AMBASSADOR

Effective Management of APIs and the Edge when Adopting Kubernetes

Table of Contents

Executive Summary

When integrating an API gateway with a microservices-based application running on Kubernetes, you must consider two primary challenges: how to scale the management of many services and APIs; and how the gateway can support a broad range of microservice architectures and protocols.

There are three primary strategies for managing APIs in this new context: deploying an additional Kubernetes API gateway; extending an existing API gateway; and deploying a self-service edge stack.

Managing the edge of the system has always been complicated. Adding more services with a diversity of architectures only increases the demands on the edge. Platform teams must design, choose, and implement their API gateway and edge strategy and tooling accordingly.

Building applications using the microservices pattern and deploying these services onto Kubernetes has become the de facto approach for running cloud-native applications today. In a microservice architecture, a single application is decomposed into multiple microservices. Each microservice is owned by a small team that is empowered and responsible to make the right decisions for the specific microservice.

This responsibility typically extends from the edge of the system where the user requests arrive, right the way through to the service's business logic and down into the associated messaging and data store schema.

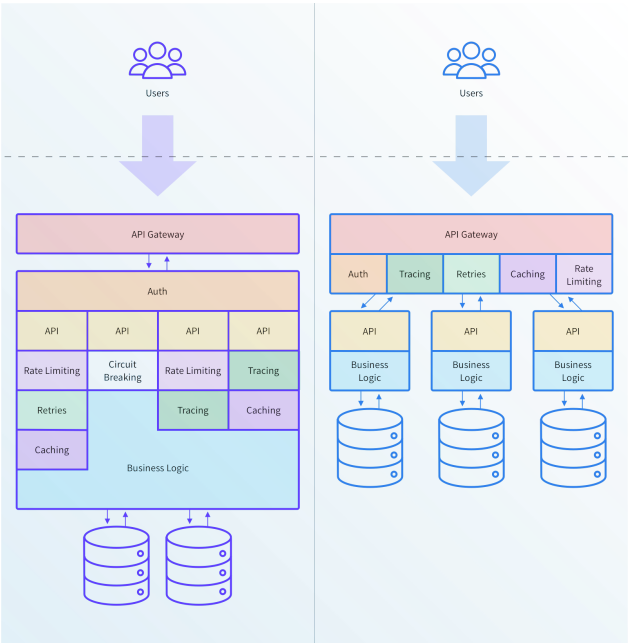
When integrating an API gateway with a microservices-based application running on Kubernetes, you must consider two primary challenges:

- How to scale the management of 100s of services and the associated APIs; and
- How the gateway can support a broad range of microservice architectures, protocols, and configuration that typically spans the entire edge stack.

The API Gateway: A Focal Point with Microservices

An API gateway is at the core of how APIs are managed, secured, and presented. It is deployed as a software component (or series of components) on virtual machines or within Kubernetes, and acts as the single entry point into a system. The primary responsibility of an API gateway is to enable multiple APIs, microservices, and backend systems to be accessed reliably and safely by users.

Microservices and Kubernetes provide implementation flexibility. For example, one team may elect to expose a container-based microservice at the edge of the system (the boundary between the internal services and end users) as a set of REST APIs over HTTP. Another team may choose Protobufs and gRPC. A team with real-time streaming requirements may expose their microservice over WebSocket APIs. Any APIgateway deployed within Kubernetes must support all of these protocols.



Each team is not only free to make these choices, but they are also responsible for the consequences; this often translates into "you build it, you run it". Although not every organisation subscribes completely to this way of working, every microservice team needs to be able to understand, diagnose, and

configure all aspects of the handling of each service and each user's request into the application. The diversity of runtime requirements related to applications and APIs means that each team will be working with all layers within the edge stack, for example, the dynamic request handling, the WAF, and any caching implementation.

The Edge and Kubernetes Ingress

Microservices need to be accessible to end users. The boundary between internal microservices and end users is known as the **edge**. In order for end users to access internal applications, traffic needs to cross the edge. In Kubernetes, traffic crosses the edge using a piece of software known as an **ingress**.

The development paradigm of microservices -- independent, empowered, and responsible teams -- creates a new set of challenges for microservice teams working with API gateways, Kubernetes ingress and the edge. In this article, we identify two important challenges for the edge: managing independent microservices, and having access to a full-service gateway.

