Moving to the Cloud

Exploring the API Gateway to Success

Daniel Bryant
Head of Developer Relations

Table of Contents

03 **Part 1:** Moving to the Cloud: Exploring the API Gateway to Success

08 **Part 2:** Microservice Service Discovery

16 **Part 3:** Load balancing strategies in Kubernetes

20 **Part 4:** Configuring Kubernetes Ingress on AWS

25 **Part 5:** HTTP/3: Use Cases, Envoy Support, and Google's Rollout

Part 1:

Moving to the Cloud:

Exploring the API Gateway to Success

What is an API gateway?

An API gateway is a front door to your applications and systems. It's on the hot path of every user request, and because of this, it needs to be performant, secure, and easily configurable. The fundamentals of API gateway technology have evolved over the past ten years, and adopting cloud native practices and technologies like continuous delivery, Kubernetes, and HTTP/3 adds new dimensions that need to be supported by your chosen implementation.

Moving to the cloud through the lens of API gateways

This article explores the benefits and challenges of moving to the cloud through the lens of API gateways and highlights the new practices and technologies that you will need to embrace.

At Ambassador Labs, we've learned a lot about deploying, operating, and configuring cloud native API gateways over the past five years as our Ambassador Edge Stack API gateway and CNCF Emissary-ingress projects have seen wide adoption across organizations of every size.

Our team at Mercedes-Benz uses Ambassador as an Ingress for all HTTP communications from 12 million cars in production. We see about a billion hits per day, serving 14 million requests at its peak through Ambassador in a five-minute interval through all our regions combined. Mercedes-Benz collects roughly nine terabytes of traffic from requests in a day."

Nashon Steffen
Staff Infrastructure Development Engineer

Adopting cloud native: Changes, challenges, and choices

Adopting cloud technologies brings many benefits but also introduces new challenges. This is true regardless of the role in which you work. Architects need to understand the changes imposed by the underlying hardware and learn new infrastructure management patterns. Developers and QA specialists need to explore the opportunities presented by container and cloud technologies and also learn new abstractions for interacting with the underlying infrastructure platforms. And platform engineers need to build and operate a supporting platform to enable developers to code, test, ship, and run applications with speed and safety.

You must establish your goals for moving to the cloud early in the process – ideally, this is the first thing you do. Most successful organizations base their goals on improving some or all of the DORA or Accelerate metrics.

DORA metrics are used by DevOps teams to measure their performance and find out whether they are "low performers" to "elite performers." The four metrics used are deployment frequency (DF), lead time for changes (LT), mean time to recovery (MTTR), and change failure rate (CFR). You want to maximize your deployment frequency while minimizing the other metrics.

Gateway to speed: Establishing abstractions, separation of concerns, and self-service

At Ambassador Labs, we have seen a high correlation between deployment frequency and successful adoption of cloud native principles and technologies. This ability to rapidly ship new software to customers – both for feature releases and incident resolution – adds a lot of value that can be easily understood throughout the organization, from the C-level to the product and engineering and support teams.

The key to this is focusing on providing the correct platform abstractions and embracing a self-service mindset. Take the API gateway use case as an example, there are two key personas involved: the platform engineers, who want to set appropriate guardrails to minimize incidents and maximize their security posture, and the developers, who want to release services and functionality rapidly and configure API endpoints dynamically.

Cloud native API gateway

A cloud native API gateway will enable both of these personas and associated use cases. For example, with Ambassador Edge Stack, we embraced the widely adopted Kubernetes Resource Model (KRM), which enables all of the API gateway functionality to be configured by Custom Resources and applied to a cluster in the same manner as any Kubernetes configuration. For example, using build pipelines or a GitOps continuous delivery process).

We've gone one step further, though, and designed our Custom Resources with the engineering best practice of separation of concerns for platform engineers and developers in mind.

Platform engineers can configure the core API gateway functionality using resources like Listener, Host, and TLSContext. They can also provide a range of authentication and authorization options (using OIDC, JWT, etc) and rate limiting using the Filter resources. Independently from this – although appropriately coupled at runtime – developers can launch new services and APIs using the Mapping resource. They can also augment their API endpoints with required authn/authz policy and rate limiting using the FilterPolicy and RateLimit custom resources.

But don't just take our word for it!

The Ambassador platform lets us treat HTTP and TCP routes like any other Kubernetes object, which means CI/CD can manage them just like deployments or services [...] Operations does not have to get involved in setting up basics like routing, load balancing, and so on, removing development bottlenecks for us. Our developers get to declare just what they need, and the platform makes their application accessible to the world as soon as it's deployed."

Bo Daley
Platform Engineer

Ambassador makes it very easy for us to manage endpoints across all our regions worldwide and is able to seamlessly adapt and work with every region's 80 different endpoints, each with varying configuration requirements."

