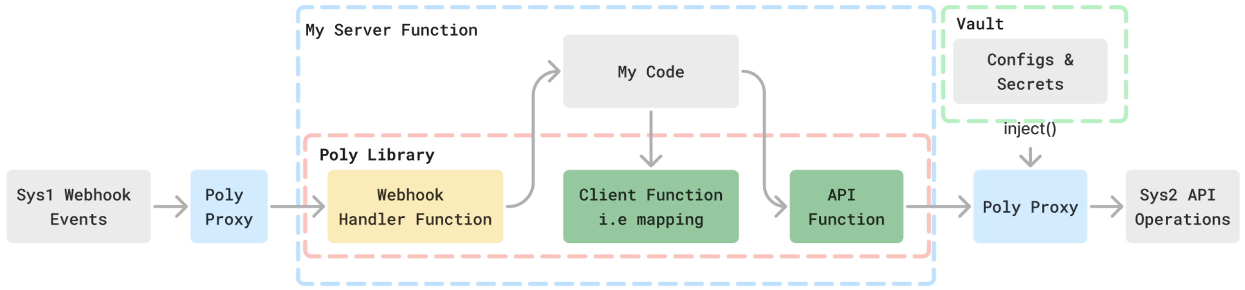
The Composites

In the realm of Poly, integrations, orchestrations, and backends are what we refer to as 'composite' types. These are more complex structures that are built using a combination of the primitives previously described. If we consider the primitives as atoms, the fundamental building blocks, then these composites are like molecules – each a unique combination of atoms (primitives) configured for a specific purpose..

Integrations

A typical integration in Poly is generally a set of ‘flows.’ Each flow is essentially a defined pathway for moving a specific type of data or object from one system to another, triggered under certain conditions. These conditions can vary - some integrations are event-driven, reacting to specific occurrences, while others are scheduled, operating as batch jobs at predetermined times.

An example of an integration could be the automatic synchronization of customer data between a CRM system and a marketing automation tool. Whenever a new customer is added to the CRM, the integration flow is triggered, ensuring that the marketing system is updated with this new information. This process doesn’t create new data; rather, it ensures that existing data is shared efficiently between systems, enabling them to work together seamlessly in support of complex business processes.

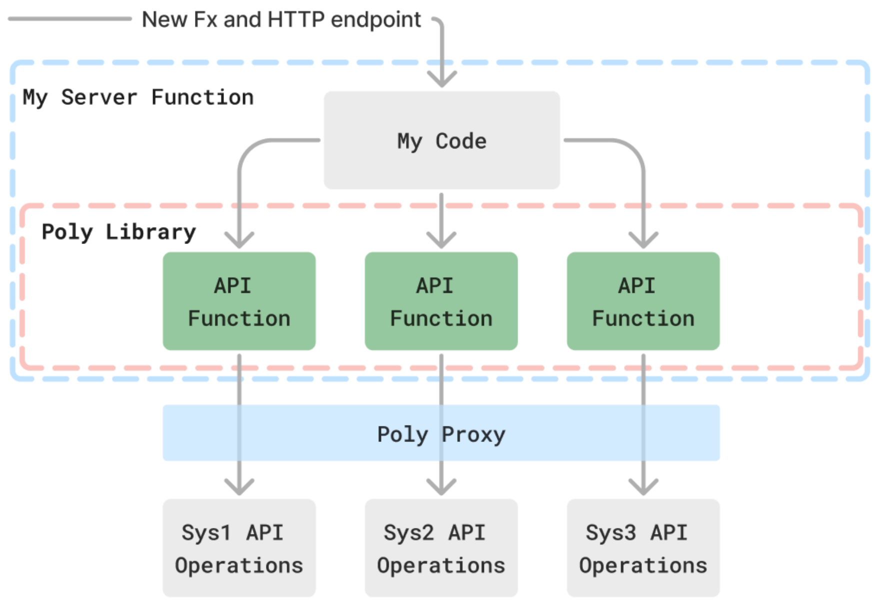


Orchestrations

In the landscape of Poly, orchestrations represent a sophisticated form of composition where new functionalities are developed. These functionalities emerge from the integration of multiple services from various systems. Essentially, the outcome of an orchestration is a novel service, complete with its own endpoint, ready for utilization via Poly functions and HTTPs.

