O'REILLY®

Compliments of solo

Omni-Directional API Management for Platform Engineering

Modern API Management in the Cloud and AI World

Christian Posta

REPORT

SOLO.IO

Architecture modernization requires modern API management

Deploy Gloo. Enterprise-grade, cloud-native API connectivity solutions designed for the platform engineer.

Find out more about API management

www.solo.io

Omni-Directional API Management for Platform Engineering

Modern API Management in the Cloud and AI World

Christian Posta

O'REILLY®

Omni-Directional API Management for Platform Engineering

by Christian Posta

Copyright © 2025 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Development Editor: Gary O'Brien
Production Editor: Aleeya Rahman
Copyeditor: Penelope Perkins
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

October 2024: First Edition

Revision History for the First Edition

2024-10-10: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Omni-Directional API Management for Platform Engineering*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Solo.io. See our statement of editorial independence.

978-1-098-17797-3

[LSI]

Table of Contents

1. The Need for Modern API Management. 1
Improving Challenges Around API Delivery 1
Emerging Use Cases Around AI and LLM Usage 3
Existing API Management Is Outdated 3
Modern API Management 4
Platform Engineering to the Rescue 6

2. Foundations Are Important. 9
Foundation of a Modern Gateway 10
Architecture for Flexibility, Automation, Decentralization, and Delegation 12
Providing API Management for an Internal Developer Platform 15
Microgateway Architecture for Tenancy 16
Bringing API Management to Traffic in All Directions 17

3. Omni-Directional API Management. 19
Using a Service Mesh for East-West API Traffic 20
Enabling Omni-Directional API Management with Istio Ambient 25
Bringing It All Together 28

4. Tying Omni-Directional API Management into Your Platform. 29
Building Golden Paths 31
Adopting Omni-Directional API Management in Your Platform 33
Improving Self-Service, Resilience, and Innovation 34

CHAPTER 1
The Need for Modern API Management

Time is a scarce resource. The amount of time we save in our daily lives compared to years ago is unbelievable. Remember when we needed to make a phone call to book air travel? Or call a taxi, sit around, and hope it shows up? The amount of self-service and improved experience we get from technology (funnily enough, from our phones) allows us to get more done in less time and focus on things that are more valuable.

Well, ideally. Damn social media!

T-Mobile did a deep investigation into why its API teams had trouble delivering software quickly and safely. They found that 25% of API developers' time was wasted on nonfunctional overhead and requirements (i.e., filing tickets and waiting). Additionally, 75% of the problems they faced in production were due to networking configuration changes. The direct translation from time to money is clear. We need to improve our API delivery. If we could save this wasted money and time, we could invest in something more valuable, like new features.

Improving Challenges Around API Delivery

APIs underpin the digital experiences an organization delivers to its customers. Through a website, mobile app, or partner integrations, adding new functionality involves making changes or delivering

new APIs. The speed and safety of API delivery is a serious business differentiator. Bottlenecks, inefficiencies, and outages cost a business in opportunity and real money. A modern API management system needs to enable self-service and tenancy, while reducing the blast radius of changes to help an organization achieve its business goals.

Unfortunately, API developers sometimes have to wait two or more weeks to get their changes reviewed, approved, and implemented. Often the API developers need to open tickets to get this work done, meaning another team must complete some task, such as changing the routing rules on an enterprise load balancer or updating an API management system, before the developers can continue their own work. The outdated technologies used in these API management systems don't lend themselves to automation or tenancy, so the work is often done manually or by pointing and clicking on some proprietary UI. This is slow and increases the risk of errors.

The DevOps movement focuses on some of these problems, reducing the siloed nature of our IT organizations between Dev and Ops and improving collaboration. Focusing on automation, collaboration, and shifting left are generally good practices, but in some cases went too far—for example, pushing application teams to own and manage their own Kubernetes platforms and distributions. Placing the burden on application teams to own their infrastructure, including observability, security, CI/CD (continuous integration, continuous delivery), etc., overloads teams that don't have the expertise to own this infrastructure. Shifting too far to the left also causes other problems, such as compliance, governance, and sprawl-related issues.

The pendulum is swinging back these days in the form of "platform engineering" and hopefully will arrive at a happy medium: reducing silos, improving tenancy, and giving teams autonomy with access to what they need. For example, we see platform engineering teams providing the tools, automation, and self-service workflows for API developers to deploy new APIs or changes much faster. These self-service capabilities also align with compliance, security, and reliability goals.

Emerging Use Cases Around AI and LLM Usage

Artificial intelligence (AI) use cases with large language models (LLMs) are a new class of API usage in enterprise applications and are getting widespread adoption. LLM providers already have API self-service, so why would an enterprise worry about making these APIs available internally? What would they need to manage?

Today, any developer with a credit card can go to an LLM provider and sign up to call its APIs. For example, a developer can go to the OpenAI website, get an account, access tokens, and secret keys, and start calling LLM APIs. How many developers are doing this? How many access tokens and secret keys are littered around production? Are they secured correctly or just floating about haphazardly? What information is sent to the LLMs? Is it approved by IT and does it meet compliance requirements? Are some teams overspending through unintended usage or self-DoS (denial-of-service) type mistakes? Is anyone monitoring or measuring any of this?

External API and LLM API usage creates spending, compliance, and security risks for an enterprise organization. Modern API management, which observes, controls, and secures egress traffic, is necessary.

Existing API Management Is Outdated

Full lifecycle API management suites do not fit in today's modern platform engineering approach. The legacy API management vendors sell a "full lifecycle, do everything" solution that forces users into the vendors' opinionated view of the world with tools that are not best of breed.

These older solutions are built on outdated technologies that are inefficient, resource intensive, and make it difficult to support tenancy and self-service. For example, it's difficult to automate a proprietary UI in a CI/CD pipeline. Or the routing and load balancing features are very coarse grained and rely on sending traffic to external load balancers, which themselves need to be configured out of band.

Organizationally, these solutions create more silos because they are complex to operate, configure, and maintain. You need a dedicated team to manage them. To make changes, you need to create tickets for the "API Management" team. This wastes time.

These legacy solutions may have added point features to try to fit a marketing message that better aligns with the needs of modern use cases. Unfortunately, it's not possible to take outdated API management tooling and fit it into modern workflows.

Modern API Management

Recently, the trend of "unbundling" your API management suite, as coined by Erik Wilde, is in full swing. API management features such as rate limiting, usage quotas, metrics/analytics, developer portals, OAuth/OIDC (OpenID Connect), and API key security are common across all API management products (outdated or otherwise). In fact, if you're looking for a modern API management product, you won't find it by comparing features because for the most part, they're almost all the same.

What really matters for modern API management use cases is how the nonfunctional requirements fit with the rest of your workflows and internal developer platforms. Modern API management is centered around the four nonfunctional tenets (see Figure 1-1).