Nashon Steffen
Staff Infrastructure Development Engineer

Gateway to the future: Becoming cloud native is a journey

In addition to adopting a good separation of concerns and a self-service approach, there are a number of other factors to consider when adopting a cloud native API gateway. The following articles in this series explore each of these considerations in more detail:

- Service discovery: Monoliths, microservices, and meshes
- Loading balancing methodologies
- Load balancing in the cloud
- Adding support for modern protocols like HTTP/3

Service discovery: API gateway and/or service mesh

When adopting a cloud native approach to service connectivity and communication, there is often a recurring question of which technology is preferred for handling how microservice-based applications interact with each other. That is, “should I start with an API gateway or use a Service Mesh?”

When we talk about both technologies, we refer to the end user’s experience in achieving a successful API call within an environment. Ultimately, these technologies can be classified as two pages of the same book, except they differ in how they operate individually. It is essential to understand the underlying differences and similarities between both technologies in software communication.

In this article, you will learn about service discovery in microservices and also discover when you should use an API gateway and when you should use a service mesh.

Kubernetes load balancing methodologies

Load balancing is the process of efficiently distributing network traffic among multiple backend services and is a critical strategy for maximizing scalability and availability. In Kubernetes, there are various choices for load balancing external traffic to pods, each with different tradeoffs.

This article offers a tour de force of various load balancing strategies and implementations, with the goal to help you choose how to get started and how to evolve this as your cloud adoption grows.

Cloudy with a chance of load balancing: AWS EKS and API gateways

We’ve helped thousands of developers get their Kubernetes ingress controllers up and running across different cloud providers. Amazon users have two options for running Kubernetes: they can deploy and self-manage Kubernetes on EC2 instances, or they can use Amazon’s managed offering with Amazon Elastic Kubernetes Service (EKS).

If you are using EKS Anywhere, the recommended ingress and API gateway is Emissary-ingress. Overall, AWS provides a powerful, customizable platform on which to run Kubernetes. However, the multitude of options for customization often leads to confusion among new users and makes it difficult for them to know when and where to optimize for their particular use case.

After working with many customers to configure their ingress controller successfully on AWS EC2 and Amazon EKS, we found a common set of questions that we were asking users. We took those questions and converted them into a series of key decisions that we’ve presented in this article.

Embracing modern protocols: To HTTP/3 and beyond

With HTTP/3 being supported by 70%+ of browsers (including Chrome, Firefox, and Edge), and the official spec being finalized in June 2022, now is the time that organizations are beginning a widespread rollout of this protocol to gain performance and reliability. As leaders in the implementation of the HTTP/3 spec, Google and the Envoy Proxy teams have been working on rolling this out for quite some time, and they have learned many lessons.

HTTP/3 is especially beneficial for users with lossy networks, such as cell/mobile-based apps, IoT devices, or apps serving emerging markets. The increased resilience through rapid reconnection and the reduced latency from the new protocol will benefit all types of Internet traffic, such as typical web browsing/search, e-commerce and finance, or the use of interactive web-based applications, all of which can encounter packet loss of 2%+ on the underlying networks.

This article provides details of the HTTP/3 protocol and highlights the benefits and challenges of adding support for this in your applications. We have also conducted preliminary benchmark tests using the Google Chrome web browser and Ambassador Edge Stack 3.0 to study HTTP/3 and test it against the previous versions of the HTTP protocol.

Adopting a cloud native API gateway: Focus on speed, safety, and self-service

Choosing to become cloud native is a big decision. There are many things to consider, both from an organizational and technical perspective. The fundamentals of API gateway technology have evolved over the past ten years, and adopting cloud native practices and technologies like continuous delivery, Kubernetes, and HTTP/3 adds new dimensions that need to be supported by your chosen implementation.

We recommend you focus on speed, safety, and self-service. You want developers to be able to move fast and out-innovate your competitors. You also want platform engineers to provide guardrails and security for your systems. And critically, you don't want your teams drowning in IT service desk requests and ticket handoffs. Self-service is the only way to move with speed and safety.

Part 2:

Microservice Service Discovery:

API Gateway or Service Mesh?

When managing cloud-native connectivity and communication, there is always a recurring question on which technology is preferred for handling how microservice-based applications interact with each other. That is; “Should I start with an API gateway or use a Service Mesh?”.

When we talk about both technologies, we refer to the end-user’s experience in achieving a successful API call within an environment. Ultimately, these technologies can be classified as two pages of the same book, except they differ in how they operate individually. It is essential to understand the underlying differences and similarities between both technologies in software communication.

