O'REILLY®

Compliments of solo

Sidecar-less Istio Explained

Lowering the Barrier to Service Mesh Adoption with Ambient Mode

Lin Sun and Christian Posta

REPORT

SOLO.IO

Deploy Istio ambient mesh in production with Gloo Mesh.

Enterprise-grade service mesh.
Powered by open source.

Find out more
about Ambient Mesh

www.ambientmesh.io

Sidecar-less Istio Explained

Lowering the Barrier to Service Mesh Adoption with Ambient Mode

Lin Sun and Christian Posta

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Sidecar-less Istio Explained

by Lin Sun and Christian Posta

Copyright © 2024 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 (*https://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: Gregory Hyman
Copyeditor: nSight, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

September 2024: First Edition

Revision History for the First Edition

2024-09-20: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Sidecar-less Istio Explained*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors 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.

978-1-098-17802-4

[LSI]

Table of Contents

Foreword. v

1. Introducing Istio Ambient Mode. 1

Current Challenges 1
Benefits of Istio Ambient Mode 2
Additional Benefits 3
What About the Sidecar? 4

2. Istio Ambient Architecture. 5

Istio Ambient Mode 5
Ambient Is Designed for Scale and Security 13
Wrapping Up 15

3. Exploring Istio Ambient Mode. 17

The Ambient Profile 17
The Benefits of Running Your Applications in Ambient 19
Incrementally Adopting Ambient 21
Understanding How Istio Ambient Architecture Works 30
Wrapping Up 33

4. Key Takeaways and Next Steps. 35

Takeaways 35
Next Steps 36

Foreword

Istio has come a long way since the early days of the project. It's now a graduated CNCF project and ranks as the third-most active project in that ecosystem among some pretty impressive peers. Istio is solving big problems for users in just about every kind of industry imaginable. Istio was founded on a highly principled approach to distributed zero-trust security, and much of its adoption growth is tied to the demand for stronger security in enterprises. Zero trust is no longer an unreachable pipe dream but a roadmap expectation for most CSOs.

Like any technology, Istio must adapt to its users, and the introduction of "ambient mesh" is the largest evolutionary step the project has taken since the 1.0 release. We've learned a lot as a community over the last seven years about the good, bad, and ugly of service mesh. We often like to talk about how we're trying to make Istio delightful and boring at the same time. We chose the name "ambient" to evoke the idea that a service mesh should be part of the underlying infrastructure, something that is delightfully powerful when you need it to do something but that is otherwise just part of the networking wallpaper.

This book is an exploration of the critical ideas behind ambient mesh, how they are made reality, and how to use ambient mesh to solve real-world problems. Lin Sun and Christian Posta have been with the project since the very beginning, and their wealth of insight and experience with production use makes them ideally qualified to write this guide. If you are responsible for platform

infrastructure, have been tasked with getting your organization to a zero-trust posture, or are wondering how ambient mesh can make your application more reliable, this book is for you.

— Louis Ryan
CTO at Solo.io and cocreator of Istio

CHAPTER 1

Introducing Istio Ambient Mode

Istio ambient mode is a sidecar-less data plane for Istio service mesh originally developed by Solo.io and Google. The goal for Istio ambient is to improve the operational experience of adopting, deploying, upgrading, and generally managing Istio throughout its life as critical infrastructure. Additional benefits over Istio's sidecar deployments include resource cost savings, performance improvements, and improved security while maintaining Istio's core feature set of zero-trust security, resilience, observability, traffic routing, and policy enforcement.

Current Challenges

Before we get too deep into what Istio ambient mode is and how it works, we should understand the motivation for its creation. Using sidecars to implement mesh functionality has been battle-tested and used successfully at scale to provide a lot of value. So why introduce an alternative approach?

We (the creators of Istio) have always intended to make the service mesh transparent and incrementally adoptable, but in practice, the sidecar approach has had drawbacks.

For example, injecting a sidecar proxy is an invasive operation. To enable this, the application needs to be restarted once auto-injection has been enabled. The application will need to know it's there to properly size things like memory and CPU. Scaling out across large numbers of workloads increases the memory and CPU overhead of

running the sidecar. This can get very expensive at scale. From a security standpoint, if the application is compromised, the attacker will have access to the sidecar and secret key material. If a malicious user wants to opt out of the capabilities provided by the mesh, they can work around the sidecar.

Injecting a proxy and tying the application lifecycle to the infrastructure lifecycle causes transparency to be lost. Deployment descriptors that go through continuous integration/continuous deployment (CI/CD) may not be exactly what is deployed in production when auto-injecting a sidecar proxy. This is because auto-injection mutates the pod spec that happens after deployment. Additionally, when upgrading Istio's data plane, applications need to be redeployed or restarted. This churn in the cluster needs to be coordinated to avoid unplanned outages.

Another drawback is the adoption of Istio features is an "all-or-nothing" proposition. For example, users who wish to adopt mutual transport layer security (mTLS) for compliance reasons must inject a sidecar proxy to get mTLS, but that proxy also implements complex Layer 7 (L7) handling (retries, traffic splitting, complex load balancing, observability collection, etc.). This co-location of features introduces the risk of unexpected behaviors. Istio is a powerful service mesh with many capabilities, but the current adoption curve can be very steep without the ability to adopt features incrementally and absorb risk.

These challenges make it harder to adopt a service mesh at scale, cause turbulence in a rollout of mesh functionality, and introduce L7 behavior risk that may be unnecessary for the features being adopted. Istio ambient mode aims to solve this transparency and incremental adoption problem while introducing cost, performance, and security improvements.

Benefits of Istio Ambient Mode

Istio ambient mode addresses the challenges of transparency and incremental adoption by introducing a sidecar-less data plane and splitting the behaviors of the mesh into two separate layers, each handling concerns that can be combined to provide the full features of the service mesh.

By removing the sidecar from the application pod, workloads are no longer susceptible to container race conditions caused by injecting a sidecar proxy. Job workloads, which can otherwise stay around because of the sidecar, now run to completion correctly. Additional benefits seen with this model include making it harder for workloads to go around the data plane (ignoring the sidecar, forgetting to inject the sidecar, maliciously removing the sidecar, etc.).

Application onboarding and adoption are easier when you don't need to add a sidecar proxy to the workload. For example, applications that may already be running can be dynamically added to the service mesh by applying labels either per workload or at the namespace level. Workloads can be similarly removed from the mesh dynamically, which makes experimentally or incrementally adding workloads to the mesh possible without disturbing the pods.

Upgrades become easier and safer when no data plane component is intermingled with the application. The data plane components can be upgraded independently without restarting the workloads or many fleets of workloads.

Istio ambient mode splits the data plane into two layers: the secure overlay layer and the L7 waypoint proxy layer. We'll discuss each layer in more detail in Chapter 2. This layering approach allows incremental adoption of mesh features. For example, users who want to adopt zero-trust networking properties may opt to use mTLS. By starting with Istio ambient mode's secure overlay layer, which handles only Layer 4 (L4) behaviors of the network like establishing mTLS and telemetry collection, users can adopt the features they want without introducing unnecessary risk.

Additional Benefits

We originally designed Istio ambient for ease of adoption, operations, and maintenance; however, the design of the data plane provides users with a number of additional benefits:

- Better resource usage for data plane components because there are fewer proxies
- Reduced cost of running a service mesh (by at least 90%) as a result of lower resources

- Right-size scaling of L7 proxies based on traffic and not over-provisioning
- Better performance for workloads needing only mTLS because all L7 processing is bypassed
- Separation of application code from the data plane for security improvements
- Better support for server-send-first protocols and nonconformant HTTP implementations

What About the Sidecar?

