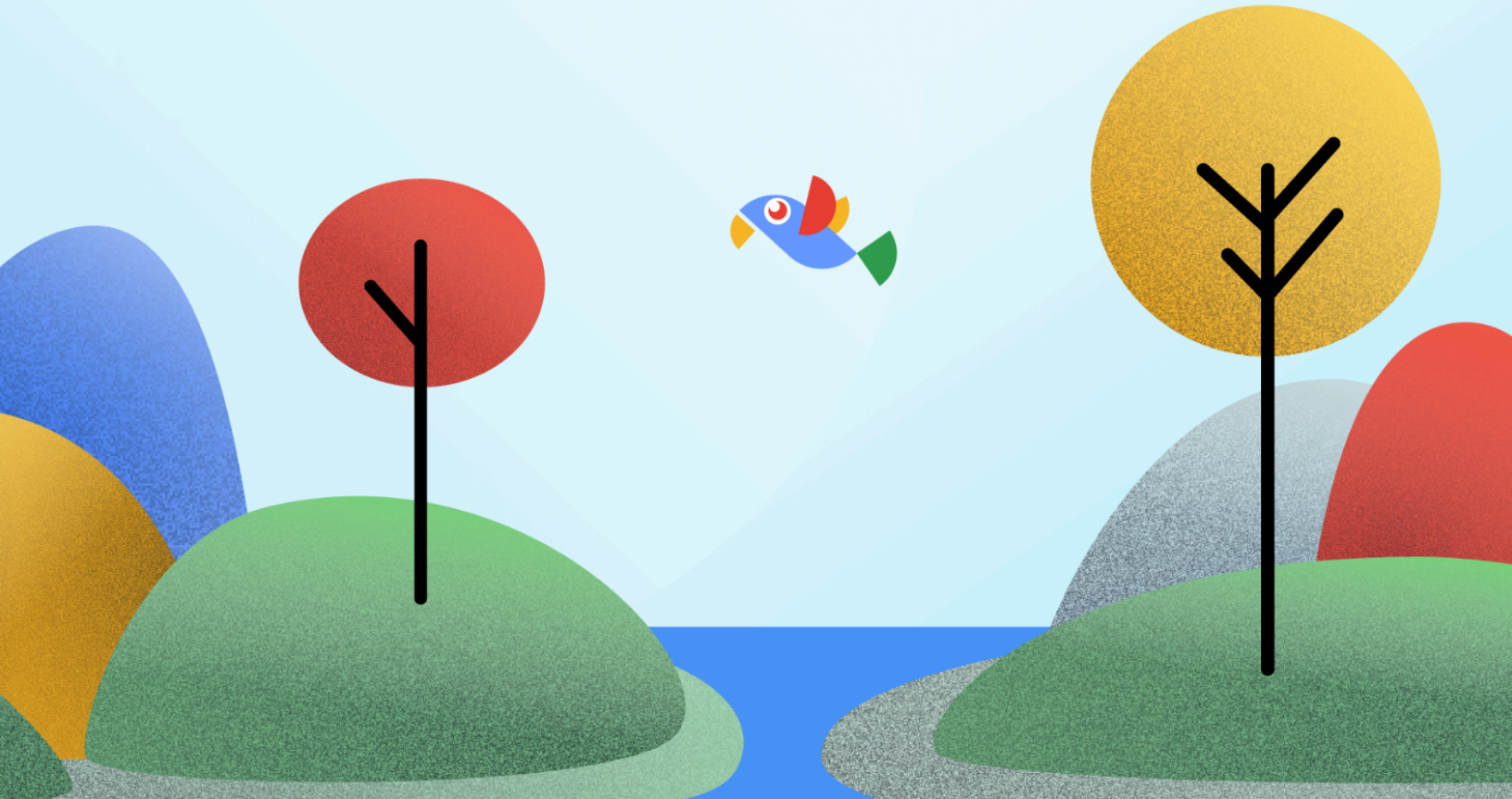
PolyAPI

Multi-Tenant Integrations

The Right Way

Darko Vukovic • Founder & CEO



PolyAPI

PolyAPI

Why Multi-Tenant Integration Matters Now

SaaS platforms today face growing pressure to offer integrations that are either pre-built or easily configurable on demand. Customers expect their platforms to connect seamlessly with their existing systems, no matter how varied or complex those systems may be.