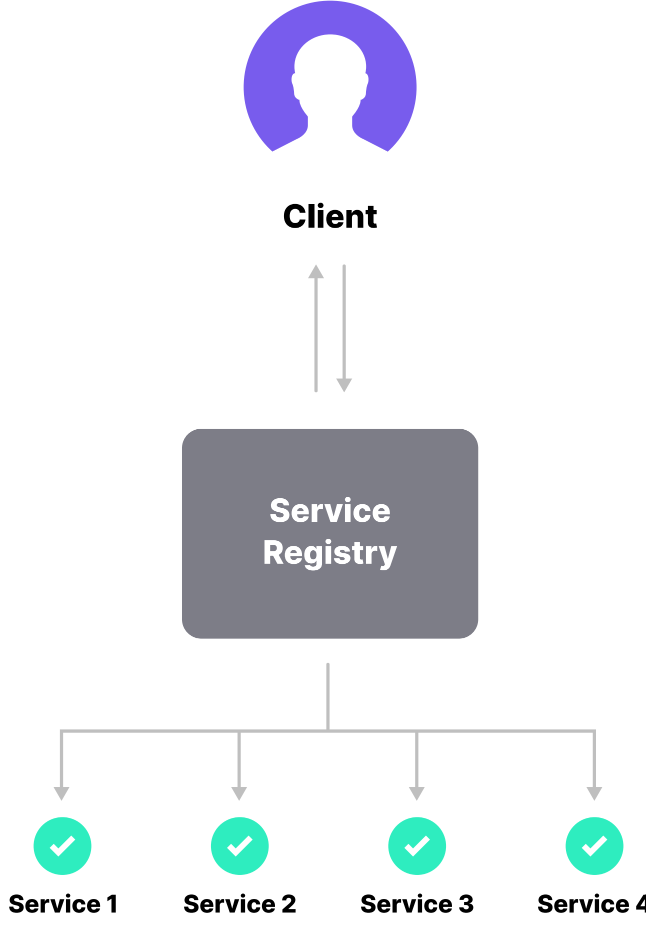
In this article, you will learn about service discovery in microservices, and also discover when you should use a Service Mesh or API gateway.

Introduction

In a microservice architecture, for clients to communicate with the backend, they need a service such as a service mesh or API gateway to relay these API requests. This technology receives the clients' requests and transports them to the back end. As a result of how dynamic the ports to these backend services get, they may change from time to time for different reasons (nodes fail, new nodes are added to the network etc).

The API gateway, however, doesn't know by itself how to identify the particular backend service a client requests, so it forwards the request to another service: called service-discovery. The API gateway asks the service-discovery software (e.g ZooKeeper, HashiCorp Consul, Eureka, SkyDNS) where it can locate different backend services according to API requests (by sending the name). Once the service-discovery software provides the necessary information, the gateway forwards the request to that address.

Before we dive in, let's quickly talk about the service registry, as most of the service discovery concepts are based on it.



What is a service registry?

The service registry is a database that holds the data structures for network service instances. It serves as a messaging system that transports data for application-level communication.

Every service registers itself with the service registry providing all details on where it can be located; this includes host, port, node name, and any other service-specific metadata.

In a situation where the service registry is unavailable, connecting to the microservices might be difficult or impossible. That's why the service registry is expected to be available and updated at all times for clients to get information on network locations.

What is service discovery?

The Microservices architecture is made up of smaller applications that need to communicate with each other through REST APIs constantly.

Service discovery (otherwise known as service location discovery) is how applications and microservices can automatically locate & communicate with each other. It is a fundamental pattern in service architecture that helps track where every microservice can be found. These microservices register their details with the discovery server, making it easy to communicate.

Types of service discovery

There are two types of service discovery patterns you should know about — server-side discovery and client-side discovery.

Let's break them down in detail:

A server-side discovery:

This discovery pattern permits client applications to request a service via a load balancer. The load balancer then queries the service registry before routing the client's request. Think of the server-side discovery like the receptionist (load balancer) that attends to you when you phone an organisation. The receptionist will enquire about the details of the person you wish to communicate with and redirect your call to the person.

A client-side discovery:

With the client-side discovery, the client is responsible for selecting network services available; by querying the service registry. It then proceeds to use the load balancing algorithm to select available service instances and requests. This is similar to how we interact with our search engines — you search for a topic on your browser and your browser (service registry) will search and return a list of URLs and port numbers. As a user, you will look for URLs that provide accurate answers to the request you made, then select the preferred URL that meets your demands.

Now that we understand what service discovery is, let's look at what API gateway and service mesh are, and which is preferred for your microservice architecture.

What is an API gateway?