The sidecar implementation of Istio service mesh has worked well enough, but as discussed previously, it can be improved. Istio ambient mode is steadily marching toward GA sometime in the second half of 2024 (beta was released in May 2024). Remember, the beta label in Istio has typically signaled the project will support production usage, while GA indicates significant production usage and hardening. At some point after GA, ambient mode will become the default. Sidecar implementation will still be available, but the project will recommend that users migrate. If you're looking at Istio service mesh, starting with Istio ambient mode is the recommended approach per the project maintainers.

CHAPTER 2

Istio Ambient Architecture

Istio ambient mode implements a "sidecar-less" architecture that is transparent to the workloads in the mesh. This approach has a number of benefits, like ease of adoption, improved operations, infrastructure cost savings, and more, as discussed in Chapter 1. In this chapter, we'll dig into the architecture of ambient mode and understand how the various components work together to provide mesh functionality like resilience, observability, security, and policy enforcement.

Istio Ambient Mode

The main difference between the Istio sidecar and ambient mode is the data plane. As the ambient name suggests, the data plane attempts to be completely transparent and fade away "into the network":

- Istio ambient does not use sidecars.
- Istio ambient separates out L4 capabilities from L7.
- L4 can be used independently or combined with L7.

Applications are not explicitly aware of the ambient data plane and are not injected with a sidecar or init container. No part of the data plane exists within the application pods. Additionally, pods do not need to be restarted or deal with obscure container race conditions that can cause network failures.

In the Istio ambient architecture, the data plane is explicitly separated into two layers: a secure overlay layer that handles L4, and a waypoint proxy layer that handles L7, as shown in Figure 2-1. As mentioned in Chapter 1, you will want to adopt the features of a service mesh incrementally and absorb risk commensurate with the features you use. Since a very common starting point for adopting service mesh is driven by the need for zero-trust networking and compliance, mTLS is a very popular initial feature. By splitting out the mesh into two layers, users can adopt mTLS without the risks of L7 handling (or mishandling) and later adopt L7 capabilities as needed.

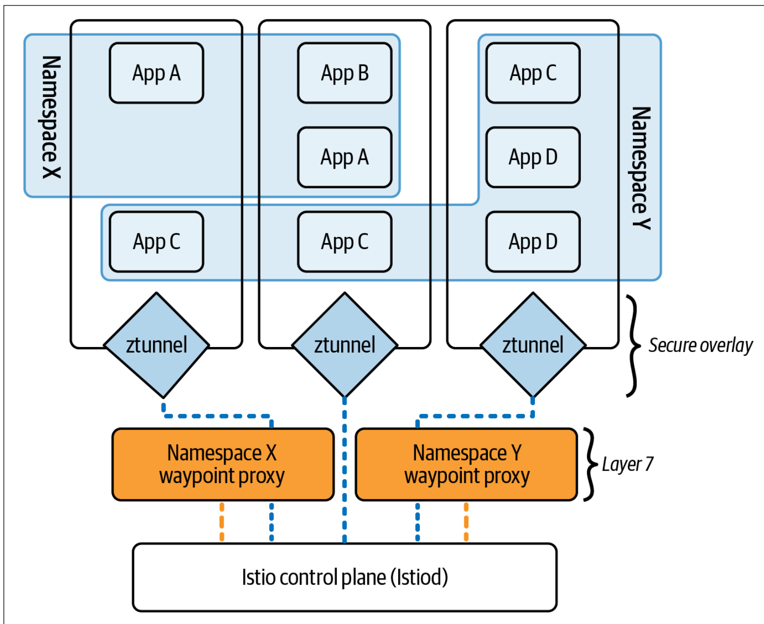

Figure 2-1. Logical view of Istio ambient mode consisting of different layers to handle specific areas of functionality

This split allows users to adopt features incrementally, but it also creates an opportunity to optimize the data path for service-to-service communications. L7 processing can be expensive, and the sidecar approach forces L7 processing on both sides of the connection even if not using L7 functionality. With the ambient data plane mode, we can improve performance by skipping any L7 capabilities and keep traffic only in the secure overlay layer.

Secure Overlay Layer

The secure overlay layer of ambient mode is responsible for establishing secure connections using strong identity between mesh workloads. It does this by leveraging a CNI plug-in and a component called the *ztunnel*. The ztunnel specifically handles collecting L4 telemetry, opening connections, and establishing mTLS with workload identity cryptography.

The ztunnel gets deployed as a DaemonSet on Kubernetes and is a lightweight, purposely built proxy with Rust. As a node proxy, the ztunnel becomes shared among all of the workloads that run on the respective node. In many ways, the ztunnel component becomes an extension of the CNI.

If a connection from a workload destined for another workload does not have any L7 capabilities, the ztunnel will be smart enough to keep the traffic only in the secure overlay layer, as shown in Figure 2-2. If the workload is deployed on another node in the cluster, the ztunnel will tunnel the connection to the remote ztunnel on the destination node where the workload is deployed. From there, the ztunnel on the target node will forward the connection to the target workload.

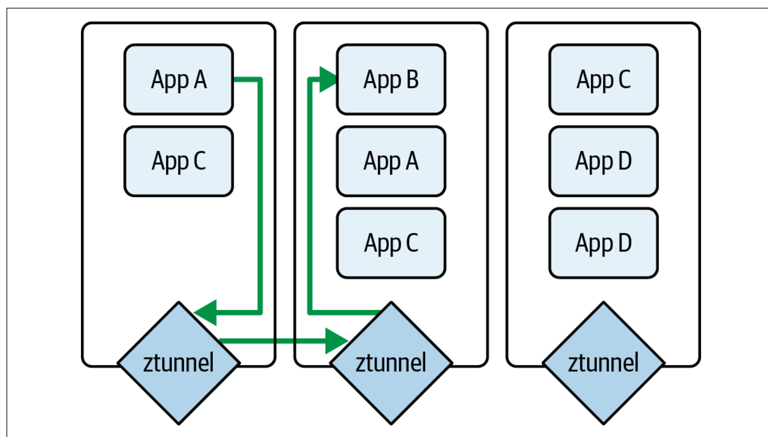

Figure 2-2. The ztunnel will handle all traffic at L4

NOTE

If source and destination pods are located on the same node, the traffic between them is encrypted using mTLS and any L4 policies are enforced by the ztunnel on the node. The ztunnel serves as source and destination ztunnel in this case, providing the secure overlay layer similarly as source and destination pods are placed on different nodes.

Waypoint Proxy Layer

L7 service mesh capabilities are implemented with waypoint proxies in the architecture of Istio ambient mode, as shown in Figure 2-1. This data plane fully parses the connection into requests and can apply policies based on properties like headers and credentials found in the request. L7 functionality includes things like:

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

Waypoint proxies are implemented using the Envoy proxy and deployed per namespace by default. They can be scaled independently depending on the request load to services within the namespace. You can think of these waypoint proxies as individual gateways for their namespaces, as shown in Figure 2-3.

Compared to the sidecar deployment architecture, the ambient architecture allows you to independently scale the waypoint proxy layer to better fit the incoming traffic for services within the specific namespace. For example, you may have 10 instances of App A, and there is a load of 100 requests per second to App A (so approximately 10 requests per second per instance). In the sidecar scenario, there would be 10 sidecar proxies deployed with each instance of App A. You may also have 20 instances of App B serving a load of 200 requests per second in the same namespace. With Istio ambient, you could deploy two waypoint proxies for App A's and App B's residing namespaces to handle the 100 requests over the 10 App A

instances and the 200 requests over the 20 App B instances. This gives a better fit of service mesh data plane to actual workload usage versus the brute force approach of deploying a proxy per instance.