Challenge #1: Scaling Edge Management

The challenges of managing the edge increase with the number of microservices deployed

In a microservices architecture, engineers will be managing many more services and applications. Each team needs to be able to manage their services independently, in order for releases to be decoupled from other teams schedules. The traditional approach to exposing applications at the edge is typically done through a centralized operations or platform team. However, a single ops team cannot scale to handle the volume of changes that are necessary when an organisation has hundreds of microservices.

Typical changes that require modification of configuration at the edge:

- New version of a service being deployed.
- Modifying endpoints, routing instructions, or the associated backend services.
- Changes to authentication and authorization services.
- Modification of nonfunctional requirements, such as rate limiting, timeouts, retry patterns, and circuit-breaking.
- User testing of new functionality, for example, enabling a feature for a small subset of beta test users.

Adopting a microservices-based architecture will result in a significant rise in the number of releases. This increase only magnifies the edge management challenges and increases the strain on a centralized approach to operation.

Challenge #2: Supporting a Diverse Range of Edge Requirements

Microservices introduce a number of new concerns at the edge

The microservice architecture enables architectural flexibility. Application developers take advantage of this flexibility to choose the programming language and architecture that best fits the specific requirements of the service. The edge needs to support the broad spectrum of functionality that need to be exposed to users, regardless of architecture. This extends the traditional role of the API gateway, and some of the challenges related to the need of consolidated tooling at the edge includes:

- The ability to adeptly route a wide variety of protocols. Common protocols include HTTP/1.1, HTTP/2, WebSockets, gRPC, gRPC-Web, and TCP.
- Provide the full aggregate set of edge capabilities needed by any specific service, ranging from traffic management to observability to authentication and beyond.
- Exposing these capabilities in a self-service model for application developers.

Encouraging a diversity of implementation within microservice teams allows engineers to choose the "right tool for the job". However, consolidation of the underlying platform offers many benefits. Rather than allowing developers to build bespoke implementations for additional protocol support or security handling, it is much more manageable and scalable to present them with a pre-approved "buffet" of options at the edge, so that they can pick and choose the most appropriate combination of functionality.

The Increasing Importance of the Edge

As organizations adopt Kubernetes and shift to a microservices-based architecture, a new set of challenges emerge at the boundary between end users and the internal microservices. This "edge" of the system, and related technologies like an API gateway, are therefore a focal point when adopting microservices. These new challenges at the edge are driven by the organizational model of microservices, where independent teams are empowered and responsible to make the right architectural and implementation decisions for a microservice.

Challenge #2: Supporting a Diverse Range of Edge Requirements

Microservices introduce a number of new concerns at the edge

The microservice architecture enables architectural flexibility. Application developers take advantage of this flexibility to choose the programming language and architecture that best fits the specific requirements of the service. The edge needs to support the broad spectrum of functionality that need to be exposed to users, regardless of architecture. This extends the traditional role of the API gateway, and some of the challenges related to the need of consolidated tooling at the edge includes:

Encouraging a diversity of implementation within microservice teams allows engineers to choose the "right tool for the job". However, consolidation of the underlying platform offers many benefits. Rather than allowing developers to build bespoke implementations for additional protocol support or security handling, it is much more manageable and scalable to present them with a pre-approved "buffet" of options at the edge, so that they can pick and choose the most appropriate combination of functionality.

The Increasing Importance of the Edge

As organizations adopt Kubernetes and shift to a microservices-based architecture, a new set of challenges emerge at the boundary between end users and the internal microservices. This "edge" of the system, and related technologies like an API gateway, are therefore a focal point when adopting microservices. These new challenges at the edge are driven by the organizational model of microservices, where independent teams are empowered and responsible to make the right architectural and implementation decisions for a microservice.

Three Strategies for Managing APIs and the Edge with Kubernetes

There are three strategies that engineering teams can apply in order to effectively manage the edge when migrating to microservices and Kubernetes: deploying an additional Kubernetes API gateway; extending an existing API gateway; and deploying a comprehensive self-service edge stack.

The presentation of each strategy includes a technical overview, a discussion of the associated pros and cons, and an analysis of how the solution can meet each of the two primary challenges with an API gateway when adopting Kubernetes.