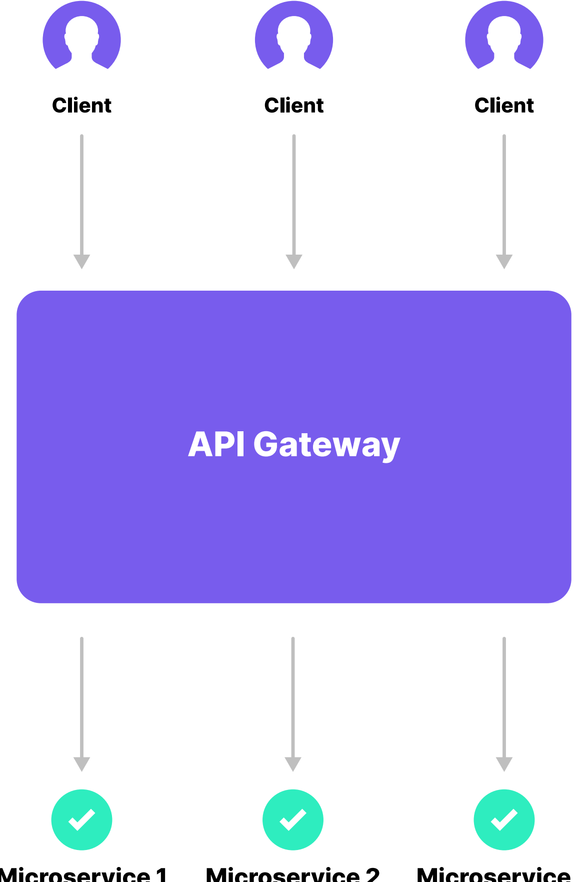
An API gateway is a management service that accepts API requests from clients, directs the requests to the correct backend services, aggregates the results retrieved, and returns a synchronous response to the client.

To better understand the meaning of an API gateway, consider an e-commerce site where users invoke requests to different microservices like a shopping cart, checkout, or user profile. Most of these requests trigger API calls to more than one microservice, and due to the vast number of API calls that are made to the backend, an API gateway acts as a mid-layer between the clients and the services and retrieves all product details with a single request.

Developers can encode the API gateway features within the application to execute such tasks, without having to use an API gateway. However, that would be a tedious task for the developers to take on. This method also poses security risks of exposing the API to unauthorized access.

In essence, an API gateway help to simplify communication management such as API requests, routing, composition, and balancing demands across multiple instances of a microservice. It can also perform log tracing and aggregation without disrupting how API requests are handled.

Examples of API gateways in the cloud native ecosystem include: Ambassador Edge Stack, Apigee, Amazon API Gateway, MuleSoft, Kong, Tyk.io, Nginx, etc.



What is a service mesh?

A service mesh is an infrastructure layer that handles how internal services within an application communicate. It adds microservice discovery, load balancing, encryption, authentication, observability, security, and reliability features to "cloud-native" applications making them reliable and fast.

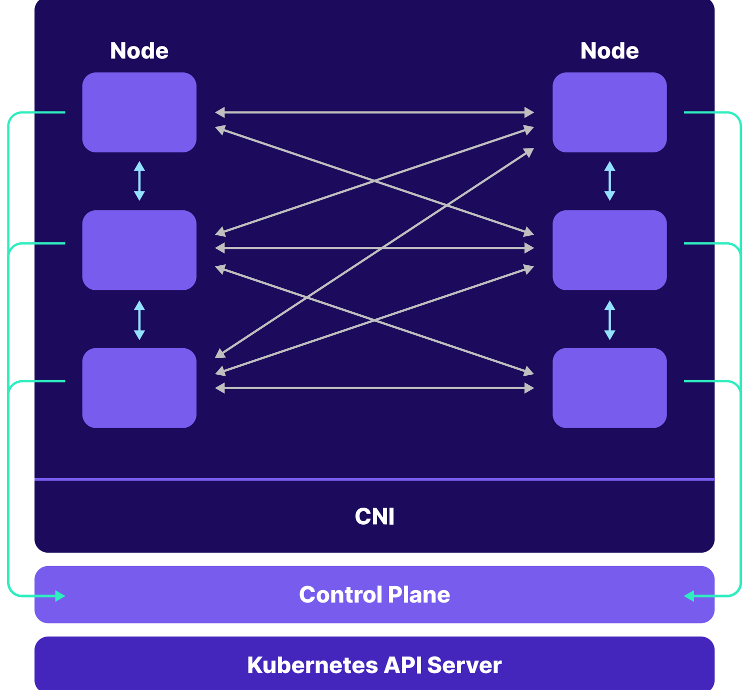
Fundamentally, service meshes allow developers to create robust enterprise applications by handling management and communication between multiple microservices.

It is usually implemented by providing a proxy instance, called a sidecar for service instances. These proxies handle inter-service communications and act as a point where the service mesh features are introduced.