The core challenge is that every customer operates a different stack. Even when they use systems in the same category, such as CRMs or ERPs, they configure them differently. Custom objects, fields, and business processes make every integration unique. Some customers go further and rely on proprietary systems that require bespoke logic.

Most teams respond by duplicating integration flows for each customer. This leads to a sprawl of redundant code, higher compute usage, and increased maintenance effort. Mappings are often embedded deep in the implementation, accessible only to developers, when they should be externalized so operations teams or customers can manage them.

Security adds another layer of risk. Credentials are commonly stored in shared vaults without tenant-level isolation. This prevents customers from managing their own credentials and introduces the possibility of cross-tenant data leakage if identifiers are misrouted.

Adding new systems or supporting new variants often requires re-testing and redeploying flows. This slows down delivery and maintenance becomes a drag on engineering.

The business result is a familiar pattern. A few stable integrations that work well, surrounded by a long list of supposed integrations that fall apart under scrutiny. We have all seen the partnerships or integrations pages filled with logos. Impressive at first glance, but when put to the test, many of them don’t reflect reality. Some exist only as internal demos or proof-of-concepts that require substantial effort to activate.

SaaS companies are spending too much building and maintaining integrations and too much licensing tools that were never designed for this kind of scale or flexibility. A better approach is needed.

This paper outlines the core challenges with multi-tenant integration, the architectural outcomes a modern platform must support, and how Poly enables a better way through a secure, scalable, developer-centric model.

PolyAPI

What Makes Multi-Tenant Integrations Hard

At first glance, offering integrations across customers might seem straightforward "connect system A to system B, replicate for each tenant". In reality, multi-tenant integrations introduce deep complexity, fragmentation, and constant change. Here's why most platforms struggle:

1. One Category, Many Systems

Your customers may all need a CRM integration—but that doesn't mean they use the same CRM. One uses Salesforce, another HubSpot, a third Zoho, and a fourth something homegrown. These are fundamentally different systems with different capabilities and constraints, even though they serve the same purpose.

2. Different APIs, Different Worlds

Each system has its own API format, protocol, authentication flow, and onboarding experience. Some use REST, others GraphQL, SOAP, or even file-based batch uploads. Authentication may require OAuth, API keys, service accounts, or custom header signing. Credential rotation policies, rate limits, access scopes, and audit requirements all vary widely. Even the documentation quality and developer experience are inconsistent across vendors.

3. Same Vendor, Different Reality

Even when customers use the same system, such as Salesforce or Oracle, no two instances are alike. You'll encounter different API versions, different instance URLs, regional or multi-org setups, varying product SKUs, and customer-specific feature sets. Two calls to the same endpoint can return structurally different data—or fail entirely due to SKU limitations or permission scopes.

4. Business Logic Is Never the Same

Most enterprise systems are highly configurable. Business rules, workflows, approval processes, and validations differ from one customer to another. This means the same API call might trigger different downstream behaviors or return different data structures,

depending on how the system is set up. Integrations that assume consistent behavior often break in subtle, difficult-to-diagnose ways.

5. Custom Objects Everywhere

Many enterprise systems support deep customization. Customers add fields, rename standard objects, or create entirely new custom objects. Some platforms operate almost entirely on custom objects by design. This means any robust integration must support dynamic object discovery, mapping, and behavior—there's no universal schema to work from.

6. Everything Is Always Changing

Fields are added. Credentials are rotated. APIs are deprecated. New services are turned on or off. Usage patterns evolve. Customers migrate versions or systems. You are building on constantly shifting ground. Without guardrails in place, small changes lead to broken flows, failed authentications, or silent data loss.