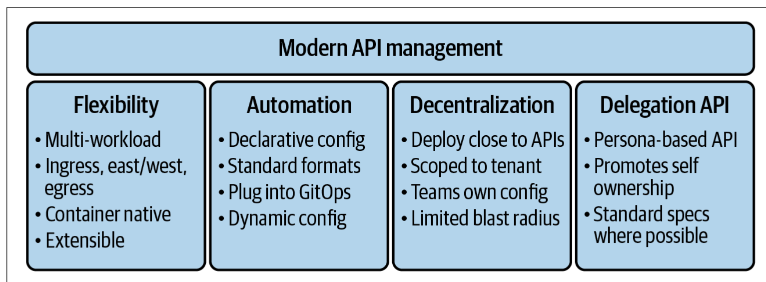

Figure 1-1. The four nonfunctional tenets of modern API management.

Flexibility

APIs are meant to encapsulate functionality and standardize how that functionality is invoked. Where they are deployed should not matter—Windows, Linux, mainframes, big servers, small servers, cloud servers, containers, Kubernetes, functions,

etc. A modern API management solution must be flexible enough to provide the features shown in Figure 1-1 wherever APIs may be deployed. It must also be flexible enough to be deployed as close to the APIs as possible to give more fine-grained control. A modern API management solution should apply its fine-grained policy control and capabilities to traffic in any direction: ingress, east-west, and egress. For more complex scenarios, a modern API management system should be extensible so you can add customizations as necessary to interoperate with a given environment.

Automation
Automation for a system like this is foundational. It's okay to have a UI dashboard, but the entire system should fit into a declarative configuration pipeline typically seen in GitOps workflows. Common configuration formats (rather than a proprietary, black box configuration) must also be used. Configuring the API management system should be dynamic and eventually consistent. Deploying large fleets of the API management system should be controlled and configured following this automated approach, and the mechanism for configuring the system should support updates to policy and infrastructure without dropping connections or impacting traffic.

Decentralization
Building on the tenet of flexibility, a modern API management system should be decentralized rather than tied to a specific cloud or on-premises technology. Traffic should not have to "hairpin" through some centralized system. Through decentralization, API teams should be able to own and configure their API infrastructure with strong tenancy. A common configuration control plane may be used to "federate" the API infrastructure to make it all behave as a single, common infrastructure. Federation allows failures to happen with a blast radius scoped to a specific tenant. A modern API management solution should use consistent technology and features throughout, not make trade-offs between centralized and decentralized deployments.

Delegation
Lastly, a modern API management system should support a *delegated API model*. Delegation means some parts of the API configuration are intended to be configured by platform users,

while other parts are configured by API developers. Separating the configuration of the API this way enables teams to work in parallel without dependencies on each other, which speeds up changes. With a delegated model, developers can configure routing, transformation, and other route specific capabilities. Delegation is key to tenancy, decentralization, and federation.

API developers should have the ability to pick the developer tools they like to define, test, and implement their APIs. They should be able to self-service deploy to an infrastructure that includes dashboards for visibility of their changes and how their APIs get used or behave. These tools should fit the way they want to work. What does this look like in an enterprise?

Platform Engineering to the Rescue

Solving these API management challenges with legacy solutions won't work. At the other extreme, allowing developers to write common API management functionality directly into their code is a recipe for disaster. Choosing libraries and frameworks, across multiple languages, to handle security, load balancing, rate limiting, and other tasks create technical debt and consistency issues. Who's got the time to learn all these new technologies? Will they just be point solutions for a developer's immediate pain without taking into account the organization? Will configuration and security issues crop up because of the lack of governance and oversight? How will this fit into the rest of the delivery pipeline or lifecycle?

For API developers to build and deliver APIs safely and securely at a rapid frequency, the platform team needs to enable flexible tool selection with self-service golden paths. A platform team can make common choices available, while allowing developers to plug in their own tools as needed.

Platform engineering teams are charged with improving developer experience and compliance, eliminating inefficiencies, and generally accelerating the delivery of business value. They select cloud technologies such as containers, Kubernetes, CI/CD, observability, and security tools that fit well together, can be automated, and improve the ability to deliver software.

Developers are finicky, however, and may push back on the toolchains forced on them for building APIs. For example, legacy API management systems have their own proprietary tools for defining API contracts, generating mocks, and tying it all into the rest of the lifecycle. A developer may find that tooling lacks certain features, is difficult to use, or doesn't support their development environment. Postman, for example, is a popular client-side tool for testing, designing, and debugging APIs; developers should have the option to pick this tool and tie it in with the rest of the API platform.

On the other hand, when it comes to the deployment lifecycle, including implementing security, cataloging APIs, gathering observability metrics, and implementing resilience features like failover or rate limiting, there should be automated golden paths for developers to use.

Developers should be able to obtain the necessary access to deploy APIs, make changes to routing, publish to a catalog, and so on via self-service mechanisms, not by filing tickets and sitting around and waiting. The platform team is responsible for creating the automation and tooling to enable self-service for the developers. See Figure 1-2 for how API management tools align with developers and platforms.

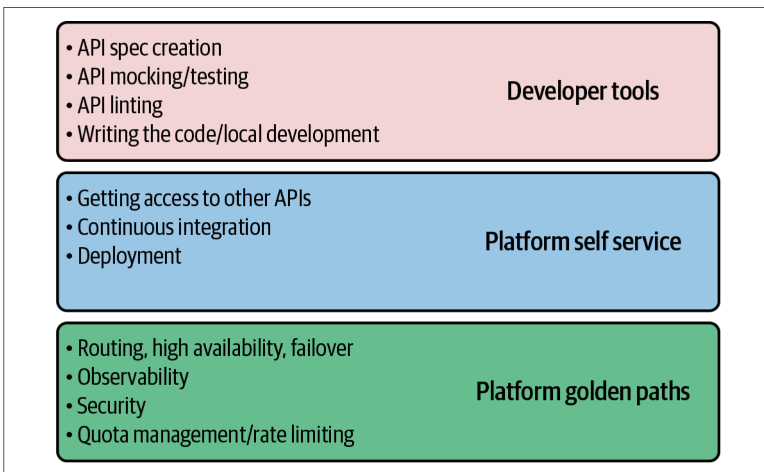

Figure 1-2. Where API development and management tools fit in a modern internal developer platform.

This report will dig into the foundations for platform teams to provide an omni-directional, federated API management system built on Cloud Native Computing Foundation (CNCF) technologies such as Envoy, Istio, and others. These tools and technologies, combined with architecture, will enable scale, self-service, tenancy, and improve security and compliance. This approach has been proven to shorten the time needed for API developers to get their changes into production safely.

CHAPTER 2

Foundations Are Important

When thinking about modernizing your infrastructure, investing time laying the right foundation is important. Not doing so can lead to very costly and time consuming repairs, which could jeopardize your modernization efforts. Similarly, large construction projects, like building a skyscraper, require strong foundations. For example, the Salesforce Tower in San Francisco, California, is 1,070 feet tall and must withstand strong winds and earthquakes. The foundation for this building took 18 hours to pour and is anchored deep into the bedrock 300 feet below the surface (see Figure 2-1). Conversely, another skyscraper in San Francisco did not implement a strong foundation. The Millenium Tower has its foundation directly on the soil and is not anchored into bedrock. The Millenium Tower has been sinking and leaning since its construction was completed and is now undergoing costly repairs to retrofit its foundation.

Just like in building materials, the foundational technologies of your platform matter. Kubernetes is the right foundation to build a modern cloud platform, but what's right for API management? Making the incorrect choice, or trying to cut corners, will end up like the sinking Millennium Tower in San Francisco.