Returning to the e-commerce illustration used earlier, let's imagine the user proceeds to check out their order from the shopping cart. In this case, the microservice retrieving the shopping cart data for checkout will need to communicate with the microservice that holds user account data to confirm the user's identity. This is where the service mesh comes into play! It aids the communication between these two microservices thereby ensuring the user's details are confirmed correctly from the database.

Just like API gateways, service mesh features can also be hardcoded into an application. However, this will be a tedious job for the developers as they might be required to modify application code or configuration as network addresses change.

Examples of service meshes in the cloud native ecosystem include: Linkerd, Kuma, Consul, Istio, etc.



Similarities and differences between a service mesh and API gateway

Implementing an API gateway or a service mesh for enterprise-level application development is a recurring question amongst developers.

This section of the article will help you understand the differences and similarities between them and help you decide which to go with.

Similarities between an API gateway & a service mesh:

Resilience: With either or both technologies in place, your application can recover quickly from difficulties or failures encountered in your cloud-native application.

Client-side discovery: In both the API gateway and service mesh, the client is responsible for requesting and selecting available network services.

Traffic management: Without a service mesh or API gateway in place, the traffic from API calls made by clients would be difficult to manage. This will eventually delay the request processing and response time.

Service discovery: Both technologies facilitate how applications and microservices can automatically locate and communicate with each other.

System observability: Both technologies can manage services that can be accessed by clients. They also keep logs of clients that have accessed specific services. This helps to track the health of each API call made across to the microservice.

Differences between an API gateway & a Service mesh

Capabilities: API gateways serves as an edge microservice and perform tasks helpful to your microservice's business logic, like request transformation, complex routing, or payload handling, while the service mesh only addresses a subset of inter-service communication problems.

External vs. internal communication: A major distinction between these technologies is their operation. The API gateway operates at the application level, while the service mesh operates at the infrastructure level. An API gateway stands between the user and internal applications logic, while the service mesh stands between the internal microservices. As discussed above, API gateways focus on business logic, while service mesh deals with service-to-service communication.

Maturity: API gateways are a more established technology. Based on how popular this technology has grown, there are many vendors of API gateways. In comparison, the service mesh is a new and nascent open source technology with very few vendors today.

Tooling and support: API gateways work with almost every application or architecture, and can work with monolithic and microservice applications. Service mesh is typically designed only to work in specific environments, such as Kubernetes. Also, API gateways have automated security policies, and features that are easy to get started with; service meshes often have complex configurations and processes that have a steep learning curve.

Monitoring and observability: API gateways can help you track the overall health of an application by measuring the metrics to identify flawed APIs. Meanwhile, service mesh metrics assist teams in identifying issues with the various microservices and components that make up an application's back end rather than the entire program. Service mesh helps in determining the cause of specific application performance issues.

Can an API gateway & service mesh co-exist?

Both technologies have so many things in common, but their significant difference lies in how they operate. The API gateway is a centralized control plane that works at the application level, managing traffic from edge level client-to-service. The device mesh operates on the infrastructure level, dividing application functionality into microservices & managing internal service-to-service communication.

When combined with a service mesh, the API gateway can operate as a mediator. This can improve delivery security and speed, ensuring application uptime and resiliency while ensuring your applications are easily consumable. This will, in turn, bring additional functionality to your application stack.

Simplified Kubernetes management with Ambassador Edge Stack API Gateway

Routing traffic into your Kubernetes cluster requires modern traffic management. And that's why we built Ambassador Edge Stack to contain a modern Kubernetes ingress controller that supports a broad range of protocols, including HTTP/3, gRPC, gRPC-Web, and TLS termination.

Ambassador Edge Stack provides traffic management controls for resource availability. Try Ambassador Edge Stack today or learn more about Ambassador Edge Stack.

Conclusion

API gateways and service meshes overlap in several ways, and when these technologies are combined, you get a great end-to-end communication experience.

To maximize the agility of your application and minimize the effort developers spend on managing communications, you may need both a service mesh and an API gateway for your application.

Part 3:

Load balancing strategies in Kubernetes:

L4 round robin, L7 round robin, ring hash, and more

What is loading balancing in Kubernetes?

Load balancing is the process of efficiently distributing network traffic among multiple backend services, and is a critical strategy for maximizing scalability and availability. There are a variety of choices for load balancing Kubernetes external traffic to Pods, each with different tradeoffs.

Selecting a load balancing algorithm should not be undertaken lightly, especially if you are using application layer (L7) aware protocols like gRPC. It's all too easy to select an algorithm that will result in a single web server running hot or some other form of unbalanced load distribution.

Let's explore these in more detail.