Strategy #1: Deploy an Additional Kubernetes API Gateway

As suggested by the name, the implementation of this strategy simply consists of deploying an additional API gateway at the edge. Typically an organisation with a web-based application that has existed for more than a few months will already have a series of components knitted together that provide edge and API management, such as a Layer 4 load balancer, Web Application Firewall (WAF), and traditional API gateway. With the "deploy an additional Kubernetes API gateway" strategy a platform team will simply deploy a completely separate gateway within the new Kubernetes clusters.

There are two variations for implementing this strategy. The first is to deploy the additional API gateway directly "below" the existing gateway. Here relevant traffic is forwarded from the existing gateway to the new and onto appropriate services. The second variation consists of deploying the new API gateway "alongside" the existing solution. With this approach, traffic is forwarded from an edge component higher up the stack, such as the layer 4 load balancer, to the new gateway, which in turn forwards the traffic to the Kubernetes services.

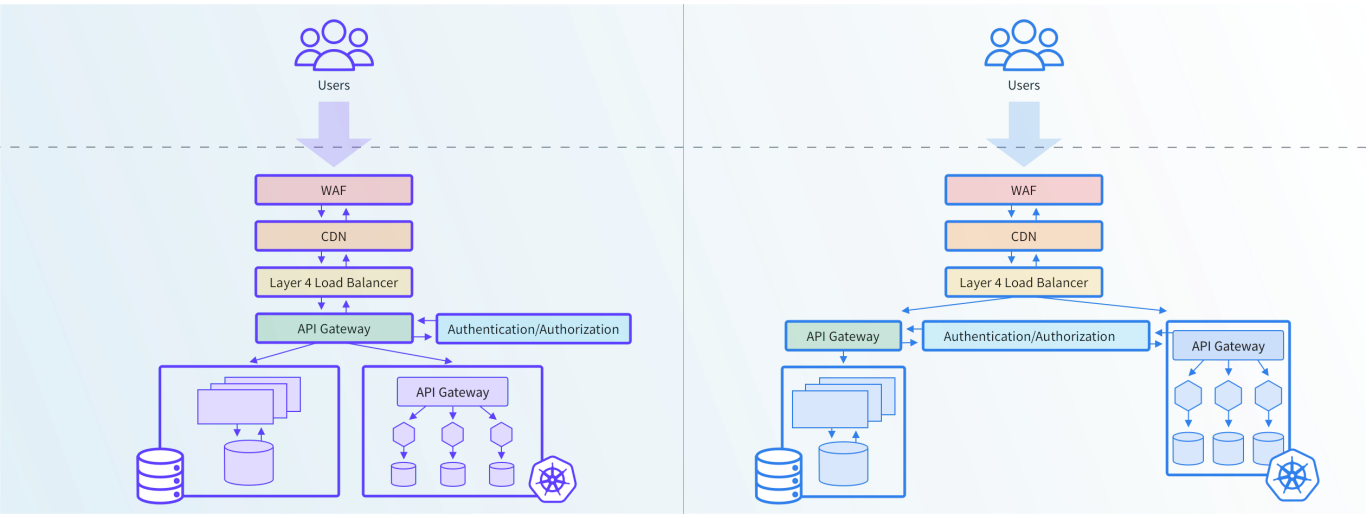
Once this strategy has been employed it can be operated and maintained using one of two approaches. Within some organisations, the existing platform or infrastructure team will be responsible for deploying and maintaining the new gateway. Other times, a completely separate team will own the new solution, or the responsibility will be pushed onto the individual microservice development teams. Regardless of the exact organisational structure, many of the routine changes that need to be made to the edge will need to be coordinated across the multiple API gateways and related components.

Key Points: Deploy an Additional Kubernetes API Gateway

This strategy is easy to plan and implement: simply deploy and operate an existing Kubernetes-specific API gateway alongside the existing solution. However, this approach typically adds additional management and coordination costs to routine API changes, and dealing with extra interdependent layers at the edge can increase the time taken to locate and fix faults.

Architecture

The following diagram provides an architectural overview of the two typical implementations of the "deploy an additional API gateway" strategy. The left image shows the deployment of the new gateway "below" the current gateway, and the right image demonstrates the deployment of the new gateway "alongside" the existing solution:



Trade-offs

The following section highlights the key pros and cons with the "add an additional API gateway" strategy:

Pros:

- There is minimal change to the core edge infrastructure. This strategy typically allows the use of all of the existing edge components.
- Incremental migration between VM-based and Kubernetes-based applications is easily supported by maintaining the two gateways concurrently.
- This approach provides a limited blast radius when catastrophic failures at the edge are encountered e.g. only the Kubernetes-based applications are affected if the new API gateway encounters an issue.