Figure 2-1. San Francisco's solid bedrock layer can be good for building skyscrapers if the foundation is dug deeply enough and done correctly. The right foundation is important for the intended use case.

One of the most important pieces of modern API management is an API gateway. The API gateway plays a vital role in routing, policy enforcement, security, quality of service, and observability. Most API gateways on the market can perform these functions well, but only a few were built in the modern era and with modern usage in mind.

Foundation of a Modern Gateway

Envoy proxy is to API gateways what Kubernetes is to building developer platforms: a powerful foundation upon which to build for the modern era. Envoy is an open source project built to run as a highly performant HTTP2 and gRPC proxy and load balancer written in C++. It's flexible and can be run as an ingress/egress gateway, a microgateway, or even a sidecar proxy in a service mesh. For example, it powers the Google Front End (GFE) service, which handles all incoming traffic to Google Cloud. It can be used to build API gateways such as the open source projects Gloo Gateway and Ambassador's Emissary-Ingress. Other reverse proxies, such as NGINX, existed when Envoy hit the scene, so why is Envoy the right choice?

Envoy proxy provides the foundational pieces for what's needed in a modern gateway:

- Bidirectional, dynamically configurable API
- Service endpoint discovery and routing
- Active and passive health checking
- Multiple extensibility points
- Standard integration points

Envoy proxy was built to handle updates to its configuration dynamically. This may sound trivial, but it's crucial. Envoy pioneered the xDS APIs, which are bidirectional, streaming APIs between the proxy and a separate control plane. Envoy's endpoints, routes, security policies, observability, and so on can be dynamically and continuously configured over a large fleet from a centralized location without reload or downtime. In ephemeral infrastructure environments, like container platforms, this is critical.

A big reason for Envoy's open source community success and general adoption is that it's very extensible. Envoy was built on a "pipes and filter" architecture, where all of its functionality—from routing and load balancing to security and observability—is built on extensible filters. These filters can be chained together in various ways to get the desired behavior of a gateway. Envoy can also call out to other systems over standard protocols (ext auth, OIDC, OAuth, etc.) to implement security, enrichment, or observability (logging, metrics, etc.).

Envoy, out of the box, is not an API gateway. To get table-stakes API gateway features such as rate limiting, request/response transformation, or a developer portal, we need to extend Envoy with additional components and features.

Architecture for Flexibility, Automation, Decentralization, and Delegation

Envoy provides the foundation for an API gateway, but just like all good open source projects, it draws boundaries around what features it will implement or expand into. Since Envoy is built to fit many use cases (including an API gateway), it leaves the last mile features and expansions to end users or vendors by way of an extensibility model.

Projects like Gloo Gateway, Contour, and Emissary-Ingress extend Envoy to add the features needed for it to be an API gateway. For this report, I'll use Gloo Gateway, which is built on Envoy proxy, as an example of a modern API gateway. Gloo Gateway adds transformation, rate limiting, and more sophisticated security protocol handling, such as OAuth 2.0, to create a full-blown API gateway.

As outlined in Chapter 1, flexibility, automation, decentralization, and delegation are key tenets for a modern API management platform. The Gloo Gateway architecture and capabilities listed below achieve these tenets and give a better fit for an internal developer platform, as illustrated in Figure 2-2:

- Separation of control plane from data plane
- Declarative configuration
- Delegated configuration model
- Federation/tenancy model

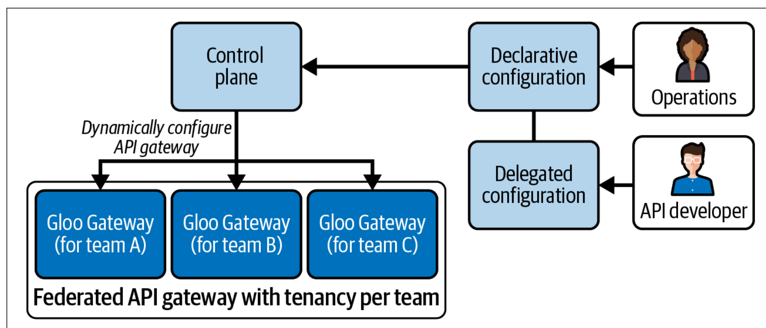

Figure 2-2. Separate control plane and data plane alongside a delegated API for driving configuration better fits a decentralized API management architecture.

Separation of Control Plane from Data Plane

Envoy proxy handles requests and connections and is considered the *data plane*. How does Envoy get its configuration? That's where the *control plane* comes into the picture. Getting infrastructure and user configurations to the data plane is foundational for enabling the type of flexibility and automation we need in our modern platforms. Envoy doesn't ship with a control plane out of the box. A number of Envoy control planes exist, and rolling your own is a very risky and expensive undertaking. Gloo Gateway implements a control plane for Envoy proxy and has been battle-tested at scale.

One important detail about the Envoy control plane is that it should be built and deployed *separately* from the data plane. The control plane often deals with sensitive data, connecting to a Kubernetes API server or other sensitive systems, and should not be colocated with the data plane. If the control plane is deployed separately, it can suffer an outage and the data plane can continue to process requests. Gloo Gateway is implemented with a separation of control plane and data plane. It even provides a way to persist the last known good configuration, which can help to overcome the ripple effects often seen in outages when teams blindly restart things in an effort to fix misbehaving infrastructure. This control plane/data plane separation is a major difference between Gloo and Emissary-Ingress.

Declarative Configuration

Gloo Gateway's configuration model is built on a *declarative configuration*. More specifically, it's built as *custom resources* (CRs) in Kubernetes using the Kubernetes Gateway API, a specification driven by the Kubernetes community to standardize traffic routing rules. Gloo Gateway implements the Kubernetes Gateway API and adds extensions to support complex rate limiting, security protocols, and other features not included in the specification. The fact that Gloo Gateway is built on a declarative configuration makes it easy to include the API gateway in a GitOps automation toolchain. Driving everything through Git allows versioning, auditing, reviewing, and controlled rollouts via something like Argo CD. It also allows you to treat gateway configuration and policy as code and not some separate, invisible infrastructure concern.

Although Gloo Gateway uses the Kubernetes Gateway API for its configuration, it is not tied to Kubernetes; it can run natively on VMs as well. Gloo Gateway can route to any network endpoint, including natively to AWS Lambda endpoints.

Delegated API Configuration

Gloo Gateway is the only API gateway implementation that provides a delegated API configuration layered on top of the Kubernetes Gateway API. This means some API objects are responsible for configuring certain parts of the API gateway functionality, while others can be delegated to API developer teams, so they can manage the specific configurations that are important to them. This provides the foundation for a self-service model and allows teams to configure their gateways without stepping on other teams' toes or requiring a central team to do it. The delegation API also allows centralized platform teams to configure the pieces they are concerned with, such as security, gateway health, and integration.

Federation/Tenancy Model

A big part of supporting self-service in an internal developer platform is tenancy, ideally with some sort of cell boundary with capabilities to deploy separate *microgateways* per domain. For Gloo Gateway, this means deploying API gateways per namespace across multiple Kubernetes clusters and configuring the gateways for failover.