L4 Round Robin Load Balancing with kube-proxy

In a typical Kubernetes cluster, requests that are sent to a Kubernetes Service are routed by a component named kube-proxy. Somewhat confusingly, kube-proxy isn't a proxy in the classic sense, but a process that implements a virtual IP for a service via iptables rules. This architecture adds additional complexity to routing. A small amount of latency is introduced for each request which increases as the number of services grows.

Moreover, kube-proxy routes at Layer 4 (L4), i.e., TCP, which doesn't necessarily fit well with today's application-centric protocols. For example, imagine two gRPC clients connecting to your backend Pods. In L4 load balancing, each client would be sent to a different backend Pod using round robin load balancing. This is true even if one client is sending 1 request per minute, while the other client is sending 100 requests per second.

So why use kube-proxy at all? In one word: simplicity. The entire round robin load balancing process is delegated to Kubernetes, the default strategy. Thus, whether you're sending a request via Ambassador Edge Stack or via another service, you're going through the same load balancing mechanism.

kube-proxy and IPVS

While kube-proxy uses iptables for routing by default, kube-proxy can also use IPVS (IP Virtual Server). The advantage of IPVS over iptables is scalability: no matter how many routing rules are required (which are directly proportional to the number of services), IPVS runs in O(1) time. Thus, for clusters that consist of thousands of services, IPVS is generally a preferred option. That said, IPVS-based routing is still L4-level routing and is subject to the constraints listed above.

This brings us to layer 7 (L7) routing for load balancing Kubernetes traffic, which we will discuss next.

L7 round robin load balancing

What if you're using a multiplexed keep-alive protocol like gRPC or HTTP/2, and you need a more fair round robin algorithm? You can use an API Gateway for Kubernetes such as Ambassador Edge Stack, which can bypass kube-proxy altogether, routing traffic directly to Kubernetes Pods. Ambassador is built on Envoy Proxy, a L7 proxy, so each gRPC request is load balanced between available Pods.

In this approach, your load balancer will typically use the Kubernetes EndpointsSlices API to track the availability of Pods. In older versions of Kubernetes the Endpoint API can be used instead. When a request for a particular Kubernetes service is sent to your load balancer, the load balancer round robins the request between Pods that map to the given service.

Ring hash

Instead of rotating requests between different Pods, the ring hash load balancing strategy uses a hashing algorithm to send all requests from a given client to the same Pod. The ring hash approach is used for both "sticky sessions" (where a cookie is set to ensure that all requests from a client arrive at the same Pod) and for "session affinity" (which relies on client IP or some other piece of client state).

The hashing approach is useful for services that maintain per-client state (e.g., a shopping cart). By routing the same client to the same Pod, the state for a given client does not need to be synchronized across Pods. Moreover, if you're caching client data on a given Pod, the probability of cache hits also increases.

The tradeoff with ring hash is that it can be more challenging to evenly distribute load between different backend servers, since client workloads may not be equal. In addition, the computation cost of the hash adds some latency to requests, particularly at scale.

Maglev

Like ring hash, maglev is a consistent hashing algorithm. Originally developed by Google, maglev was designed to be faster than the ring hash algorithm on hash table lookups and to minimize memory footprint. The ring hash algorithm generates fairly large lookup tables that do not fit onto your CPU processor cache.

For microservices, Maglev has one fairly expensive tradeoff: generating the lookup table when a node fails is relatively expensive. Given the transient nature of Kubernetes Pods, this may not work. For more details on the tradeoffs of different consistent hashing algorithms, this article covers consistent hashing for load balancing in detail, along with some benchmarks.

Learning More

The networking implementation within Kubernetes is more complex than it might first appear and somewhat more limited than many engineers understand. Matt Klein put together a very informative blog post in 2017 that stands the test of time "Introduction to modern network load balancing and proxying". This provides a great foundation for understanding key concepts.

A series of additional posts explain why organizations have chosen to use Layer 7 aware proxies to load balance ingress traffic, such as Bugsnag, Geckoboard, and Twilio.

Ambassador Edge Stack API Gateway

Built on Envoy Proxy, Ambassador Edge Stack is a Kubernetes native API Gateway that supports all of the methods for load balancing Kubernetes traffic discussed above. Visit the Ambassador Labs website or join our Slack channel for more information.

Part 4:

Configuring Kubernetes Ingress on AWS? Don’t Make These Mistakes

Key considerations for developers looking to configure an ingress controller on AWS or Amazon EKS

What is Ingress in AWS?

Kubernetes Ingress is an API resource that allows you to manage external or internal HTTP and HTTPS access to Kubernetes Services running in a cluster. AWS provide several load balancer types that can be used in conjunction with a Kubernetes ingress, and these include both transport-based layer 4 (L4) and application-based layer 7 (L7) options.