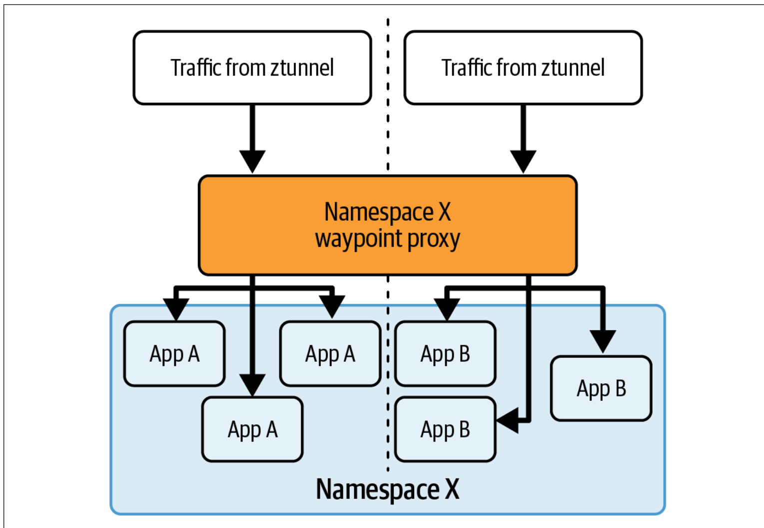

Figure 2-3. The waypoint proxy is deployed per namespace and can be thought of as a "gateway per namespace"

Waypoint proxies get deployed by namespace owners, platform owners, or through automation. When a waypoint proxy is deployed, and the destination is configured to use the waypoint proxy, the secure overlay layer will route the connection to the destination's L7 waypoint proxy, as shown in Figure 2-4.

The characteristics of tenancy for L7 are similar in the Istio ambient mode to a sidecar deployment. L7 capabilities are not shared for multiple tenancies in a single L7 proxy. By default, you can deploy a waypoint based on namespace, where Apps A and B share the same waypoint proxy. In some scenarios, a namespace-based waypoint may not be granular enough. For example, if you only want a waypoint proxy for App A and not App B, you can deploy a waypoint proxy for App A instead of a namespace waypoint proxy. Or if you simply want App A to have its dedicated waypoint proxy so it is not impacted by noisy neighbors in the same namespace, you can deploy a dedicated waypoint proxy for App A in addition to the namespace waypoint proxy.

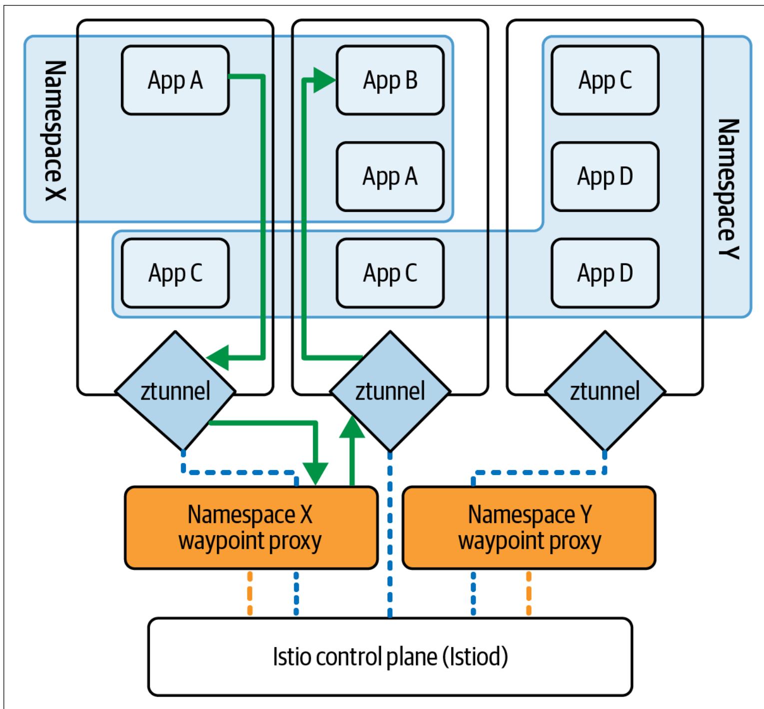

Figure 2-4. Traffic will flow through L7 waypoint proxies to enforce any potential L7 policies

Istio CNI Plug-in

Istio currently has a CNI plug-in (called *istio-cni node agent*) that handles traffic redirection for sidecars. Istio ambient mode extends this CNI plug-in for scenarios where there is no sidecar and the secure overlay layer running on the node needs to handle the traffic (both incoming and outgoing). In ambient mode, the Istio CNI plug-in detects any workload on the node as it is added to ambient mode (as designated by labels—see Chapter 3), enters the workload pod's network namespace, and establishes the network redirection rules inside the pod's network namespace (also called *in-pod*) so that all traffic to and from the workload pod is redirected to the ztunnel's listening sockets created in the pod's network namespace, while the ztunnel itself is running outside of the pod and its network namespace.

As services are added or removed from Istio ambient mode, the istio-cni plug-in will dynamically update the redirection rules. If labels are removed, or Istio is uninstalled, the redirection rules will be flushed and reset to what they were before the installation/enablement of Istio ambient.

The key innovation of the CNI plug-in is to configure the in-pod network redirection, while informing ztunnel about the application pods' network namespaces so the ztunnel can create the listening sockets in the network namespace of each of the co-located pods. This in-pod approach was not obvious to Istio maintainers because the ztunnel itself runs in its own network namespace, and thus we started traffic redirection between the application pod and ztunnel within the node network namespace first in the initial implementation. With this approach, the traffic redirection between ztunnel and application pods happens in the pods' network namespace, which is very similar to sidecars and application pods today and is strictly invisible to any Kubernetes primary CNI operating in the node network namespace. Network policy can continue to be enforced and managed by any Kubernetes primary CNI, regardless of whether the CNI uses eBPF or iptables, without any conflict.

HTTP-Based Overlay Network Environment

Istio 1.15 introduced HTTP-Based Overlay Network Environment (HBONE), a new tunneling mechanism for interservice mesh communication. This is not something an application owner sees or uses directly. HBONE operates transparently behind the scenes between proxies.

This transport protocol runs on a dedicated port (15008) between the proxies in the data plane and uses mTLS and strong identity, which is similar to how Istio currently works. However, HBONE is hidden to workloads—it's not used by applications. The reasons to support a better transport mechanism include:

- Better support for protocols, including server-send-first
- Better support for incrementally adopting Istio, especially when apps use their own TLS certificates
- Support for calling pod IPs directly and eliminating ways around Istio mTLS/encapsulation that we see today

HBONE is based on HTTP/2 CONNECT and tunnels requests between workloads as streams, reusing the HTTP/2 connection where possible. Istio ambient uses this transport between the data plane components (ztunnel, waypoint proxies, etc.) by default, as shown in Figure 2-5.