Cons:

- There is an increased management overhead (and increased cognitive load) of working with different components, e.g., each API gateway will most likely have a different UI, SDK, or API.
- There is increased operational overhead with this approach, as an additional mission-critical API Gateway needs to be updated, maintained, and monitored.
- It can be challenging to expose the complete functionality of the edge components to each independent microservice teams (and difficult to support coordination). For example, if the existing API Gateway is used for authentication, microservice teams may not be able to directly configure this.
- It may potentially be difficult for all teams to understand the new architecture that is composed of multiple similar components. On a related theme, it may also be challenging to locate and debug where within the edge stack any issues occur.

Addressing Challenges

To scale edge management in this strategy, an organisation should implement a "multi-tier" edge management strategy:

- As much edge functionality as possible should be pushed into the Kubernetes API Gateway, and directly exposed to application developers. This maximizes agility and velocity within the development teams.
- For edge functionality that needs to remain centralized, the operations team should create a workflow for application developers, and support this with SLAs.
- Application development teams should use these SLAs in their release planning to minimize release delays.
- Properly chosen, the additional Kubernetes API Gateway can provide the breadth of functionality necessary to support a diverse range of microservices. This includes support for contemporary L7 protocols such as gRPC and gRPC-Web as well as observability and resilience features such as circuit breakers and automatic retries. Crucially, for newer protocols, existing edge components should be configured to pass-through traffic on specific routes to the additional gateway for proper functionality. Note that there is a tradeoff in this approach, as for these routes the rest of the existing edge stack cannot be used.

Strategy #2: Extend Existing API Gateway

The "extend existing API gateway" strategy is implemented by modifying or augmenting the existing API gateway solution that is currently deployed in production. The key goal with this modification is to enable synchronization between the API endpoints that a user accesses and the location of the corresponding services deployed within the new Kubernetes clusters.

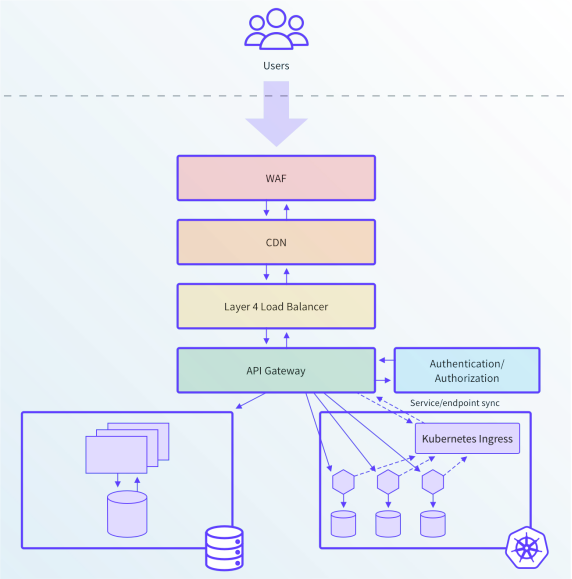
In Kubernetes, this is typically done by deploying a custom ingress controller for the existing API Gateway or load balancer. The ingress controller will parse Kubernetes ingress configurations, translate the ingress configuration into the native API Gateway format, and configure the API Gateway or load balancer via their respective APIs.

Key Points: Extend Existing API Gateway

This strategy ensures that the platform team will only be dealing with a single API gateway and set of edge components. However, this approach requires modifying the existing gateway configuration workflows in order to avoid any potential collision between the existing routes/services that are running outside of the Kubernetes cluster and the routes/services that are being deployed within the new cluster.

Architecture

The following diagram provides an architectural overview of the typical implementation of the "extend existing API gateway" strategy. In this diagram, the ingress controller is deployed on a standalone pod in the cluster, reading Kubernetes ingress configuration. The configuration is then passed to the existing API Gateway / load balancer, which is deployed outside of the cluster.