Configuring K8s Ingress in AWS

We've helped thousands of developers get their Kubernetes Ingress Controllers up and running across different cloud providers. Amazon users have two primary options for running Kubernetes on AWS: you can deploy and self-manage Kubernetes on EC2 instances, or you can use Amazon's managed offering with Amazon Elastic Kubernetes Service (EKS).

If you choose EKS, you can either run this within the AWS public cloud platform, or use EKS Anywhere, which allows you to create and operate Kubernetes clusters on your own infrastructure, supported by AWS. The default Ingress solution for AWS EKS is Emissary-ingress.

Overall, AWS provides a powerful, customizable platform on which to run Kubernetes. However, the multitude of options for customization often leads to confusion among new users and makes it difficult for you to know when and where to optimize for your particular use case.

After working with many customers to configure their AWS Ingress Controller successfully on EC2 and Amazon EKS, we have found a common set of questions that we were asking users. We took those questions and converted them into a series of key decisions that we've presented here.

If you're struggling to configure Kubernetes Ingress on AWS, here's our recommended consideration path.

Choose the Right Load Balancer Type

The most important choice you will make when deciding how to handle ingress in AWS is the type of load balancer you want to use. The other major cloud providers make this easy by not providing so many options. Configuring a "type: LoadBalancer" Service in several other providers always gives the same L4 load balancer.

In AWS, a "type: LoadBalancer" Service in Kubernetes can mean a classic Load Balancer in L4 or L7 (called an Elastic Load Balancer or ELB) or a Network Load Balancer (NLB). Additionally, users can also manually provision an Application Load Balancer and point it at their Ingress exposed as a "type: NodePort" Service.

Layer 4 Load Balancers: ELB & NLB

The L4 ELB and NLB are layer 4 load balancers which route requests to your AWS Ingress Controller at the TCP layer.

This means that they are typically very efficient but can be limited in the types of traffic they can route and how intelligently they are routing requests to your Ingress Controller. For example, the L4 ELB is widely deployed in long-live AWS deployments but it cannot handle WebSockets connections. The NLB is the fastest and most efficient AWS load balancer, but it cannot load balance to multiple Kubernetes cluster namespaces.

All L4 load balancers are limited to round robin load balancing algorithms. They are also limited in their ability to preserve information about the client to the Ingress Controller.

Layer 7 Load Balancers: ELB & ALB

The L7 ELB and ALB are layer 7 load balancers which route requests to your AWS Ingress Controller at the "application" protocol layer. This means that they are able to more intelligently decide how to route requests, but are typically less efficient. For example, the ALB can route requests based on information sent in the request, such as the Host or URL path.

While this is powerful, if your application is living in Kubernetes and your load balancer is just routing requests to your Ingress Controller, you typically do not need this level of control over how you are routing requests.

Another benefit of L7 load balancers is their ability to preserve information about the client in the X-Forwarded headers.

Committing to an AWS Load Balancer Type in Kubernetes

Before you determine which type of load balancer is best for your use case, you'll want to consider these four key criteria:

1. Where to terminate TLS
2. How to manage certificates
3. Where to terminate TLS
4. Where to terminate TLS

Consideration #1:

TLS Termination

TLS encryption is a common requirement for modern web apps. Users want to be sure they are communicating with the intended recipient without anyone intercepting or modifying their requests. If you want to encrypt connections you need to terminate TLS at an entrypoint into your application.

Since AWS allows you to terminate TLS at any of the four load balancers available, deciding where to terminate TLS is dependent on your choice of load balancer.

- L7 load balancers are required to terminate TLS so they can read information from the request.
- L4 load balancers are able to perform SSL passthrough, which allows your AWS Ingress Controller to terminate TLS.

If you choose to terminate TLS at your load balancer, your Ingress Controller will receive traffic over clear text, which creates another trade-off: L7 load balancers can inform your Ingress Controller of whether the request originated encrypted by setting "X-Forwarded-Proto" whereas L4 load balancers cannot.

If you choose to terminate TLS at your Ingress Controller, you can fully control and manage TLS certificates, Server Name Indication (SNI) for multiple host/domain name support, and also how connections are encrypted and unencrypted.

Consideration #2:

Certificate Management

How TLS certificates are managed in AWS is dependent on where you are terminating TLS. If you are terminating TLS at the load balancer, you can use Amazon Certificate Manager (ACM) to manage your TLS certificates.

If you are terminating TLS at your AWS Ingress Controller, then your Ingress Controller is responsible for how it manages TLS certificates. Some Ingress Controllers, such as Ambassador Edge Stack, can automatically manage certificates whereas others require that you use other tools, like cert-manager, or store and rotate certificates in Kubernetes manually.

Consideration #3:
Cleartext Redirection

While most modern web apps want TLS encryption, many users will still make unencrypted requests to your application. Therefore, it is important for your application to be able to properly handle these unencrypted requests. So, your next decision is how your Ingress Controller will be configured to handle cleartext.