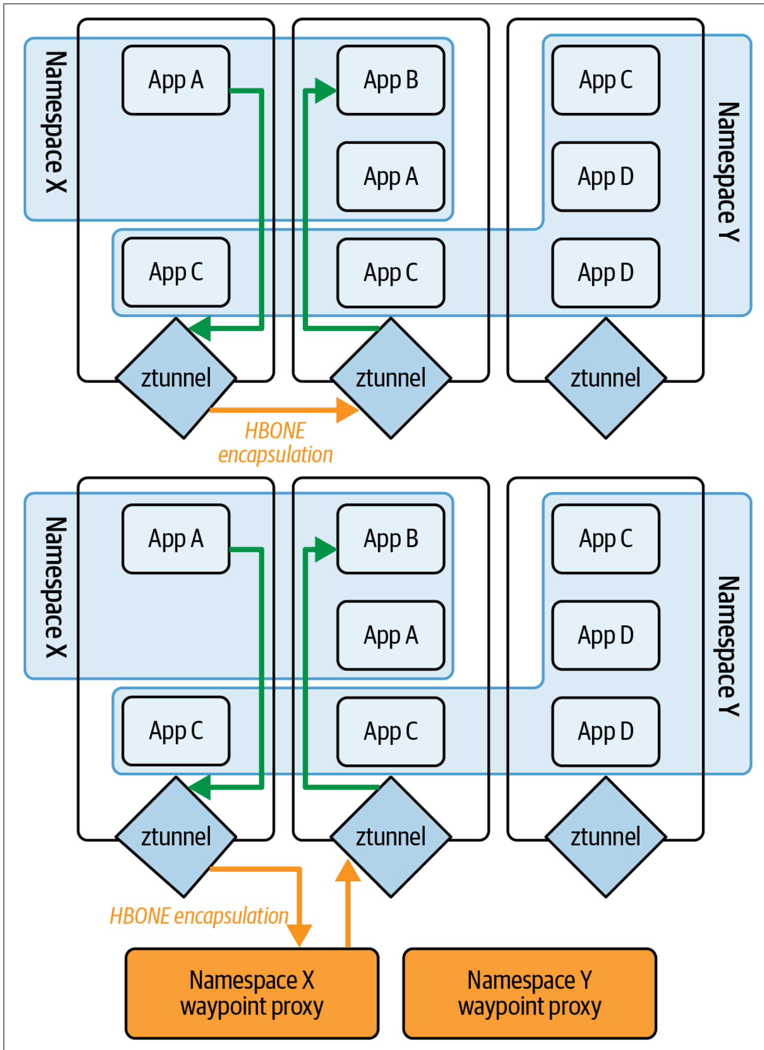

Figure 2-5. HBONE is used as the internal encapsulation mechanism between ztunnel and waypoint proxy data plane proxies

At this point, you should have a good understanding of the components that make up the Istio ambient mode architecture. In the next section, we'll review some of the important consequences of the Istio ambient architecture.

Ambient Is Designed for Scale and Security

We have designed ambient mode to handle very large Kubernetes clusters with tens or hundreds of thousands of pods and to be as secure as (or more secure than) sidecars. Istio ambient mode is built upon five-plus years of experience and expertise in building service-mesh infrastructure, and the design decisions have been taken to try and balance the best of both worlds (sidecar versus sidecar-less). Let's cover some areas where it's important to understand the implications of Istio ambient mode.

Why Did We Rewrite ztunnel from Scratch?

When Istio ambient service mesh was initially announced in 2022, the ztunnel was implemented using an Envoy proxy. Given that we use Envoy for the rest of Istio—sidecars, gateways, and waypoint proxies—it was natural for us to start implementing ztunnel using Envoy.

However, we found that while Envoy was a great fit for its rich L7 feature set and extensibility, it was challenging to debug our Envoy-based ztunnel, which requires deep knowledge of extremely complex Envoy configurations. We purposefully built ztunnel to be simpler and performant using Rust, which processes very minimal human-readable configurations from the Istio control plane, compared with the large, complex, nearly not human-readable Envoy proxy configuration. You'll not only be able to understand the Discovery Service (xDS) configuration from Istiod to ztunnel much more easily, but you'll also notice much-reduced network traffic and cost between the Istiod control plane and ztunnels due to the minimal configurations.

Is the Ambient Architecture Susceptible to Noisy Neighbor Problems?

The ambient architecture redesigns the service mesh data plane, and the separation of L4 and L7 proxies can raise the question about noisy neighbor problems. The noisy neighbor effect can be seen when one workload affects the behavior or performance of the data plane for other workloads using the same data plane.

The waypoint proxies for the ambient architecture specifically avoid this problem. Each tenant can have its own proxy, so configuration, extensions, or behavior between different target workloads from different tenants are not shared.

The ztunnel, however, is a shared component, so can it be affected? The ztunnel is strictly an L4 component and only deals with L4 connections and bytes, much like Linux or Kubernetes already does. It cannot be configured for complex app-specific behavior, nor can it be extended with things like WebAssembly, Lua, or calls out to external services—all of which can cause a noisy neighbor effect. Since the scope of the responsibility for the ztunnel is tightly bound on the data path, whatever effect a noisy neighbor may exhibit will be present to all components in the data path (the Linux kernel, network devices, etc.).

Does Ambient Mode Cause an Increase in Latency?

Since Istio ambient splits the data plane into two layers, there are more options for tuning performance. In use cases where waypoint proxies or policy does not need to be in the data path, we see quite a significant performance improvement. A big part of the reason for this is bypassing any unnecessary L7 (e.g., HTTP 1.1, HTTP/2, gRPC, etc.) parsing of the byte stream, which can consume a lot of cycles.

In the scenarios where an L7 policy needs to be enforced in the waypoint proxy, Istio makes a call out to a remote proxy, and this could potentially increase overall latency for a specific call. From what we've seen in our performance testing for the beta release of Istio ambient, the hop to the waypoint proxy has similar latency for application response time as the sidecar model, which processes L7 on both ends of the connection. We're exploring different options

to improve over sidecars, and that's the goal we'll continue to tune toward.

What About Security?

We believe the security boundaries in ambient mode are improved compared to the sidecar model. In the sidecar model, the data plane is run next to the application and shares the same pod boundary as the app. If the application is compromised, the attacker will have access to all secret material in the pod, including tokens and certificates used by the data plane. In ambient mode, no part of the data plane or identity material is exposed to the applications and the application is unaware of Istio's presence.

The ztunnel is a shared component, however, albeit with a greatly reduced attack surface since it only handles L4 functionality. This component should be treated with the same regard for security as any of the other shared components on the node, such as the kubelet or a CNI agent. Compromise of this component would have a lower blast radius from the other comparable shared components, as only the currently running identities on that specific node would be at risk.

With a large focus on operational improvements for Istio ambient, platform owners have the ability to upgrade the Istio data plane more quickly and safely without forcing application restarts. This gives operators the chance to more predictably and consistently patch common vulnerabilities and exposures (CVEs) that are discovered to keep the system in a secure state.

Wrapping Up

We covered a lot in this chapter, from the main architectural components in Istio ambient to how it works, to some implications of this architecture. We encourage you to check out the documentation or educational workshops (see Chapter 4) to learn more about Istio ambient. In the next chapter, we will walk through Istio ambient mode use cases, adoption strategy, and the benefits it can bring to your organization.

CHAPTER 3
Exploring Istio Ambient Mode

Now that you have an overview of the Istio ambient architecture, let's walk through some Istio ambient mode use cases, adoption strategy, the benefits it can bring to your organization, and how ambient mode works. We'll begin by looking at the new ambient profile, which installs the Istiod control plane and data plane components, such as the istio-cni and ztunnel.

The Ambient Profile

There are two different ways to install Istio, *istioctl* and *helm*. To start, istioctl is the most straightforward approach to installing Istio ambient mode. For production, we recommend using helm to install Istio since helm can help you manage components separately with more flexibility.

Istio has a few built-in configuration profiles, such as demo, default, minimal, etc., that you can use when installing Istio. These profiles provide customization of the Istio control plane and data plane. Istio ambient introduces the new ambient profile. The ambient profile is not yet the default profile, and we recommend that it be used in production with precautions. Because it supports both the sidecar-less and sidecar architectures, ambient will become the default profile for Istio after it is production-ready and used by many users. For now, though, to install ambient, you must specify the ambient profile.

Without any customization, the ambient profile installs the Istio custom resource definitions, Istiod, ztunnel, and CNI plug-in. The Istio CNI plug-in is required for the ambient profile because it is responsible for detecting which application pods are part of ambient and configuring the traffic redirection between the ztunnel to the co-located pods. This is a significant change because Istio CNI was previously an optional component.

HBONE is enabled by default to configure sidecars, ztunnel, and Istio gateways (if enabled) to use HBONE so they can communicate with each other. This configuration is only valid for newer Istio (v1.15 or later). Refer to Chapter 2 for more details on HBONE.