Trade-offs

The following section highlights the key pros and cons with the "extend existing API gateway" strategy:

Pros:

- This strategy provides the ability to use the existing API gateway; this technology is tried and tested, and the platform team have familiarity with this
- The platform team can leverage existing integrations with on-premises infrastructure and services e.g. hardware load balancers routing to a new on-premises k8s cluster
- There is minimal need to learn Kubernetes networking technologies, especially for the development teams who will not interact with the edge or cluster ingress

Cons:

- Existing configuration workflows will need to change in order to preserve a single source of truth for the API gateway configuration
- Most custom ingress controllers expose a limited amount of configuration parameters via Kubernetes annotations, and require cumbersome ConfigMaps or other workarounds to configure more advanced options
- Additional education and training may be required to educate developers, especially configuration parameters that are not native to the ingress controller

Addressing Challenges

To scale edge management using this API gateway extension strategy, an organisation should fully embrace the recommended configuration approach as specified by the Kubernetes ingress controller, and shift away from the traditional API/UI-driven configuration model of their existing gateway. In addition, a standardized set of scripts should be used so any modification of routes to services running outside the Kubernetes cluster does not conflict with the services running inside the new cluster

Before adopting the strategy, an architectural roadmap review of current and anticipated edge requirements for microservices is essential. Ensuring the existing gateway / load balancer supports these requirements will avoid a situation where adding another API gateway is required to support an unexpected requirement.

Strategy #3: Deploy a Comprehensive Self-Service Edge Stack

Deploying a self-service edge stack consists of using a Kubernetes-native API gateway that contains additional integrated supporting edge components such asWAF and authentication/authorization. The edge stack is installed in each of the new Kubernetes clusters, and replaces and consolidates the majority of existing edge and gateway functionality that previously ran outside the cluster.

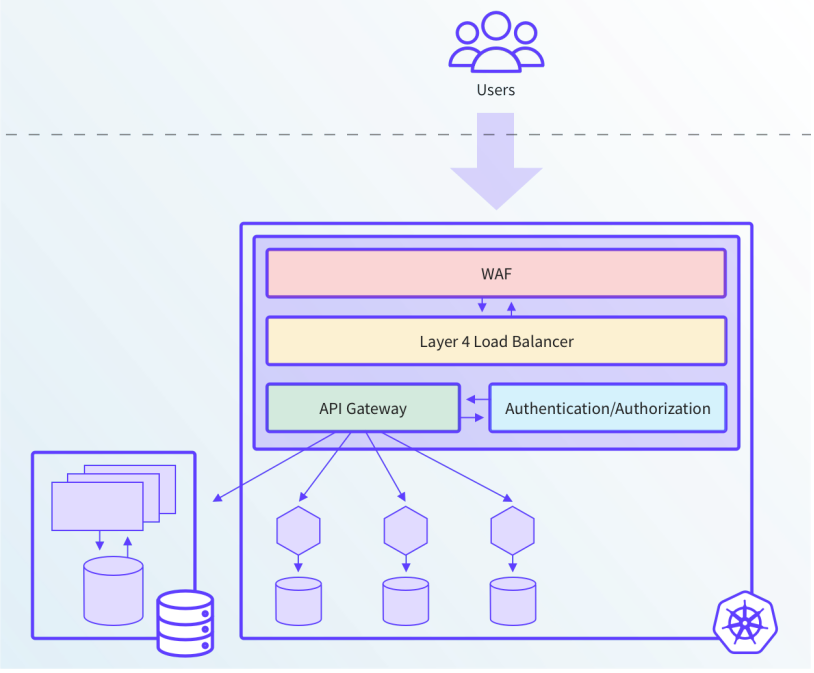
Once this strategy is deployed, it is common to see the platform team responsible for the underlying infrastructure components (and their successful operation and monitoring) and also providing "sensible defaults" for managing cross-functional requirements, such as default timeout values, retries, and circuit breaking. The independent microservice teams are then fully empowered and responsible for configuring the edge stack as part of their normal workflow; for example, when a new service is deployed, the development team will also specify the routing configuration and override the existing cross-functional defaults.

Key Points: Deploy a Self-Service Edge Stack

Using this strategy ensures that the platform team will only be dealing with a single API gateway and consolidated set of edge components. Using a Kubernetes-native edge stack also provides easy integration with this platform, and supports cloud native best practices, such as self-service, declarative configuration and a "single source of truth" for configuration.

Architecture

The following diagram provides an architectural overview of the two typical implementations of the "deploy a comprehensive self-service edge stack" strategy.