For stronger tenancy guarantees, aligning to the cluster model or deploying multiple control planes to configure dedicated microgateways is an option. When you have multiple control planes for the microgateways, you start to get to a *federated API management* model with multiple independent gateways working together to provide the API management fabric.

Legacy API management vendors don't offer this kind of flexibility. They usually offer a single, monolithic, centralized gateway; if you want to use a microgateway, that implementation is just a less-featured gateway that handles a subset of functionality.

Providing API Management for an Internal Developer Platform

Macquarie Bank, a large, regulated financial institution in Australia, experienced an explosion in API traffic, leading to a desire to move from on prem to public cloud. The bank was also dissatisfied with its existing processes, which caused API developers to sit and wait weeks to make changes. To address issues with these processes, Macquarie implemented a solution to provide multitenant, self-service capabilities to its API developers.

As the company moved to microservices, it faced more complex needs for resilience, policy enforcement, and modern security protocols. It needed more fine-grained rate limiting, circuit breaking, distributed tracing, and authentication/authorization (JSON Web Token [JWT] handling, TLS) deployed closer to its applications and with more self-service functionality for API calls.

Macquarie Bank built its developer platform using Argo CD and similar tools, along with an infrastructure as code (IaC) philosophy on top of Kubernetes as depicted in Figure 2-3. The company ended up using Gloo Gateway to create microgateways for each team to achieve the necessary granularity of tenancy. All of this was abstracted away with a simple self-service UI, monitoring dashboards, GitOps, and Helm templates.

Macquarie Bank was able to give API developers the self-service tools necessary to onboard new APIs or deploy changes *within hours*, compared to the multiple weeks it took with the previous processes and API management tools.

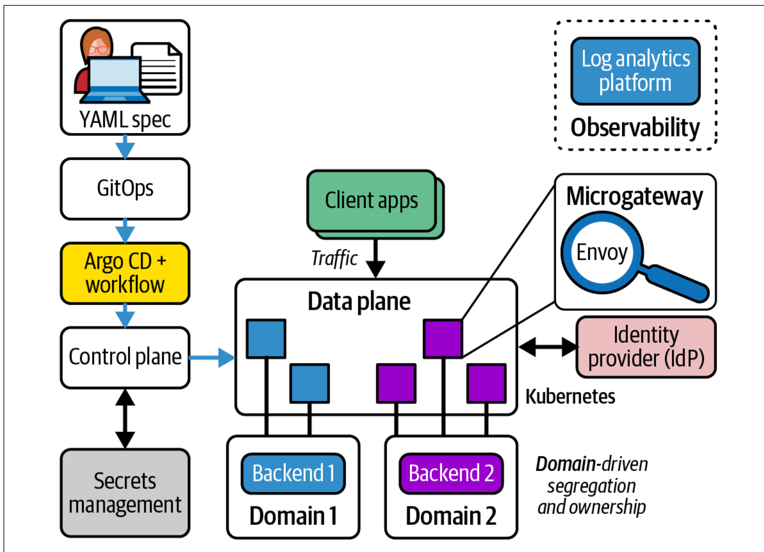

Figure 2-3. Macquarie Bank architecture for a modern API management system based on Gloo Gateway–based microgateways.

Let's take a closer look at the deployment architecture of API gateways to understand why a cloud native API gateway is a better fit.

Microgateway Architecture for Tenancy

Macquarie Bank adopted a microgateway architecture, which deployed a dedicated API gateway per "domain," as shown in Figure 2-4. A domain can represent a single service, a group of related services, or a set of services owned by a single team.

This was quite a bit different than the bank's existing legacy API management solution and architecture. By deploying multiple, smaller microgateways, teams were better able to control their configurations and eliminate issues such as "noisy neighbor" problems. Noisy neighbor becomes a problem when, for example, API A configures a gateway to behave a certain way that can impact the runtime behavior of API B. This also reduces the blast radius of misconfigurations: if API A takes down its microgateway, API B can still operate under its own microgateway.

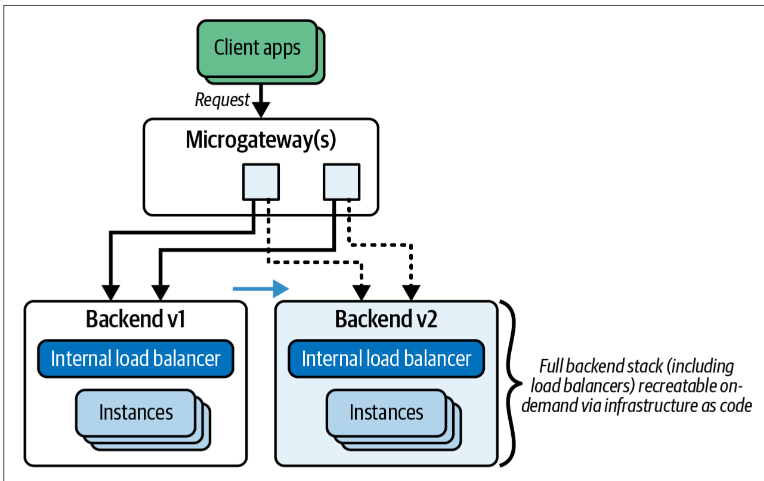

Figure 2-4. A microgateway pattern fronts various backends or domains of services.

But most legacy API management vendors offer a slimmed down “microgateway option”—so why didn’t that work for the Macquarie use case? In the case of the legacy vendors, the microgateway option is usually implemented differently from their core gateway and implements only a small subset of the overall API gateway. For example, Macquarie Bank needed the ability to do complex rate limiting, circuit breaking, and distributed tracing for its services deployed on Kubernetes in a decentralized and highly tenant way. The legacy microgateways do not offer these capabilities.

Bringing API Management to Traffic in All Directions

Decentralizing the API gateways and bringing them closer to the applications helps solve a lot of the ownership and tenancy problems discussed earlier. It allows platform teams to build self-service automation on top of the gateways. It also brings the API security and policy control close to the specific API endpoints. This helps to avoid single points of failure and gives more fidelity to load balancing and routing in highly dynamic environments like Kubernetes, without needing to involve additional infrastructure teams (external load balancers, etc.).

As we saw in the Macquarie Bank use case, we can use a modern API gateway combined with an internal developer platform to automate away a lot of the configuration needed to provide self-service. This model is a step in the right direction and solves a lot of the original challenges we discussed regarding legacy API management solutions. But this approach still treats API traffic as an "ingress problem." What about API calls in the so-called east-west direction? Or egress?

For these internal API calls, do you need to pass API keys? Do you need to establish TLS/mTLS? How do you end up enforcing service-to-service policy? How do service calls get appropriately routed to where their data is? For egress use cases, like AI/LLM, how can you ensure calls to these APIs are appropriately routed through an egress API gateway? How can you get monitoring, security, or other LLM-specific functionality like prompt management, augmentation, or failover? How can we extend this microgateway model of federated API management to be an omni-directional solution?

In the next chapter, we see how a service mesh can be combined with this microgateway pattern to solve the API-to-API connectivity problems, not only for the ingress direction, but for the east-west and egress directions as well.