Both istio-cni and ztunnel are deployed as Kubernetes DaemonSets. Running on every node is intentional because each Istio CNI plug-in pod checks all pods co-located on the same node to see if they are part of ambient mode. For any pods in ambient mode on the node where the Istio CNI pod runs, the Istio CNI plug-in pod configures in-pod traffic redirects automatically for you so that all incoming and outgoing traffic to any pod in ambient is redirected to its co-located ztunnel's sockets in the pod's network namespace first. As new pods are deployed or removed on the node, the co-located Istio CNI plug-in pod continues to monitor any pod creation or removal, and it updates the redirect functions to reflect these pod changes accordingly.

Essentially, the Istio CNI plug-in takes care that the incoming and outgoing traffic to pods is redirected to ztunnel's sockets, similar to how it configures the traffic redirection to sidecars, all within the pod network namespace. For any incoming or outgoing traffic captured for the ztunnel, the ztunnel is configured to route intelligently, based on the original destination.

NOTE

For pods without sidecars and not in ambient, Istio CNI won't attempt to set up any traffic redirection. For pods with sidecars, Istio CNI sets up traffic redirection between the application container and its sidecar.

After you install ambient mode, you may begin to add applications. Before we discuss how to do that, let's focus on the benefits, so you can justify paying for the control plane and data plane resources and keep them running.

The Benefits of Running Your Applications in Ambient

In this section, we'll discuss the key benefits of running your applications in ambient, compared to no mesh or sidecar architecture. We hope these benefits will help you decide whether you should invest in running your applications in ambient mode.

Simplified Operation

By simply labeling a namespace, you can enroll your applications in ambient without requiring any restart or change of your application deployment. Your application can remain running without any change, and enrolling it to ambient adds little or no latency to it. When there is a new Istio or Envoy release, you no longer need to restart your application to pick up the release. You can update ztunnel or waypoints independently without any change to your application.

Better Incremental Adoption

In a nutshell, a service mesh provides security, observability, and traffic management. But not everyone needs every feature, and most users adopt service mesh incrementally with sidecars prior to ambient mode. We designed ambient mode knowing that nearly all users would want encrypted traffic with mTLS among their applications, where only trusted identities can call a given application. However, some users may only want to trust certain trusted identities with specific rules and conditions, such as a particular method or header. Some users may want observability at L4 or L7 or both, and some users may want traffic control or resiliency. Or users may want all these functions. In any case, they should have the flexibility to choose what they need. These benefits are the same, regardless of sidecars and sidecar-less architecture.

Ambient mode allows better incremental adoption because the secure overlay layer is separated via ztunnel, and the L7 processing layer via waypoint proxy. This means waypoint proxies are only required when you need any L7 telemetry or rich L7 authorization policy or traffic management functions. In other words, you don't need to operate waypoint proxies or allocate any resources for them when you don't need any L7 processing. Essentially, we expect many

users will simply run ambient mode using *only* the secure overlay layer, which isn't possible with sidecar architecture.

The two-layer architecture also enables a more granular transition from no mesh or sidecar to the secure overlay layer (on a pod level, namespace level, or mesh level), to the L7 processing layer (on a service account level or namespace level).

Simpler Application Onboarding

Besides the service mesh features common to sidecar and ambient, what is unique about ambient is transparency and that it is non-intrusive to the application. If you have had issues with running your applications with a sidecar, or have had startup or shutdown sequence issues between your application container and sidecars, ambient mode could be a much better solution for you, as you no longer need to worry about managing the sidecar startup or shutdown sequence compared with your application container.

The benefit is beyond not needing to inject the sidecar or restart the application pod. In Chapter 2, you learned about HBONE and how it provides better support for applications that speak server-send-first protocols (e.g., MySQL) or call pod IP directly (e.g., Kubernetes StatefulSets). Ambient mode's broader support for applications will reduce your time and surprises when onboarding your application onto the mesh.

Reduced Infrastructure Cost

When running your applications in service mesh, you pay for the control plane and data plane CPU and memory allocation, regardless of the actual usage. You may also pay for the network cost associated with transferring configurations from the Istio control plane to the data plane. Compared to sidecars, ambient mode reduces infrastructure costs by having the required ztunnel handle the secure overlay layer for all co-located pods and having the optional waypoint proxy deployed outside of the application pod.

Let's walk through a simple scenario with Apps A, B, C, and D, running on a four-node Kubernetes cluster (see Table 3-1). If each application has one replica, you won't see much savings. However, no one runs critical applications with one replica in production environments. If you increase to 10 or 20 replicas, you'll start to notice the difference between the number of required data plane

containers you need to pay for as part of the infrastructure cost. You don't have a choice to have fewer sidecars as the replica number grows, but with ambient you can control which application requires a waypoint proxy and scale the waypoint proxy independently from the application. Further, a large number of sidecars also means more network costs to transfer configurations from the Istio control plane to each of the sidecars versus a few ztunnels and optional waypoint proxies.

Table 3-1. A comparison of a service mesh with sidecars, ambient with ztunnel, and ambient with optional waypoint proxies

	Sidecars	Ambient with ztunnel	Ambient with optional waypoint proxies
1 replica for each app	4	4	2
10 replicas for each app	40	4	4
20 replicas for each app	80	4	4+ (depending on features used and load of the proxy)

Now that you understand the two layers introduced by ambient mode and the benefits brought by each layer, you'll learn how to add your applications to ambient mode next.

Incrementally Adopting Ambient

Adding applications to ambient mode allows you to leverage the mesh to secure, connect, and observe services incrementally, without any sidecars. You can either enroll applications in ambient after your services are deployed or as your services are being deployed. Ambient mode is transparent to your applications, without any changes in either scenario.

We recommend you adopt Istio ambient incrementally to leverage the two-layer data plane architecture fully. Incremental adoption provides the following benefits:

- To minimize impacts to your application, you can adopt ambient mode per application, per namespace, or mesh-wide as needed.
- Enjoy the secure overlay layer benefits as you adopt service mesh, without paying for the L7 processing layer.

- Learn only the required Istio resources for what you need (could be none), which helps you minimize learning, as Istio provides very rich APIs.

A common adoption strategy is to secure inbound traffic with the Istio ingress gateway first, as it is the least intrusive, then gradually add one or more of your services to the ambient service mesh to leverage L4 mesh capabilities provided by the secure overlay layer. If you still require L7 mesh capabilities provided by the L7 processing layer, you can add waypoint proxies as needed.

Securing Inbound Traffic

Most users adopt service mesh by securing inbound traffic from a source outside the mesh. Prior to ambient, this was the most popular first step, not only because it is a common requirement but also because it doesn't require services running in the mesh. Istio ingress gateway can be programmed via Kubernetes `Gateway` and `HTTPRoute`/`TCPRoute` (or Istio `Gateway` and `VirtualService`) resources to terminate TLS or mTLS traffic and route to services running outside the mesh. For example, you can deploy the Kubernetes `Gateway` and `HTTPRoute` resources for the `web-api` service to allow the service to be securely accessed from outside the cluster via the Istio ingress gateway. While it was perfectly fine to secure inbound traffic without sidecars for the `web-api` service prior to the ambient launch, the connection between the Istio ingress gateway and the `web-api` service is not mTLS without sidecars for the `web-api` service.

Securing inbound traffic is the same with sidecar or ambient, using either Kubernetes `Gateway` and `HTTPRoute`/`TCPRoute` (or Istio `Gateway` and `VirtualService`) resources. This ensures a smooth transition from sidecar to ambient without changing any of your Istio resources. What is unique with ambient is that you can secure the connection between the Istio ingress gateway and the `web-api` service without a sidecar by simply enrolling the `web-api` service in ambient.