Imagine an orchestration as a maestro conducting an orchestra, where each musician (service) plays a part. When combined, they create a symphony (a new service) that didn't exist before. For instance, an orchestration might take input from a CRM system, process it through a custom analytics service, and then use a notification service to send tailored updates to users. The resulting new service, which might offer personalized customer insights, is an entity in its own right, distinct from its constituent services.

This process of blending multiple services to forge a new, unified service is why orchestrations are often referred to as 'composite services.' They are particularly versatile, finding use within integrations or as standalone entities in backend applications. The defining characteristic of an orchestration is this very act of composition – bringing together disparate services to create a cohesive, new service offering.



Backends

In the context of software architecture, 'Backends' refer to services that are specifically developed to support front-end applications, such as mobile, desktop, or web apps. The unique aspect of backend services is that they are typically tailored for a particular front-end, without a primary focus on being reusable by other developers in future projects.

The main purpose of these backend services is to offload complex logic and processing from the front-end. This separation of concerns not only makes the front-end application more streamlined and easier to manage but also contributes significantly to the overall system's manageability and security. By shifting intensive tasks to the backend, developers can ensure better performance, enhance data security, and improve the observability of the system's operations.

For instance, in a web application, the backend might handle data processing, user authentication, and server-side logic, thereby allowing the front-end to focus solely on presenting a responsive, user-friendly interface. This division ensures that the complex, behind-the-scenes operations are handled efficiently, paving the way for a smooth and secure user experience.

We have not included a separate diagram for backends in this paper, as their structure generally adheres to the server function pattern outlined in the primitives section. Typically, a backend comprises multiple server functions, each designed to abstract and process data from various backend systems. The culmination of these functions is a tailored data model, ideally suited to meet the specific needs of the client application or developer.

This pattern of leveraging server functions for backend development is not limited to just internal client applications. It also extends to the creation of specialized APIs and services geared towards partners. Such an approach is instrumental in facilitating strategic, bespoke integrations. By adopting this versatile pattern, developers can craft solutions that not only meet the immediate requirements of a client application but also pave the way for valuable partnerships and unique integration opportunities.

Key Benefits

Having defined the primitives and composites in Poly's architecture, we now turn to the key technical benefits of this approach and their corresponding business advantages. The adoption of serverless microservices in developing integrations, orchestrations, and backends brings forth several significant benefits: **Implicit Scaling, Infrastructure Utilization, Iterative Development, and Code Maintenance and Reuse**. Each of these plays a crucial role in enhancing the efficiency and effectiveness of our system. We will refer to integrations, orchestrations and backend as "services" or "server functions" in this section for simplicity.

Implicit Scaling

The first major benefit, Implicit Scaling, is inherently tied to the nature of serverless functions, particularly when services are run in a serverless infrastructure like KNative. This setup allows individual server functions to scale up or down autonomously in response to demand, which is crucial for handling varying loads and traffic spikes. For developers, this means a significant reduction in the complexities traditionally associated with managing service scalability, such as load balancing and traffic management.

Moreover, from an operational perspective, the serverless model simplifies the DevOps team's responsibilities. They can pre-approve the infrastructure setup without needing to be constantly involved in capacity planning for each new service. **This translates to faster deployment times, as there is no need for provisioning infrastructure** – it's already available and adaptable to the service's needs. Integrations and services can be deployed much faster, bringing agility and efficiency to the development process.

Infrastructure Utilization

The second key benefit is the significant improvement in infrastructure utilization. This is achieved by reducing idle time in services and minimizing the active container footprint – a crucial factor in cloud computing efficiency. In most scenarios, integrations are not constantly active; they are often idle, waiting for specific events or triggers. It is relatively rare for integrations to operate under a sustained high load for a long time.