CHAPTER 3

Omni-Directional API Management

In Chapter 2, we introduced a modern API gateway as a foundation that, when combined with platform engineering, puts you on the right track for a powerful self-service API management solution. Up until now, this approach, and just about every other API management solution, considered traffic from an "ingress" perspective—that is, traffic coming into a boundary (domain, cluster, data center, etc.). What about traffic between APIs within a boundary? What about traffic from APIs to services outside the boundary?

Modern API management should be omni-directional, not just ingress. In fact, most API usage within an enterprise is likely "internal" API calls, that is, APIs calling other APIs. Treating this traffic as ingress traffic could work, but there are a lot more moving pieces and ways to slip up. For example, to call an API you first need to identify the calling API in some way (API key, JWT token, etc.). To do this, you need to either exchange credentials (user name, password) at runtime or store long-lived tokens that can be retrieved at runtime. Managing all of these security tokens or credentials for potentially thousands of APIs and services can be problematic and risky.

We can simplify a lot of the challenges of API-to-API service traffic with a *service mesh*. A service mesh can also help with the challenges of egress traffic. But how does that fit with the foundation we discussed in the previous chapter? The legacy vendors have always

told us that an API gateway and a service mesh are two different tools for solving two different problems. This chapter will turn that premise on its head. We will get into some of the details of how these technologies come together and go deeper into the technology than the previous chapters. Buckle up!

Using a Service Mesh for East-West API Traffic

A service mesh solves service-to-service traffic challenges such as security, observability, and reliability. A service mesh uses proxies to intercept traffic and transparently control it, adding authentication, authorization, tracing, auditing, load balancing, traffic routing for blue/green deployments, and much more. Istio is a popular and widely deployed open source service mesh. Istio's initial service mesh offering deployed a proxy with each instance of an application to inject its traffic handling capabilities. This proxy, running alongside the application, is referred to as a *sidecar*. For example, with an Envoy sidecar proxy deployed with an application instance, as seen in Figure 3-1, Istio can transparently encrypt the traffic from source to destination with mTLS (mutual TLS) without the application knowing about it. This enables authentication of service calls on the wire without additional API tokens or JWTs. Istio handles all of the automation and operations for minting workload certificates used in the mTLS connections, including rotating the certificates. As we saw in Chapter 2, Envoy is a very powerful proxy, but is not a full API gateway.

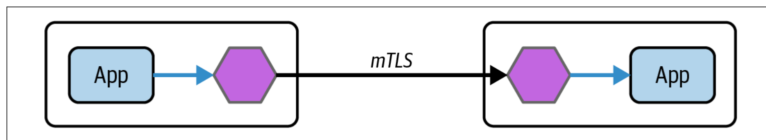

Figure 3-1. Most service mesh implementations follow a sidecar approach, which deploys a powerful L7 proxy to handle mTLS, routing, and observability.

Although a sidecar-based service mesh can deliver a lot of value to manage service connectivity and APIs, it's not a panacea. Injecting a sidecar into all instances of a service turns out to be an invasive operation. In Kubernetes, it forces the application pods to restart, changes assumptions about pod resources, can cause race conditions, and fundamentally couples the infrastructure to the

application, making it more difficult to patch and upgrade. In addition, a sidecar implements all of the mesh capabilities, so when you introduce it, it's difficult to introduce just certain features. Lastly, at scale, the resources the sidecars reserve can be fairly costly.

Istio Ambient Mode

Back in September 2022, Solo.io and Google built a sidecar-less implementation of Istio called *ambient mode*. In this mode, the required proxies are run outside of the application, as we'll see in the following sections. Additionally, Istio introduced a new, specialized proxy that allowed features to be layered and composed, solving the drawbacks of the sidecar and making incremental adoption of Istio's features possible. With Istio ambient mode, applications don't need to restart, and adoption of the mesh is truly transparent. Since there are no sidecars, ambient mode is almost an order of magnitude more resource efficient. It also lays the groundwork for a more powerful API management using microgateways, which ties in nicely with the concepts introduced in Chapter 2. This makes Istio ambient mode, combined with a powerful API gateway like Gloo Gateway, capable of powering an omni-directional, federated API management solution.

How Is Istio Ambient Mode Implemented?

Istio ambient mode implements its functionality with two layers (as shown in Figure 3-2), which can be adopted incrementally and composed together. The first layer is known as the *secure overlay* layer. It's responsible for creating mTLS connections on behalf of the services. This layer can be adopted independently of any other mesh features and forms the foundational layer of a zero-trust architecture, in which all connections and API requests are authenticated, authorized, audited, and encrypted.

The second layer, which sits on top of the secure overlay layer, is called the *waypoint* layer. It's responsible for any Layer 7 (L7) functionality (request load balancing, retries, traffic splitting, L7 telemetry, etc.) in the mesh. This layer is extremely flexible, as we'll see, and is where we can add powerful API management capabilities.

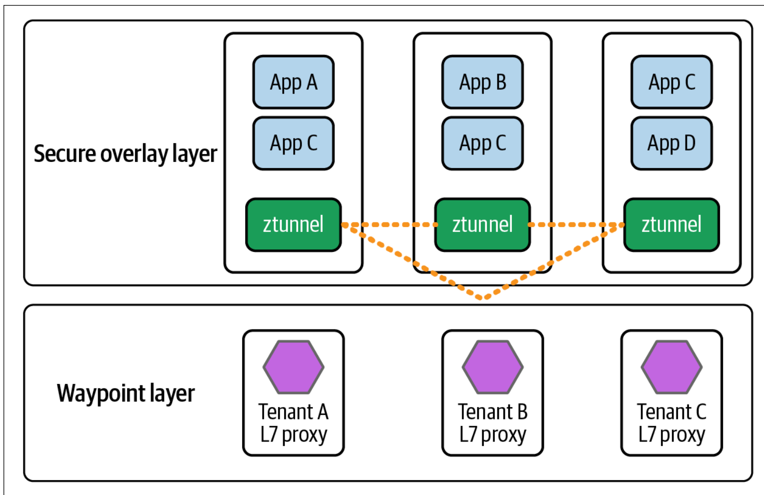

Figure 3-2. Istio ambient mode implements the mesh with two composable layers: a secure overlay layer to handle mTLS and a waypoint layer to handle L7 functionality.

Secure Overlay Layer

The secure overlay layer is implemented with a highly optimized Rust-based proxy called *ztunnel*. This node agent is responsible for associating x509 certificates from Istio's CA (certificate authority) to the connections that are opened from the app. These certificates use the SPIFFE (Secure Production Identity Framework for Everyone) spec to encode workload identity. With workload identity, we can specify authorization policies that are durable and not dependent on ephemeral constructs like IP address or network locations.

The secure overlay layer establishes mTLS connections with the remote application and does very basic Layer 4 (L4) telemetry reporting and connection load balancing. The mTLS connection actually originates in the application's pod, but the ztunnel does not run in the application's pod (see Figure 3-3). This layer works with any Container Network Interface (CNI) implementation including popular ones like Cilium, Calico, and Amazon Virtual Private Cloud (VPC). This layer is also interoperable with sidecar mode for migration purposes.