7. Customer Expectations Are Rising

Customers expect integrations to *just work*, and they want changes made quickly when things shift. They assume your platform can support their use case without requiring custom code or weeks of back-and-forth with engineering. Failing to meet these expectations not only slows implementation—it erodes trust.

These challenges are not edge cases. They are the norm. Supporting multi-tenant integrations at scale requires an architecture that treats variability, customization, and change not as exceptions, but as core design constraints. The next section outlines the capabilities a modern integration platform must have to meet these demands.

PolyAPI

What the Right Integration Platform Must Do

Supporting multi-tenant integrations at scale is not just a matter of running the same code for different customers. It requires architectural capabilities and operational guardrails that address customization, security, maintainability, and speed. A modern integration platform must meet the following core requirements:

1. Multi-Tenant Awareness at the Core

At the heart of a scalable integration platform is tenant context. Every flow should be tenant-aware and should automatically receive and extract a tenant identifier that drives logic, configuration, and credential resolution. This tenant ID should never be hardcoded, passed manually, or guessed. It should be securely injected and enforced at runtime.

2. Credential Isolation and Secure Storage

Credentials must be stored in a secure vault with strict tenant-level isolation. The platform should support secure *reference* to credentials during development and runtime. Secrets should not be available to the runtime code, meaning that they are not visible to developers and cannot be logged. Tenant ID injection attacks must be prevented by design, ensuring a rogue or misrouted request cannot cross boundaries.

3. Dynamic Configuration and Mapping

Each tenant requires its own mapping logic. Sometimes the mapping is subtly different, sometimes entirely custom. A modern platform must support per-tenant mappings, allow for updates without code changes, and expose mapping management via API for automation and integration to the end user SaaS App. This ensures that customers, operations teams, or solution engineers can tweak behaviors without involving developers or triggering new deployments.

4. No-Code Tenant Onboarding

Adding a new tenant should never require a new flow, redeployment, or custom code. Flows must be designed to generalize across tenants, driven by configuration, not duplication. The system should scale from one tenant to thousands without architectural changes.

PolyAPI

5. Modular Orchestration per Tenant (When Needed)

While most tenants can share flows with different configs, some will need custom business logic. The right platform enables tenant-specific overrides or orchestrations without disrupting the core flow, and without creating a maintenance nightmare.

6. Developer Velocity, Powered by SDKs

Out-of-the-box support for popular systems to enable direct integration development without spending countless hours researching APIs, backed by typed SDKs in popular languages, allows developers to build without reading obscure docs or wrestling with third-party quirks.

7. Bring Your Own System, Easily

Supporting new systems is inevitable. The platform must allow onboarding new APIs and schemas, including custom objects or models, without refactoring existing flows. Adding a new CRM integration, for example, should not require retesting every existing Salesforce flow. Object and system level isolation is essential.

8. Operational Excellence Built In

Troubleshooting must be fast and accessible. Rich logs, traceable executions, and real-time alerting allow the platform and customer teams to understand issues immediately. Alerts should be routed not only to internal ops teams but to affected customers—ideally through the SaaS platform itself. Visibility builds trust.

9. A Codebase Built for Scale

Code should be modular and atomic. New engineers must be able to onboard quickly, take ownership of flows, and make safe changes without fear of breaking everything. A platform that supports reusable functions, isolated logic units, and composability sets the foundation for long-term agility.

These are not wishlist features—they are essential requirements for any team serious about multi-tenant integration. Anything less leads to brittle flows, bloated codebases, and unsustainable costs. In the next section, we'll show how Poly delivers on this vision through a developer-first, security-forward, and operations-friendly architecture.