Including Workloads in Ambient

Workloads can be included in the service mesh (i.e., without a sidecar) with two options:

- By specific namespace (`DEFAULT`)
- Individual pods

We recommend you start with the `DEFAULT` mode, adding workloads to ambient on a namespace basis. You simply add the `istio.io/dataplane-mode=ambient` label to your namespace, and all pods in your namespace will be part of ambient (e.g., `kubectl label namespace default istio.io/dataplane-mode=ambient`). As the example in Figure 3-1 shows, it's that simple. The best part is that there is no need to restart or redeploy anything. Because ambient mode is transparent to the workloads, your workloads should continue running without any interruption as they are included in ambient mode. This is why we called it *ambient*.

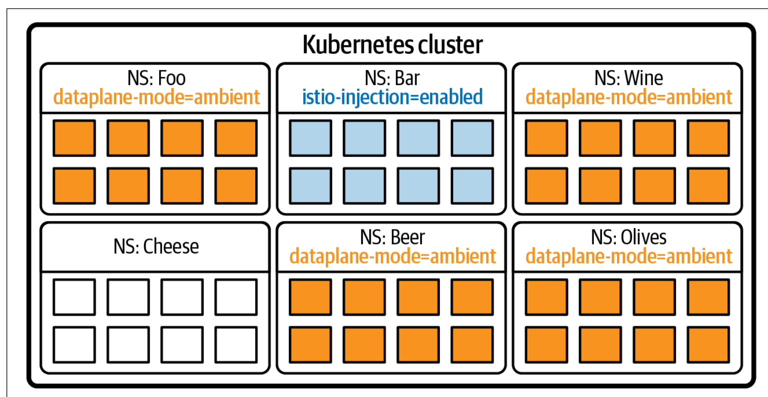

Figure 3-1. Workloads can be added to the mesh by labeling the namespace with `istio.io/dataplane-mode=ambient`

If you recall, the sidecar architecture requires each sidecar to be deployed to each application pod and connected to Istiod to get the latest xDS configuration, which could place a limit on how many connections a single Istiod instance could support. With the elimination of sidecars, the ambient architecture is designed to drastically reduce the connections to Istiod, which allows the Istio control plane to manage more application pods and send less configuration

to the data plane. This means you pay less for the control plane and network costs between the control plane and the data plane.

With sidecars, before you can add any workloads to the mesh, you need to be aware of the application requirements to ensure the services meet the minimum requirements. With ambient, you no longer need to worry about many of these requirements, such as avoiding application UID 1337 or many ports used by your sidecar, because there is simply no sidecar running next to your application container.

In addition to adding workloads to ambient after they are deployed, you can enroll workloads in ambient as they are deployed by labeling the namespace to be part of ambient first. Want to exclude a certain pod from ambient in the namespace? You simply label the `istio.io/dataplane-mode` as `none` on the pod descriptor. Or you can label individual pods with `istio.io/dataplane-mode` as `ambient` to enroll pods to ambient without the namespace label. The pod label allows you to configure individual pods into or opt them out of ambient on a per pod basis, in case you want to add pods to ambient one pod at a time to minimize impact to your application or exclude certain pods from ambient due to incompatibility.

> **NOTE**
>
> What if you also have the injection label in your namespace or pod (e.g., `istio-injection` or `istio.io/rev`)? The injection label has preference over the `istio.io/dataplane-mode` label. We designed it this way to be backward compatible as you transit from sidecar to sidecar-less with ambient mode.

What Have You Gained?

By including your workloads in ambient, you have gained many benefits provided by the secure overlay layer via ztunnel:

Security
: mTLS-encrypted communication among your applications with cryptographic-based identity; simple L4 authorization policy

Observability
: TCP metrics and logs

Traffic control
: TCP routing

We'll cover mTLS, L4 authorization policy, and telemetry in detail next.

mTLS

Similar to sidecars, each service account will be assigned to its own identity, and key/certificate pairs are signed for each service account via certificate signing requests (CSRs) from ztunnel. Adding your workloads to ambient enables your workloads to communicate with encrypted traffic using cryptographic identity automatically, without any code change or sidecar injection.