Traditional models often require services to be ready for peak capacity at all times, leading to a substantial waste of resources due to idle services. However, with Poly's serverless approach, services that perform batch jobs or can tolerate brief wake-up times (like 5-8 seconds) are allowed to 'sleep' when not in use. This minimizes idle resources and ensures that only the necessary services are active at any given time.

Furthermore, the intrinsic ability of the system to scale implicitly allows for a more precise approximation to the actual load, keeping the unit of compute smaller and more efficient. This means that even the services which have to idea due to their latency sensitivity can be configured to use a smaller footprint. **These aspects of serverless architecture optimize resource usage and also translate into a significant reduction in compute infrastructure costs for Poly customers**. By smartly managing resource allocation, customers can enjoy a more cost-effective and environmentally friendly cloud computing experience.

Iterative Development

The third major benefit within Poly's ecosystem is the facilitation of Iterative Development, owing to each service or server function having its own development lifecycle (SFLC). This distinct lifecycle for each service means that smaller components can be developed and deployed to production much more rapidly than in traditional models. **Such granularity leads to a significant advantage: the parallelization of development.** It allows multiple developers to work simultaneously on different aspects of the same project.

In practical terms, an integration can be broken down into several independent flows, each being developed concurrently. Similarly, a composite service can be divided into multiple client functions, each with its own interface. These functions can be developed in isolation and later integrated into a higher-order server function. This modularity also enables front-end developers to begin building against completed server functions while backend developers continue working on others.

A crucial aspect of this approach is the clear definition and documentation of interfaces between these small, independent units, a process that Poly manages implicitly during deployment. Moreover, Poly's architecture supports the use of multiple programming languages for server functions. This diversity allows different functions, developed in their native languages, to be utilized in conjunction with one another. SDKs generated on top of each server function deployed to Poly further enhance this interoperability, providing a seamless integration experience across diverse programming environments.

Code Maintenance and Reuse

The final major benefit within Poly's framework is the enhanced code maintenance and reuse made possible by systematic cataloging of services. This cataloging facilitates easier identification and reuse of services. When those services require updates or evolution, the scope of retesting is substantially reduced. Maintaining interfaces or ensuring backward compatibility allows independent microservices to be updated without affecting others. This approach not only simplifies code maintenance but also accelerates the evolution of services.

Poly aids in reuse in several key ways. Firstly, it maintains a comprehensive catalog of all API, Server, and Client functions. This catalog serves as a foundation for various discovery methods, including search, prompt, and browse functionalities. Looking ahead, we anticipate integrating AI recommendations, similar to services like GitHub Copilot, to further streamline this process.

A significant innovation in Poly is the concept of client functions. These functions, capable of being utilized within many server functions, represent an efficient approach to code execution. This contrasts with the alternative of spinning up multiple server functions for minor tasks, which could result in excessive overhead from RPC invocations and container runtime management. **Client functions centralize code management, and when they are updated, these changes automatically propagate to all server functions that use them**. This means code can be managed in one place and disseminated effortlessly to all relevant integrations. Such an architecture marks a substantial leap forward in code maintenance efficiency and greatly enhances reusability.

Conclusion

This paper outlines the compelling advantages of leveraging serverless architecture for developing integrations, orchestrations, and backends. Poly harnesses this innovative approach, leading to significant enhancements in developer productivity. This productivity gain is achieved through simplified capacity planning, enabling parallel development, and fostering more efficient code maintenance and reuse. Beyond these technical benefits, our approach also translates into more tangible business advantages - notably lower development and infrastructure costs and faster delivery times for our Poly customers.

Our choice of this architectural paradigm was deliberate and aimed at capitalizing on the latest technological advancements. We have developed a comprehensive framework within Poly to make these benefits a reality for enterprise integration development. This framework is not just about meeting today's needs but is a testament to our commitment to innovation and future readiness.

If Poly's approach resonates with your enterprise's vision or if you are curious about how it can specifically benefit your organization, we encourage you to reach out for a more detailed discussion at info@polyapi.io.

Thank you for your time and interest.

Darko Vukovic
CEO & Founder of PolyAPI