Poly Makes Multi-Tenant Integration Easy

Poly is purpose-built to solve the complexity of multi-tenant integrations. It provides a modern runtime, a unified modeling layer, built-in security, and a high-performance developer experience that makes onboarding, extending, and operating integrations dramatically simpler. Here's how it works:

1. Server Functions: Tenant-Aware by Design

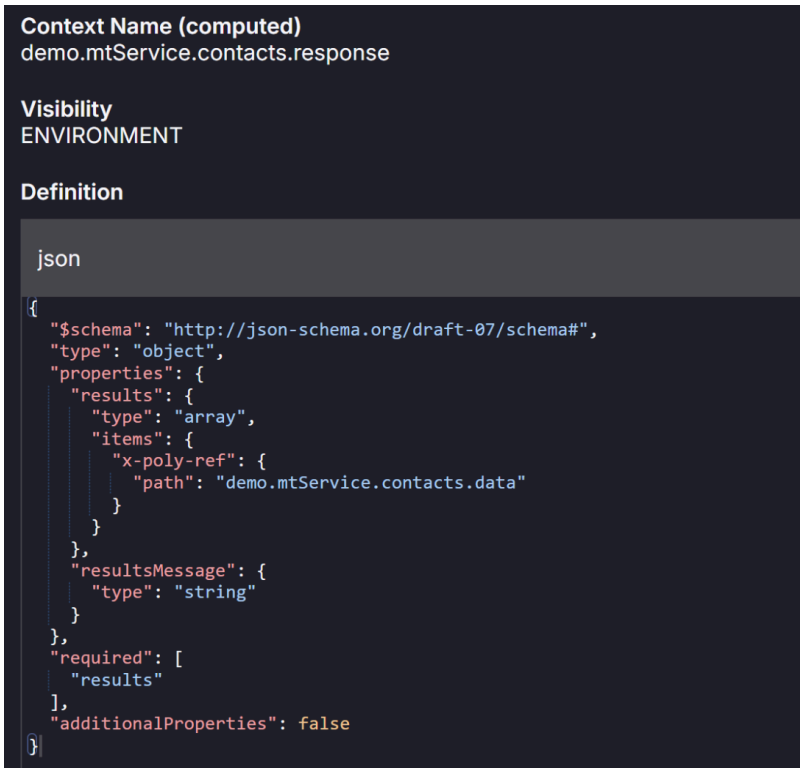
Poly server functions are the core execution units of your integrations. Built on a Kubernetes-native runtime, they can run in our PaaS or be self-hosted in your own cluster. Each function accepts a tenant identifier—passed via URL, header, or payload—which scopes logic dynamically without duplicating code. Server functions are automatically generated into the SDK, making them easy to invoke programmatically. You can call them through the PolyAPI-generated SDK or expose them through a custom-designed API. The example below shows it as a simple argument as an illustration with a deploy command above the signature.

```
//npx poly function add getContacts MTDemo\demoMTServerFx.ts --context=demo.mtService.services --server
async function getContacts(tenant: string): Promise<schemas.Demo.MtService.Contacts.Response> {
```

2. Schemas: Strong Typing, Centralized Models

Schemas in Poly define structured object models that make coding integrations simple and safe. They're centrally managed through the UI or API and can be imported from OpenAPI specs. These models ensure consistency across functions and enable rapid onboarding by documenting data shape in a shared, accessible format.

Below is a view of the Schema in the PolyAPI Management UI:

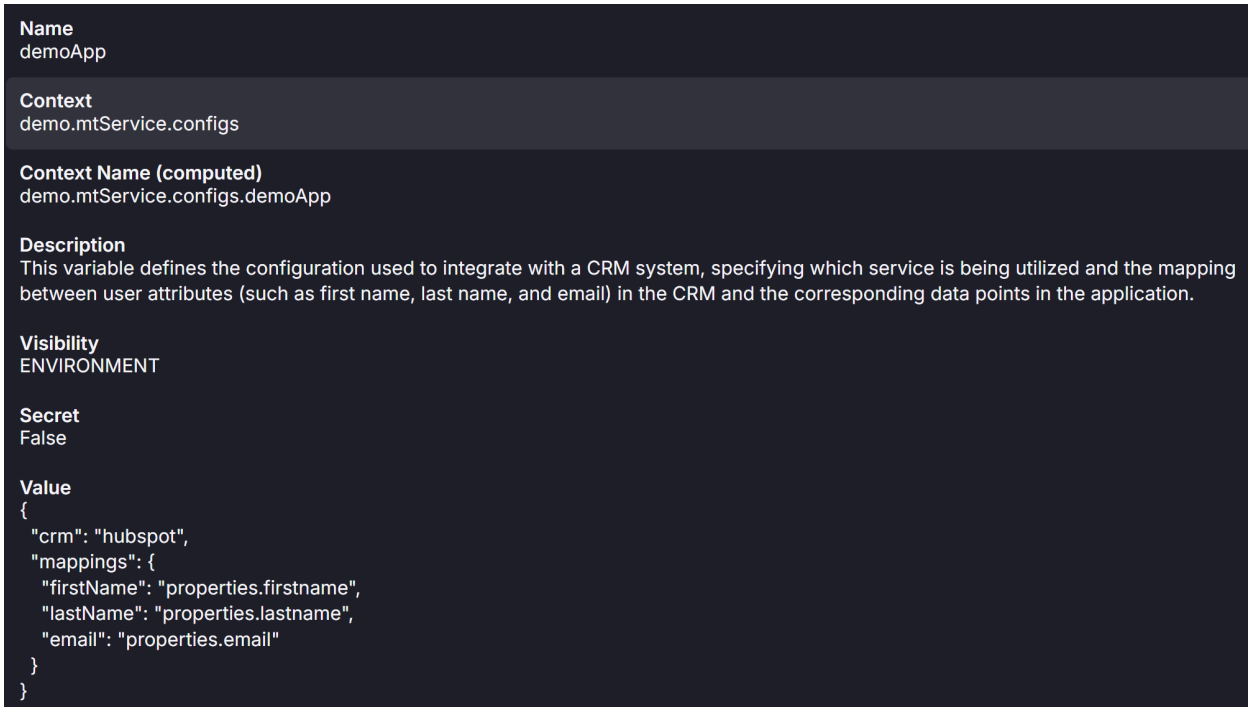


And an example of the equivalent generated schema in the generated TypeScrit SDK:

```typescript
declare namespace schemas {
  interface DemoMtServiceContacts {
    data: Demo.MtService.Contacts.Data;
    response: Demo.MtService.Contacts.Response;
  }
  namespace Demo {
    namespace MtService {
      namespace Contacts {
        type Data = {
          firstName?: string;
          lastName?: string;
          email?: string;
        };
        type Response = {
          results: Demo.MtService.Contacts.Data[];
          resultsMessage?: string;
        };
```

3. Variables: Configuration and Credential Management

Poly supports three types of variables: none (plain text), obscured (masked), and secret (unreadable at runtime). This is where per-tenant configurations, credentials, and mappings are stored. Access to these variables is driven by the tenant context, enabling strict isolation and runtime configuration without hardcoding anything into your logic. The screenshot below shows a simple configuration example stored in a tenant specific variable. In this case our demo tenant ID is “demoApp”.



Note that this example illustrates a configuration that this tenant uses “hubspot” (should have been all caps, sorry developers) as the CRM and that the specific mapping structure looks for just three attributes (to keep the example simple).

4. Dynamic References: Keep Code Tight and Flexible

In Poly, variables and functions can be accessed dynamically through reference paths. This makes it possible to add new tenants, mappings, and even systems without modifying the integration logic. These references keep the code clean and maintainable while allowing it to adapt to new configurations seamlessly.