Ztunnel can request key/certificate pairs to be signed via CSR *only* for each service account used by pods running on the co-located node. For example, there are `sleep` and `nonsleep` service accounts on the ambient-worker node. The ztunnel pod running on the co-located node of the sleep pod can only impersonate the `sleep` service account, not the `web-api` service account. You can view the X.509 certificates managed by your ztunnel by using the `istioctl ztunnel-config certificate` command, then base 64 decode each of the certificates. After decoding, you can also step through the X.509 certificate to view the `Issuer, Validity`, `Subject Alternative Name` (for example, `spiffe://cluster.local/ns/default/sa/sleep`), etc., for each service's X.509 certificate. Similar to the sidecar architecture, these X.509 certificates will be automatically rotated well before the expiration (every 12 hours by default in Istio) without you needing to do anything:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 261030581690837559376424197472710488863...
        Signature Algorithm: SHA256-RSA
        Issuer: O=cluster.local
        Validity
            Not Before: Aug 14 21:18:49 2024 UTC
            Not After : Aug 15 21:20:49 2024 UTC
        Subject:
        Subject Public Key Info:
            Public Key Algorithm: ECDSA
            Public-Key: (256 bit)...
            Curve: P-256
            ...
            X509v3 Subject Alternative Name: critical
            URI:spiffe://cluster.local/ns/default/sa/sleep
        ...
```

Enforce L4 authorization policies

While it is great to have mTLS among all your services, to fulfill a zero-trust architecture at the network layer, you'll want to deny all access except for access that is explicitly granted. This is particularly important for the distributed nature of microservices in the cloud, commonly deployed in Kubernetes, where pods can be dynamically scaled up or down or even restarted. For example, you start by deploying an allow-nothing Istio authorization policy for everything in your namespace. This is effective right away to disallow any service in your namespace to be accessed. Then you can gradually allow your source workloads' service accounts to access `web-api` by creating and deploying an authorization policy that selects the `app: web-api` by label with the `ALLOW` action from the source principals (for example, `cluster.local/ns/default/sa/sleep` and `cluster.local/ns/istio-system/sa/istio-gateway-service-account`).

> **NOTE**
> Only a simple authorization policy is supported with the secure overlay layer via ztunnel, such as denying all or allowing a specific service account to access a given service. We'll cover some rich authorization policies that contain headers, paths or methods, etc., soon.

The simple L4-based Istio authorization policy is the exact same policy you use today for sidecars. We purposefully designed most of the Istio APIs to remain the same to ensure a smooth transition from sidecar to ambient.

L4 telemetry

You'll also get L4 metrics or logs automatically without any extra steps. For example, from the destination ztunnel's */stats/prometheus* endpoint, you'll see several L4 metrics that start with "istio_tcp" such as `istio_tcp_connections_opened_total`. These metrics contain useful information about the TCP connections, such as the source and destination workloads with their identities and how many TCP connections are closed between the source and destination workloads. You can use these metrics to check for unexpected unauthorized connections (and which source identities they come from).

> **NOTE**
>
> In addition to L4 telemetry, a provider may provide L7 telemetry in the ztunnel so that users don't need to deploy a waypoint proxy for the sole purpose of L7 telemetry. For example, Solo.io's ztunnel provides high-performant L7 telemetry, including HTTP metrics, logs, and traces.
>
> If you recall that, in Chapter 2, we discussed how we purposefully separated waypoint proxy from ztunnel to avoid noisy neighbor problems with L7 processing, you may be wondering, why L7 telemetry in ztunnel? L7 telemetry in ztunnel does not modify HTTP traffic in any way, so thus we can continue to achieve the performance and security requirements of ztunnel.

L7 Processing

You may find what is provided in the secure overlay layer doesn't meet your requirements. For example, you want a rich L7 authorization policy that configures access based on a particular method and path. Or you want to launch a newer version of your service without impacting existing traffic or to perform a canary test on the newer version. Or you want to get HTTP access logs, metrics, and distributed tracing among some of your services. In this section, we'll discuss how you can opt in to enforce L7 processing with ambient mode to perform these common scenarios.

To summarize, the L7 processing layer via waypoint proxy provides the following three key benefits:

Security
: Rich L7 authorization policy

Traffic management
: Dark launch, canary test, resiliency, chaos testing, and controlling outbound traffic

Observability
: HTTP metrics, access logs, and tracing

L7 authorization policies

L4 authorization policy is useful but may not be sufficient for your needs if you want to only allow access for a given path and a given method, or only when the request is from certain sources' JSON Web Token (JWT) claims or certain headers. The recommended

zero-trust architecture is to explicitly allow only exactly what you need and no more. For example, you want to allow the `sleep` or `istio-gateway` service to access the `web-api` service only on the `GET` method and not on the `DELETE` or `POST` method.

The target ztunnel denies L4 authorization policies. With L7 authorization policies, the `web-api` service's waypoint proxy, which serves as the L7 authorization policy enforcement point, denies the policy. Like sidecars, the waypoint proxy gets its xDS configuration from Istiod. This ensures its role-based access control (RBAC) filter is configured properly based on the access policies from the authorization policies you deployed.

Earlier, we explained how to view the X.509 certificates managed by your ztunnel. Similarly, using the `istioctl pc secret` command, you can confirm that the universal resource identifier for the `web-api`'s waypoint proxy `Subject Alternate Name` is also `spiffe://cluster.local/ns/default/sa/waypoint`. The waypoint has its own identity, different from `web-api`'s identity. This is purposefully designed so that the waypoint can be used by multiple services in the namespace (or even outside of the namespace). Similar to `web-api`'s certificates, Istio automatically rotates the waypoint's certificates without you performing any manual actions.

L7 traffic management

Getting service timeouts and circuit-breaker configurations properly set in a distributed microservice application is difficult. Similarly, it is difficult to perform canary testing without redeploying anything. Istio makes it easier to get these settings correct by enabling you to increase resiliency or shift traffic without modifying your deployment. You can continue to use Kubernetes Gateway resources (`Gateway`, `HTTPRoute`/`TCPRoute`, etc.) or Istio's classic network resources with ambient mode.

Prior to ambient, to achieve L7 resiliency or traffic management features, you would have sidecars on both source and destination, with the proxy configuration to handle these functions mostly in the source's sidecar proxy. For example, the `sleep` sidecar retries the connection to the `web-api` service three times or injects a fault with five seconds' delay. With ambient, you only need a destination waypoint proxy to handle these functions, without requiring a waypoint proxy for the source. Reducing the need for a source waypoint

proxy simplifies the operation and eliminates the cost of running the source waypoint proxy.

> **NOTE**
>
> When do you need to deploy the source waypoint proxy? You don't. This design has reduced the configuration for the destination waypoint proxy, as it only needs to be aware of the relevant destinations. If you are familiar with sidecars, you may recall you have to use the Istio `Sidecar` resource and `exportTo` configuration to trim the configuration of your sidecar to the minimum. With ambient mode, `Sidecar` resource and `exportTo` configuration are no longer applicable, as the waypoint proxy focuses on configurations related to its destinations only.
>
> You can deploy an egress waypoint proxy for controlling egress traffic. You can apply Istio's `ServiceEntry` resources to control access to specific external destinations, and use Istio's `AuthorizationPolicy` resources to control what workloads can access the egress waypoint to call these external destinations.

L7 telemetry

With the waypoint proxy deployed for the `web-api` service, you automatically get L7 metrics for the `web-api` service, the same as with sidecars. For example, you can view the 403 response code from the `web-api` service's waypoint proxy pod's */stats/prometheus* endpoint.

You'll also be able to view access logs in the waypoint proxy's log. For example, you can see *RBAC: access denied* errors in the access log if you attempt to send the `DELETE` request to the `web-api` service.

What is unique about L7 telemetry is that you don't have the sidecar or waypoint proxy on the source side. The destination waypoint proxy is intelligent enough to collect telemetry data for both the source and destination without needing to deploy a proxy on the source side. This reduces the need to operate the source waypoint proxy and the requirement for an extra hop to traverse through the source waypoint proxy when the `sleep` service calls the `web-api` service.

Waypoint proxy for your service

Depending on which service requires L7 processing, you'll need to ensure that the waypoint proxy is deployed and running for the namespace and that the namespace is using the waypoint to ensure the L7 processing is effective. Otherwise, you could have L7 policies applied but not enforced by anything.