If you chose to terminate TLS at your AWS Ingress Controller or at an L7 load balancer, your Ingress Controller will be able to identify if the request arrived over an encrypted connection or not, and fully manage if you want to allow, deny, or automatically redirect cleartext traffic to an encrypted connection.

If you choose to terminate TLS at an L4 load balancer, however, you are forced to route both cleartext and encrypted connections.

Consideration #4:
Preserving Client Information

The final consideration when choosing how to handle ingress in AWS is if you need to preserve information from the client. Like all of the other decisions, this choice depends on the load balancer you are using.

L7 load balancers easily preserve this information by appending to the X-Forwarded-For header. This passes the IP address of your client to your AWS Ingress Controller and upstream services.

L4 load balancers cannot preserve this information in the same way. Instead, the ability to do this requires some tradeoffs.

In AWS, there are two ways for L4 load balancers to preserve the client IP address.

1. The HAProxy Protocol gives L4 proxies the ability to append the client IP address by wrapping the request with additional data. This, however, requires your Ingress Controller expect requests set the proxy protocol which means all requests must carry this extra data. This is okay when all requests go through the load balancer, but this can present difficulties if sending requests directly to the Ingress Controller

2. Configuring Kubernetes to force connections only to Nodes running your Ingress Controller. This configures this so that the L4 load balancer is always connecting directly to the Ingress Controller instead of via Kubernetes networking. However, this causes stability issues when restarting and upgrading your Ingress Controller, as well as causing more uneven load balancing to your Ingress Controller pods.

Wrapping Up: AWS Ingress Options

After helping hundreds of users configure our Ambassador Edge Stack API Gateway to run effectively in AWS, we became acutely aware of how confusing this process can be.

To help developers easily configure an Ingress Controller and get this up and running in AWS faster, we have provided extension documentation on "Ambassador Edge Stack with AWS".

If you need help, please reach out and ask questions in our community on Slack or contact the sales engineering team.

Part 5:

HTTP/3

Use Cases, Envoy Support, and Google's Rollout

With HTTP/3 being supported by 70%+ of browsers (including Chrome, Firefox, and Edge), and the official spec being finalized in June 2022, now is the time that organizations are beginning a widespread rollout of this protocol to gain performance and reliability.

As leaders in the implementation of the HTTP/3 spec, Google and the Envoy Proxy teams have been working on rolling this out for quite some time, and they have learned many lessons.

Lessons Learned by Google's Rollout of HTTP/3

Alyssa Wilk, Senior Staff Software Engineer at Google, recently spoke with Daniel Bryant, Head of DevRel at Ambassador Labs. In a wide-ranging discussion that covered how HTTP/3 has been implemented over QUIC and UDP, the benefits and challenges offered by the new protocol, and the experience of Google publicly rolling out support for this protocol, a number of key themes emerged:

- HTTP/2 sped up HTTP/1 dramatically – but if you lose one packet on a connection, everything gets stalled until the packet is retransmitted.
- This is a fundamental limitation of TCP, so HTTP/3 speeds up HTTP/2 even more by implementing the protocol on top of UDP.
- The two big wins in HTTP/3 are the zero roundtrip handshake and improved congestion control. With the former, if you have already connected to the server previously you can bypass the three-way TCP handshake. With the latter, if you drop a packet, HTTP/3 will recover better and faster than HTTP/2.
- Moreover, because HTTP/3 is implemented in userspace, you get these performance benefits even if you haven't updated (or can't update) your operating system kernel.
- Because there's on average 2% packet loss on the Internet, HTTP/3 benefits virtually everyone.
- End users who see even more benefit are those on lossier networks (e.g., emerging markets, mobile, IoT use cases) and those on old kernels (e.g., Windows users at large companies that don't upgrade).
- Adding HTTP/3 support to a proxy, ingress, or API gateway is non-trivial (unlike HTTP/2) as the protocol has very sophisticated congestion control and cryptography that needs to be implemented.

Getting Started with HTTP/3 and Edge Stack

You can get started with implementing HTTP/3 with our guide, "How to Implement HTTP/3 Support with Ambassador Edge Stack 3.0". Using Ambassador Edge Stack (the implements the Envoy Proxy HTTP/3 support) and a simple web app deployed into Kubernetes, you will be able to get started with HTTP/3 in under 5 minutes.

Listen to the Full HTTP/3 Podcast with Alyssa Wilk

SOUNDCLOUD

Want to learn more?

Get started with
Ambassador Edge Stack

Get started

Learn more via the
Kubernetes Learning Center

Learn more

Sign up for the **Kubernetes Developer Accelerator Program**

Sign up

ambassador