Here is an example of pulling the configurations dynamically. If the tenant identifier is “demoApp” the tenant configs will be pulled from the variable shown above in section 3.

```
const tenantConfigs = await vari.demo.mtService.configs[`${tenant}`]?.get();
```

Since the configurations for “demoApp” (section 3) are to use a “crm” with the value of “hubspot” this will dynamically invoke a function for hubspot. This approach allows the same line of code to invoke many different functions depending on tenant specific configurations.

```
const contacts = await poly.demo.mtService.utilities[`${tenantConfigs.crm}`].getContacts(
  tenant,
  tenantConfigs.mappings,
  query
);
```

PolyAPI

5. Full Power of TypeScript or Python

Your server functions are written in real TypeScript or Python, not some proprietary scripting DSL. This gives you the full expressive power of your preferred language, including support for libraries, control structures, and complex logic. That flexibility allows Poly to handle edge cases and intricate business logic with ease. Here is an example of validation to ensure the tenant configs exist and that the crm configuration is supported:

```
// Check for a valid tenant entry
if (tenantConfigs === undefined) {
  const errorMessage = "Could not find a valid configuration for tenant " + tenant;
  console.error(errorMessage);
  return {
    results: [],
    resultsMessage: errorMessage,
  };
}

// Check if this tenant is configured to use a supported system
const supportedCRMsConfig = await vari.demo.mtService.configs.supportedCRMs.get();

// Dynamically find the CRM configuration based on tenantConfigs.crm
const crmConfig = supportedCRMsConfig.find((config) => config[tenantConfigs.crm]);
const query = crmConfig ? crmConfig[tenantConfigs.crm]?.query : null;

if (!crmConfig) {
  const errorMessage = `${tenantConfigs.crm} is not a supported system, or its configuration
  is missing. Currently supported systems are: ${JSON.stringify(supportedCRMsConfig.map(Object.keys))}`;
  console.error(errorMessage);
  return {
    results: [],
    resultsMessage: errorMessage,
  };
}
```

6. Local Testing: Native Debugging, No Guesswork

Because Poly uses native runtimes, you can test your server functions locally in your IDE, use real debuggers, and run unit tests with your favorite tools. This means faster development, faster iteration, and far more confidence in your code before deployment.

7. Client Functions: Dynamic Extensibility Without Redeploys

Poly automatically generates an SDK from your catalog of functions. These *client functions* can be invoked by your server functions without requiring any redeployment. This allows teams to add new capabilities, call other flows, and extend integrations dynamically, safely and at runtime, without touching the parent server function's code.

First, an example of a client function to abstract the code for a system-object-operation triple, in this case get-contacts-hubspot. One of these functions, conforming to the same naming convention would need to be added to the system each time a new object, operation or system is added. This code is very atomic and therefore easy to maintain due to the schemas, variables, out of box API function, and mapping utility function.

```
// npx poly function add getContacts MTDemo\hubspotGetContactsCFX.ts --context="demo.mtService.utilities.hubspot" --client
async function getContacts(
  tenant: string,
  mappings: object,
  query?: string
): Promise<schemas.Demo.MtService.Contacts.Data[]> {
  const contactRes = await poly.OOB.hubspot.contacts.get(
    vari.credentials.hubspot[`${tenant}`].accessToken.inject(),
    query
  );
  const contacts: schemas.Demo.MtService.Contacts.Data[] = contactRes.data.results.map(
    (contact: Oob.Hubspot.Contacts.Get.Result) => poly.demo.mtService.utilities.generic.mapObject(contact, mappings)
  );
  return contacts;
}
```

Another common use is for generic utility functions, like mapping. These functions are reusable across tenants, systems, and flows—compounding productivity gains over time and showcasing the power of native development in Poly.

```
// npx poly function add mapObject MTDemo\genericMappingCFX.ts --context="demo.mtService.utilities.generic" --client
function mapObject(contact: any, mappings: Record<string, string>): object {
  const mappedObject: Record<string, any> = {};
  for (const [key, path] of Object.entries(mappings)) {
    mappedObject[key] = resolvePath(contact, path) || ""; // Resolve the path or provide a default value
  }
  return mappedObject;
}

// A helper function to resolve deeply nested paths
function resolvePath(obj: any, path: string): any {
  return path.split(".").reduce((acc, key) => {
    if (!acc) return undefined; // If the path cannot be resolved, return undefined early
    return acc[key];
  }, obj);
}
```

8. Out-of-the-Box Functions: Accelerated Development with Modeled Responses

Poly includes a growing catalog of out-of-the-box functions for popular systems. These functions expose a well-defined interface and return modeled response types, allowing developers to build quickly without needing to learn external APIs. While today's functions can be easily replicated and re-trained to adapt per tenant, upcoming support for tenant-specific modeling will allow even more dynamic and personalized behavior without altering the function's core logic.