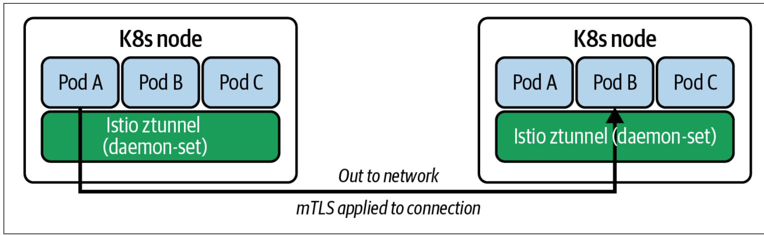

Figure 3-3. The secure overlay layer uses a custom-built data plane called the ztunnel.

Waypoint Layer

The waypoint layer is implemented with Envoy proxy (see Figure 3-4), which is the same proxy used for sidecars. You can enable the waypoint layer for all the workloads in a specific namespace or for specific services. The waypoint layer runs as a normal Kubernetes (K8s) deployment in the cluster and can be enabled when you need Layer 7 capabilities from the mesh. Layer 7 functionality includes the following:

- HTTP 1.x, 2, or 3
- Request routing
- Advanced load balancing
- Request mirroring
- Fault injection
- Request retries
- gRPC-specific capabilities
- Traffic splitting
- L7 telemetry

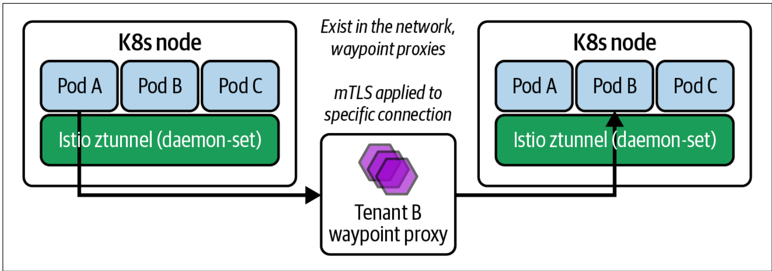

Figure 3-4. The ztunnel component can route to an L7 gateway in the waypoint layer for L7 processing.

If L7 processing is not needed, the mesh can completely bypass the waypoint layer. This saves significantly on performance and is an optimization not available for sidecars. For example, when service A wants to talk to service B, and they have been added to the mesh, they will automatically get mTLS mutual authentication between them. This is very fast L4 processing that happens through the ztunnels. If we want to add L7 processing, let's say checking a JWT token or doing some routing based on a header, we can deploy a waypoint for the namespace in which service B runs. Now, all traffic destined for services in that namespace, including service B, will go through the waypoint. You can think of the waypoint as a microgateway for each Kubernetes namespace, as shown in Figure 3-5. A waypoint can also be used as an egress gateway for services that run outside the cluster (or potentially as a SaaS). We will look closer at egress when we discuss AI/LLMs.

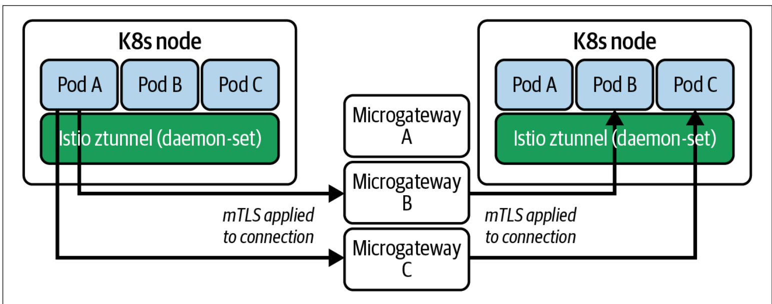

Figure 3-5. The waypoint layer follows a microgateway pattern, which can be tuned for tenancy to the namespace (default), a specific service, or even multiple namespaces.

The waypoint layer is configured for routing in Istio with the Kubernetes Gateway API. The classic Istio APIs (VirtualService) can also be used, but the community recommends switching to the Gateway API.

For more information on Istio ambient mode, please see *Sidecar-less Istio Explained* (O'Reilly, forthcoming in November 2024).

Enabling Omni-Directional API Management with Istio Ambient

Istio ambient mode brings a much more flexible solution to the table for east-west and egress traffic. As seen previously, the decoupling of the L7 functionality from the secure overlay layer means we can incrementally adopt security and gateway features. The model used for the waypoint layer, which is basically a namespace gateway, starts to look like the microgateway deployment we saw in the Macquarie Bank use case from Chapter 2. However, the proxy used in the waypoint layer for Istio ambient mode is just a stock, vanilla Envoy proxy.

Although this feature is not widely discussed, the Istio ambient waypoint architecture is intended to be pluggable: the implementation supports swapping the waypoint's default proxy with another proxy or gateway as a solution requires. At Solo.io, we replaced the default proxy with Gloo Gateway. This brings a full API gateway to the mesh in the east-west and egress directions. Now we get the benefits of mutual authentication/mTLS, telemetry, routing control, and resilience in all traffic directions with the benefits of an API gateway, as discussed in Chapter 2.

Istio ambient mode combined with Gloo Gateway, as shown in Figure 3-6, fits the requirements of a omni-directional, federated API management solution by using a delegating model for its API and being flexible, driven by automation, and fully decentralized. It can be used for ingress, east-west, and egress.

Gloo Gateway is also configured with the Kubernetes Gateway API, but just like in the ingress mode, it can be extended with more powerful features including JWT handling, rate limiting, request transformation, WAF, direct lambda invocation, GraphQL, and much more.

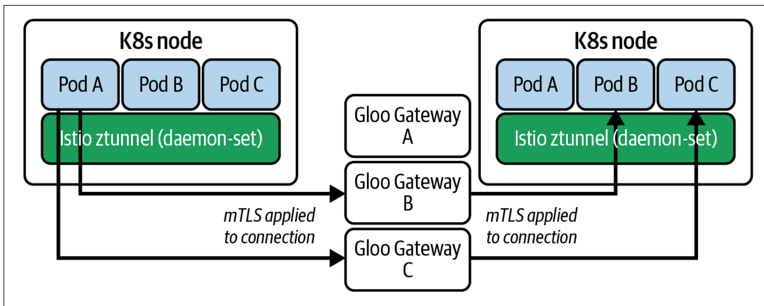

Figure 3-6. The waypoint layer can be pluggable in Istio ambient mode, which can accommodate a full API gateway such as Gloo Gateway.

Benefits of an Omni-Directional, Federated API Management

The most obvious benefit of omni-directional, federated API management is that all traffic in the API management system is secure by default because it is encrypted and authenticated with mTLS. There's no more trying to manage usernames, passwords, API keys, complex OAuth flows, and so on for API-to-API calls. Additionally, there are no single points of failure or centralized gateways that can impact all traffic if they fail. All gateways are decentralized following a microgateway pattern, and each gateway can be deployed in a highly available fashion. You can implement simple rate limiting up to complex quota usage at the route, API, client, or organization level. In fact, you can implement rate limiting based on anything you can see in the network call (connection, request, headers, body, etc.).

Traditionally, API-to-API calls require hairpinning out to a centralized gateway somewhere in a virtual private cloud or data center. This takes careful preparation, such as setting up load balancers to the API management system and to the workloads. This also takes DNS settings to point to the load balancer virtual IPs. Then you have to set up TLS certificates on the load balancer and, ideally, on the backend services. Omni-directional API management simplifies all of this.