Trade-offs

The following section highlights the key pros and cons with the "deploy a comprehensive self-service edge stack" strategy:

Pros:

- This strategy provides good integration with the new API gateway and the surrounding "cloud native" stack, such as Kubernetes, service meshes, and online identity (IdP) brokers
- Edge management is simplified into a single stack that is configured and deployed via the same mechanisms as any other Kubernetes configuration. This can enable independent microservice teams to "self-serve" all of their configuration changes.
- This approach allows the engineering team to embrace cloud native best practices, such as defining a "single source of truth" for edge configuration, supporting dynamic/elastic infrastructure, and aiding rapid config changes (via approaches like GitOps)

Cons:

- There is potentially a large architectural shift. Existing edge components will be replaced, and existing network architectures and configuration made need to change
- This strategy will require the platform team learn about new proxy technologies and potentially new edge component technologies
- There will be changes to engineering workflow. For example, along with the ability for independent microservice teams to configure the edge to support a wide range of protocols and architectures, comes increased responsibility, both at design-time and during operation.

Addressing Challenges

In the self-service edge stack strategy, each microservice team is empowered to maintain the edge configuration specific to each of their microservices. The edge stack aggregates the distributed configuration into a single consistent configuration for the edge. To support the diversity of the edge services, adopt an edge stack that has been built on a modern L7 proxy with a strong community such as the Cloud Native Computing Foundation's Envoy Proxy. The breadth of the community helps ensure that the core routing engine in the edge stack will be able to support a diversity of use cases.

Conclusion

Managing the edge of the system has always been complicated. Adding more services with a diversity of architectures only increases the demands on the edge.When integrating an API gateway with a microservices-based application running on Kubernetes, you must consider two primary challenges: how to scale the management of 100s of services and the associated APIs; and how the gateway can support a broad range of microservice architectures, protocols, and configuration that typically spans the entire edge stack.

There are three primary strategies for managing APIs and the edge of a system when migrating to a microservices-based architecture deployed into new Kubernetes clusters. This whitepaper has provided an overview of the three approaches: deploying an additional API gateway; extending an existing API gateway; and deploying a self-service edge stack.

Platform teams must design, choose, and implement their API gateway and edge tooling accordingly.

How Can Ambassador Help?

If you're looking for a single self-service solution to address the comprehensive needs of your edge stack, try the Ambassador Edge Stack.

To learn more about the Ambassador Edge Stack and get started today go to getambassador.io/products.

Trade-offs

The following section highlights the key pros and cons with the "deploy a comprehensive self-service edge stack" strategy:

Pros:

- This strategy provides good integration with the new API gateway and the surrounding "cloud native" stack, such as Kubernetes, service meshes, and online identity (IdP) brokers
- Edge management is simplified into a single stack that is configured and deployed via the same mechanisms as any other Kubernetes configuration. This can enable independent microservice teams to "self-serve" all of their configuration changes.
- This approach allows the engineering team to embrace cloud native best practices, such as defining a "single source of truth" for edge configuration, supporting dynamic/elastic infrastructure, and aiding rapid config changes (via approaches like GitOps)

Cons:

- There is potentially a large architectural shift. Existing edge components will be replaced, and existing network architectures and configuration made need to change
- This strategy will require the platform team learn about new proxy technologies and potentially new edge component technologies
- There will be changes to engineering workflow. For example, along with the ability for independent microservice teams to configure the edge to support a wide range of protocols and architectures, comes increased responsibility, both at design-time and during operation.

Addressing Challenges

In the self-service edge stack strategy, each microservice team is empowered to maintain the edge configuration specific to each of their microservices. The edge stack aggregates the distributed configuration into a single consistent configuration for the edge. To support the diversity of the edge services, adopt an edge stack that has been built on a modern L7 proxy with a strong community such as the Cloud Native Computing Foundation's Envoy Proxy. The breadth of the community helps ensure that the core routing engine in the edge stack will be able to support a diversity of use cases.

About Ambassador

Ambassador is a Kubernetes-native API Gateway built by the team at Datawire (datawire.io). With the Ambassador API Gateway, organizations can ship software with greater agility. Ambassador gives platform engineers and operators a self-service edge proxy, reducing toil. At the same time, Ambassador gives application developers fine-grained control over their edge proxy needs. Built on Kubernetes and Envoy Proxy, Ambassador provides state-of-the-art functionality for observability, resilience, and traffic management.

For more information, visit our website (www.getambassador.io), check out our blog (blog.getambassador.io), or follow us on Twitter (@getambassadorio).