Here is an example of an out-of-the-box (OOB) function used in this example:

```
const contactRes: ApiFunctionResponse<Oob.Hubspot.Contacts.Get.ReturnType>
const contactRes = await poly.OOB.hubspot.contacts.get(
  vari.credentials.hubspot[`${tenant}`].accessToken.inject(),
  query
);
```

Note that the function has a well defined response type and signature which defines the needed arguments (with descriptions) and their respective types. This information comes from generated type definition files that look like this (cropped for space).

```
/**
 * Retrieve a list of contacts from HubSpot CRM. The response includes contact details and pagination links. For more details: ht
 * @param accessToken This is a string that represents the access token used for authentication. It is used to verify the user's
 * @param queryParams This is a string that represents the query parameters to filter and customize the response. It can include
 *
 * Function ID: cd96993b-eab0-4a0a-bfb2-999d5c121377
 */
get(accessToken: string, queryParams?: string): Promise<ApiFunctionResponse<Oob.Hubspot.Contacts.Get.ReturnType>>;
```

We already provide thousands of out of the box functions, but our ambitions are much greater. We plan to provide hundreds of thousands of functions and to leverage algorithms to detect runtime discrepancies and to proactively maintain the schemas at runtime.

Together, these capabilities form the foundation of Poly's approach to scalable, secure, and cost-effective multi-tenant integration. In the next section, we'll wrap up with some parting thoughts on how this architecture transforms integration delivery and operations.

Conclusion

This paper has explored the challenges that make multi-tenant integrations uniquely difficult for SaaS platforms and the architectural requirements any serious solution must meet. We've broken down the technical realities—ranging from inconsistent APIs and custom object models to tenant-specific business logic and evolving credentials—and shown how PolyAPI addresses these issues through a modern, developer-centric architecture.

From tenant-aware server functions to dynamic credential management, modular mapping, and client functions that scale without redeployments, PolyAPI is built for teams who need to deliver integrations faster, operate more efficiently, and adapt to constant change—without compromising security or maintainability.

Whether you're just starting to scale your integration catalog or trying to replace fragile, duplicated flows with something more robust, PolyAPI offers a better way to design, build, and run multi-tenant integrations. Our goal is to make integration a strength, not a source of drag, cost, or technical debt.

If the ideas outlined here align with your challenges or vision, we'd love to connect. Our team is here to help you move faster, scale smarter, and stay focused on your core product.

Please feel free to reach out at hello@polyapi.io to start the conversation.

Thank you for your time and consideration.
Darko Vukovic
CEO & Founder, PolyAPI

PolyAPI

About the Author

Darko Vukovic
CEO & Founder, PolyAPI

Darko Vukovic is the Founder and CEO of **PolyAPI**, a powerful platform designed for **integration**, **orchestration**, and **microservices** development in enterprise environments. Darko has a deep background in **API management**, **composite application development**, and **integration development**, having built enterprise platforms that enable organizations to streamline their operations and accelerate innovation.

Throughout his career, Darko has worked extensively with major platforms like **MuleSoft**, **Oracle**, and **Google Apigee**, driving the development of large-scale solutions for **hospitality**, **eCommerce**, and **technology** sectors. His hands-on experience in delivering robust, scalable enterprise platforms has helped businesses tackle complex integration challenges and unlock new opportunities for growth.

As a technology leader, Darko is passionate about creating solutions that empower enterprises to scale their systems with speed and efficiency. Under his leadership, PolyAPI is becoming a go-to partner for enterprises seeking to modernize their infrastructure and gain a competitive edge through seamless integration and microservices development.