In an enterprise, there will often be a need for more complex or backward compatible interaction protocols that the API gateway needs to handle. Checking API keys, verifying HMAC (hash-based

method authentication code), caching authorization tokens, and so on might need to still happen. By having a full API gateway (such as Gloo Gateway in our example), we can handle these use cases without any problems.

How Do New Use Cases Like AI/LLM Usage Fit into the Equation?

As we've seen, Istio ambient mode can help with egress traffic. For the LLM use cases presented in Chapter 1, we can automatically and transparently route traffic through a waypoint gateway implemented with Gloo Gateway. When doing so, we can apply specific security policies, rate limiting, and LLM prompt management or failover. These features can be introduced transparently to the applications without any code changes.

For example, we can set up a waypoint proxy specifically for communicating with OpenAI APIs, as illustrated in Figure 3-7. This waypoint proxy can be implemented with Gloo Gateway, which provides LLM policy-specific capabilities.

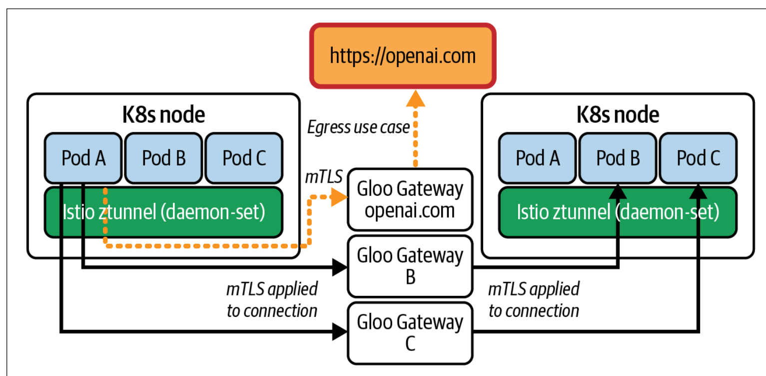

Figure 3-7. Istio ambient mode can help apply policy to egress traffic by routing to an egress waypoint, which is helpful in SaaS and LLM use cases.

Gloo Gateway can automatically inject the required API keys to call *openai.com*. It can also specify system prompts that should be included with the user prompt to promote consistency in results. Gloo Gateway can filter requests and responses for sensitive data (personally identifiable information [PII], credit card numbers, etc.) to avoid sending disallowed data to the LLM. To avoid expensive

mistakes, misuse, or rogue clients in the enterprise, this egress Gloo Gateway can also rate limit based on LLM tokens. Lastly, Gloo Gateway can be used for more advanced LLM use cases like token-based failover (used all tokens for the account) to another LLM, or things like retrieval-augmented generation (RAG).

One thing to note about egress use cases, including the LLM use cases, is that although traffic is encrypted with mTLS to the egress gateway, it is not encrypted after that. It is up to the platform operator to configure outbound connections with the proper security. Additionally, the service mesh should be used in conjunction with a Kubernetes NetworkPolicy to add defense in depth for egress use cases.

Bringing It All Together

Combining a powerful API gateway like Gloo Gateway with an east-west and egress traffic control like Istio ambient mode into a micro-gateway deployment gives us a very powerful omni-directional API management solution. The last piece of the puzzle is how this all fits together for your platform. Let's dig into that in the next chapter.

CHAPTER 4

Tying Omni-Directional API Management into Your Platform

Modern API management for internal developer platforms balances flexible developer tool choices with golden paths for runtime, including deployment, operation dashboards, and adhering to governance. To achieve this ideal, you need to practice an "everything as code" mentality and use APIs/interfaces to the internal developer platform. Following an operational model where everything is code, we can put controls and automation in place to achieve compliance and governance. We can put infrastructure as code (IaC), configuration as data, and policy as code into our GitOps workflows. This provides version control, repeatability, consistency, and safety when doing rollouts. It also lays the groundwork for self-service. The platform team can expose a UI, CLI, and APIs to facilitate developer interactions with the platform. API developers can choose the method of interaction with which they feel most comfortable to deploy changes to their APIs. This interaction is illustrated at a high level in Figure 4-1.

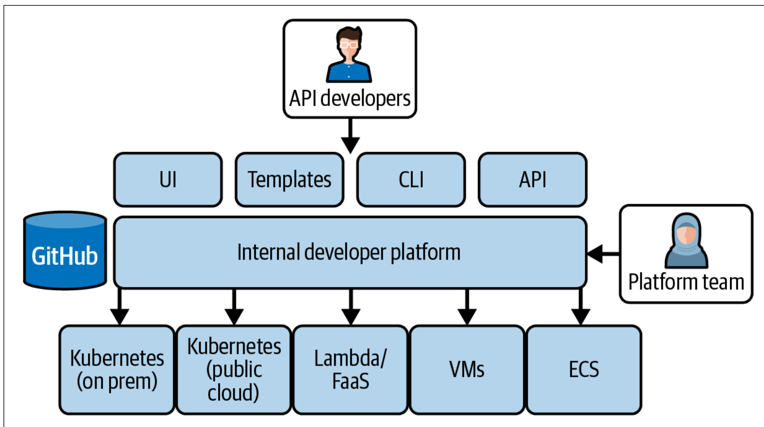

Figure 4-1. The internal developer platform enables API developers to deploy their changes following a self-service model, while abstracting away the details of the backend infrastructure. The workflow follows a GitOps-based model.

The platform automation handles the deployment, rollout, and orchestration to the backend infrastructure. Organizations can choose to use containers, functions, and/or VMs based on what's best for their use case. For example, an AWS Lambda deployment might be the right choice for APIs that are rarely used or intended to facilitate integrations between AWS services. Or if they have a good understanding of the API usage but still need horizontal scaling, organizations may choose to use Kubernetes. Lastly, some APIs may need to be deployed to VMs because they've not yet been (or cannot be) migrated to a container environment. The platform team working with the developer team can decide which is best for which use case, keeping in mind use cases may evolve, with the API moving from one infrastructure to another. This should not affect the developers' workflow.

When it comes to deployment and runtime, the platform should be tool agnostic. APIs need tools for designing, testing, mocking, linting, and others tasks. Developers can be picky about their tools; they may choose certain tools based on their own background or on the language used to build the API. This is not to say the platform needs to support custom integrations with an unbounded selection of tools, but rather to say the interface between the developers and the platform should be a standard interface based on declarative configuration and GitOps. Whatever tools developers use, they should be

able to produce the deployment documents (YAML, JSON, OpenAPI Specification, etc.) that fit with the platform and CI/CD, as illustrated in Figure 4-2.

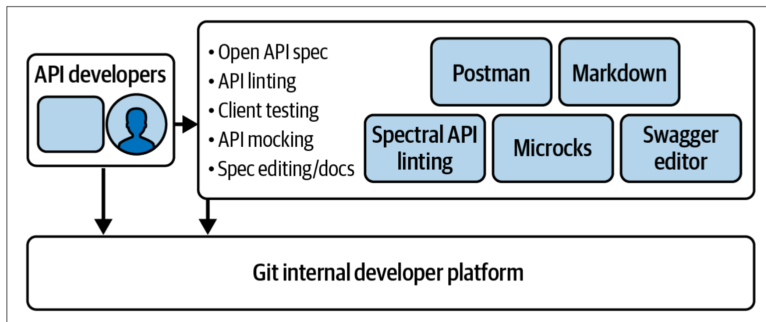

Figure 4-2. Common tooling used by developers should be pluggable into the internal developer platform through standard interfaces, such as declarative configuration.

Building Golden Paths

The primary goals of an internal developer platform are to improve developer experience, shorten the time for developers to deliver business value, and improve compliance. Building out golden paths —or push-button deployments that adhere to security, observability testing, documentation, reliability, and general operational best practices—is key to achieving these goals. Let's look at a few of the capabilities that an internal developer platform should have.

Platform User Interface and Dashboards

The platform should have a user interface that developers can use to start new API projects, manage deployments, and dig into operational metrics. Some organizations may choose to build their own tailored UI, while others might use existing portal technologies like Backstage. While a GUI is important, some power users may prefer to use a CLI or API directly, so options to do so should be incorporated into the platform as well.

Control Plane API and Templates

Users should be shielded from the underlying details of the platform. Developers should not have to know about the details of Istio or a CNI to do a deployment. Those details should be hidden

in the automation the platform team builds. However, some developers will want more direct access to the underlying API objects for customization or because they are more comfortable working that way. To facilitate this, platform teams can either build an API for their platform control plane or directly expose templates that developers can use to customize the platform. For example, using Helm to package well-known templates with some extension points is a popular choice.

Platform in Layers

Omni-directional API management fits within the rest of the platform automation as a layer of the delivery pipeline. API lifecycle is not some separate, siloed feature of a monolithic API management system; rather, it's fully integrated into the automation you've built for the rest of your software delivery system, as shown in Figure 4-3.

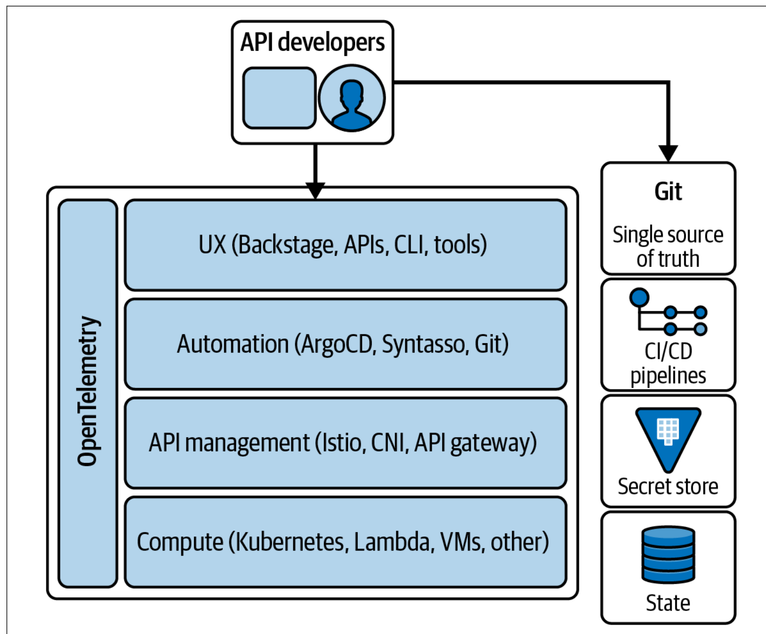

Figure 4-3. Layering tools to support an internal developer platform.

This layer must consist of tooling that handles API traffic for ingress, east-west, and egress. Trying to have separate tools for each direction creates confusion, operational challenges, overlap, gaps, and costly integrations. The federated API management solution

built on Gloo Gateway from Chapter 3 solves this problem and fits nicely in a modern internal developer platform.

Adopting Omni-Directional API Management in Your Platform

So far we've seen a number of theories, technologies, architectures, and discussion around an ideal state. Most organizations on their modernization journey are trying to bring along their outdated solutions or evaluating what might work best for them if they adopt something newer. There are generally some prerequisites to getting started on this journey and then a three-phase adoption plan.

Prerequisites

Understanding the needs of API developers and laying the foundation for an internal developer platform is a strong prerequisite for modernizing your API management to support omni-directional traffic. This chapter introduced some context and principles for putting together an internal developer platform, but the details are beyond the scope of this report. See Gregor Hohpe's book *Platform Strategy* (independently published). The most important aspects of the platform will be how you choose to automate and orchestrate it, as that is where a modern API management system would fit in (as depicted in Figure 4-3).

Phase One

The first step to modernizing your API management and integrating it with your developer platform is to find an API gateway that fits nicely into your platform. Focus on one that is flexible enough to support your IaC principles and GitOps workflows. You'll also need to consider whether it supports various runtimes, container environments, and the ability to delegate configuration decisions directly to the developer teams as needed. You should strive for a standard routing API such as the Kubernetes Gateway API. Lastly, you'll want to decide the best way for developers to interact with this system. Will they be touching configuration objects themselves and colocating these with their service? Or will they have templates or golden path configurations they can adopt? Or will it be driven primarily by a UI? Working with developers to deliver this functionality is the best approach.

Phase Two

In the next phase, you'll need to decide how best to secure traffic for your APIs. For the ingress direction, you may decide to use common mechanisms supported by the API gateway (API keys, JWT, OIDC, etc.). For internal API usage, you should probably look at automating away things such as workload identity and combining that with end user identity. Writing policies using these composite identities for API calls can be done with a service mesh in the east-west direction, as discussed in Chapter 3. Ideally, the developers should not know the details of the system running under the covers, but be able to drive it with templates and golden path configurations set up by the platform team.

Phase Three

In the last phase, you'll want to combine the API gateway and the service mesh to get consistent routing, traffic, security, and observability policies regardless of which direction the traffic is intended: ingress, east-west, or egress. Standardizing these policies will significantly simplify the toil that teams face when trying to deliver APIs. Unifying this, and driving it through automation, will reduce the errors and inconsistencies seen in organizations, which are recipes for misconfigurations.

Improving Self-Service, Resilience, and Innovation

An omni-direction API management solution, based on a modern API gateway like Gloo Gateway and Istio, provides an extremely powerful solution that integrates nicely within an internal developer platform. This solution provides end-to-end security, zero trust, tenancy, and resilience, and enables self-service access to make API delivery changes. This architecture has been proven to improve business performance as demonstrated by the Macquarie Bank use case discussed in Chapter 2. More innovation is happening in this space, including the ability to use any cloud gateway as a microgateway within a federated solution with a single pane of glass for management. Legacy monolithic API management solutions are outdated and should be avoided for modern projects. New, cloud native approaches have emerged and should be considered.

About the Author

Christian Posta (@christianposta) is VP, Global Field CTO at Solo.io. He is the author of *Istio in Action* as well as many other books on cloud native architecture and is well known in the cloud native community for being a speaker, blogger, and contributor to various open source projects in the service mesh and cloud native ecosystem (Istio, Kubernetes, etc.). Christian has spent time at government organizations, commercial enterprises, and web-scale companies, and now helps organizations create and deploy large-scale, cloud native, resilient, distributed architectures.