You can use the Kubernetes Gateway resource to deploy your waypoint proxy for your namespace and label the namespace to use the waypoint proxy, or you can simply use the `istioctl waypoint` command. Within the Gateway resource, configuring `gatewayClassName` to `istio-waypoint` indicates you want to use the default waypoint proxy provided by Istio instead of Istio ingress gateway or any custom waypoint proxy.

> **NOTE**
>
> Waypoint proxies are for services in the namespace by default. Additionally, you can configure waypoint proxies to process all traffic (including services and workloads), just the workload, or no traffic.
>
> Waypoint proxies are designed to be pluggable through the `gatewayClassName` configuration, following the design principle of the Kubernetes Gateway API. A waypoint provider can provide its own waypoint proxy for Istio ambient mode, which could be more performant or have richer features. For example, Solo.io provides a Gloo waypoint proxy that is built on top of Gloo gateway.

Understanding How Istio Ambient Architecture Works

So far, we've covered the ambient profile, the ambient mode configuration, incrementally adopting ambient starting with securing the inbound traffic, and then including workloads in ambient mode and leveraging the L7 processing layer as needed based on your requirements. It is important to understand how it all works. In this section, we'll discuss workloads in the service mesh following the ambient approach. Sidecar-based deployments are interoperable with the ambient approach, but this section will focus on workloads in ambient mode.

Source Workload Initiates a Call to Another Service

Once a workload is in the service mesh, it may make calls to other services and take advantage of the features of the mesh. As mentioned earlier in this chapter, istio-cni is a plug-in to the cluster CNI that watches for new workloads in the mesh and applies in-pod redirection rules so that traffic to workloads in the mesh will be directed to the ztunnel sockets in the pod network namespace (Figure 3-2).

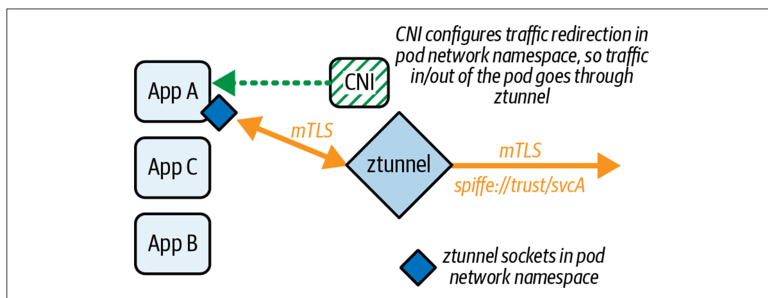

Figure 3-2. Transparent redirection of workload traffic to the in-pod ztunnel sockets

This redirection is completely transparent to the workload—i.e., the workload opens connections as it would normally, but Linux networking is used to direct the connections to the ztunnel sockets. Note that kube-proxy functionality (like service IP translation to pod IP) is skipped; the ztunnel plays the role of the kube-proxy in this case.

ztunnel Handles the Traffic and Initiates mTLS

When the workload traffic gets redirected to the ztunnel sockets, the ztunnel inspects the connection and determines where it should be routed next. The ztunnel will inspect the L3 packet to determine the source (what workload originated the packet) and destination (the target). The ztunnel will map the source address to a cryptographic identity using SPIFFE-based certificates, which will be used as the source side of an mTLS connection.

The ztunnel keeps track of all the identities present in the specific node/host and can map the source workload to the cryptographic identity on the fly. The ztunnel gets the identity certificates from Istiod by sending a CSR on behalf of the workload. The ztunnel can request a source workload's key/certificate pair to be signed via CSR

for *only* service accounts used by pods running on the co-located node. Once the source and destination are identified, the ztunnel will initiate an mTLS connection over the HBONE tunnel.

If the destination has a waypoint proxy, the connection from the ztunnel will be made to the waypoint proxy, which will terminate the destination side of the mTLS connection. If there is no waypoint proxy for the destination, the ztunnel will load-balance the connection to the ztunnel on the node where the destination pod exists (Figure 3-3).

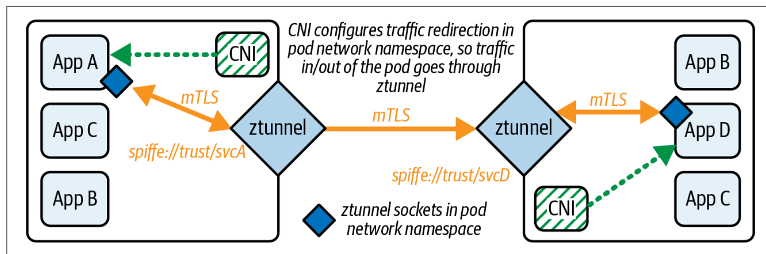

Figure 3-3. Traffic will go directly to the destination ztunnel over mTLS, with the ztunnel resolving the correct identity certificates to use, when there is no destination waypoint proxy

The ztunnel on the destination node will terminate mTLS, enforce any L4 authorization policies, and send the connection to the destination pod.

Destination Proxy Terminates mTLS and Handles Traffic

At this point, the "destination proxy" in our scenario can be the ztunnel on one of the nodes where the destination workload is deployed or the waypoint proxy that has its own identity (see Figure 3-4).

If the destination proxy is the waypoint proxy, the waypoint proxy will terminate the mTLS connection using its own certificate. Once the mTLS connection is terminated, the stream is parsed (to understand HTTP, gRPC, etc.) and various L7 policies can be applied. Once the L7 policies are enforced, an mTLS connection, using the waypoint's identity certificates, is made from the waypoint proxy to the target destination ztunnel.

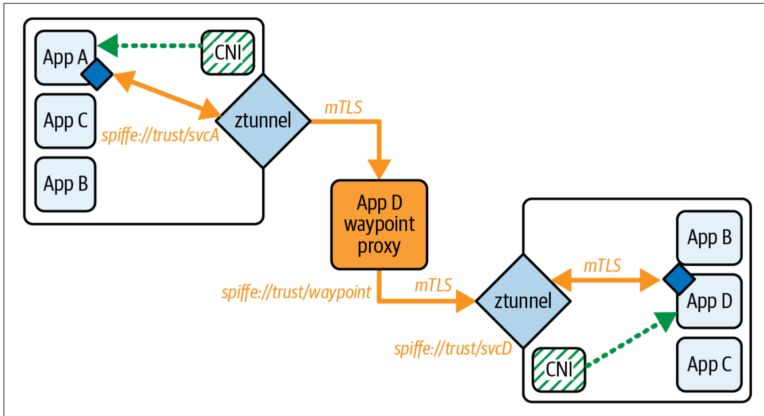

Figure 3-4. Traffic will go through a waypoint proxy for L7 policy enforcement

Traffic Gets Sent to Destination Workload

Once traffic makes it to the ztunnel on the node where the destination workload pod is deployed, the ztunnel will terminate the mTLS connection (whether that's from a downstream ztunnel or waypoint proxy), enforce any L4 authorization policies if they exist, and then deliver the connection to the pod that represents the destination workload. The workload that receives this connection will process the connection without knowledge of ambient mode and optionally return a response.

Wrapping Up

We covered a lot in this chapter, from components installed as part of the Istio ambient profile to benefits provided by ambient mode and how to incrementally adopt ambient mode from the secure overlay layer to the L7 processing layer. We also explained how Istio ambient mode architecture works. We are very excited about how easy it is to include your applications in ambient while keeping them exactly the same and the tremendous benefits ambient mode can bring you with the two-layer approach. If you use Istio sidecars today, sidecars and ambient are interoperable. Combined with the same Istio APIs (we covered `Gateway`, `VirtualService`, and `DestinationRule` resources), this enables you to easily choose or switch to the data plane architecture that is appropriate for your requirements.

CHAPTER 4

Key Takeaways and Next Steps

Istio ambient mode significantly simplifies the experience of adopting a service mesh. Istio ambient mode is transparent to the applications, bringing much broader application support, simplified operation, and reduced infrastructure cost.

If you have considered service mesh or Istio in the past but have walked away due to the complexity or operation cost of sidecars, you should evaluate ambient mode. Istio ambient mode is designed to help you deploy and manage applications transparently and at a lower total cost of ownership.

Takeaways

A service mesh is a foundational infrastructure layer necessary to support running microservices on containers. After reading this book, you should have a better understanding of the following points:

- Service mesh allows you to connect, secure, and observe your services.
- Istio ambient mode implements a "sidecar-less" architecture that is transparent to the workloads in the mesh.
- Ambient mode reduces operational and runtime overhead.
- Istio ambient mode introduces an innovative two-layer approach that separates the secure overlay layer from the L7 processing layer. This allows you to better incrementally adopt

service mesh based on your need and pay for only what you need.

- ztunnel and waypoint proxies are key components introduced in Istio ambient mode. The two-layer approach reduces the CVEs for ztunnel.
- Istio ambient mode is designed to support more workloads than what's possible with a sidecar. mTLS is enforced by default for all workloads in ambient mode.
- Istio ambient mode is designed to simplify day two operations and save infrastructure cost.
- Istio ambient mode supports Kubernetes Gateway API and is backward compatible with most of the classic Istio APIs.
- Interoperability of sidecar-based deployments with workloads in Istio ambient mode helps ensure a more seamless migration.

Next Steps

We hope we have sparked your interest in service mesh and Istio ambient mode. If you would like more information, we recommend you check out the following resources:

- Visit the Istio "Concepts" documentation to learn more about the features that Istio provides and better understand the details of Istio architecture.
- The "Introducing Ambient Mesh" blog post provides an introduction to ambient mode and why the new data plane without sidecar was born. It also contains a short video in which Christian runs through the Istio ambient mode components and demos some capabilities.
- The "Get Started" guide provides step-by-step instructions to help you get started with ambient mode and experience the secure overlay layer with mTLS and L4 authorization policy, and the L7 processing layer with L7 authorization policy, telemetry, and traffic management.
- The Istio "Ambient Mesh Security Deep Dive" blog post digs into the security implications of Istio ambient mode, comparing it with sidecars.

- In the "Istio Ambient Mesh: What Does It Mean to You?" livestream recording, a few ambient mode contributors celebrate the launch of ambient mode and share what ambient mode means to Istio users from their perspectives.
- "Egress Gateways Made Easy" digs into a much simpler configuration for L7 egress control.
- As shown in "Ambient Mesh: Can Sidecar-less Istio Make Applications Faster?", Istio ambient mode makes applications faster than the baseline: a service mesh usually is seen as adding overhead (however minimal it may be), but in the case of Istio ambient mode, the mesh path may be faster than the baseline nonmesh path.
- The "Using Istio in Ambient Mode—Do More for Less!" blog post explores how Istio ambient mode was designed to reduce the service mesh infrastructure resources typically associated with sidecars.

When you are ready, the next logical step is to apply what you have learned on your own projects to truly see the value you can achieve with Istio service mesh in ambient mode.

About the Authors

Lin Sun is the head of open source at Solo.io and a CNCF TOC member and ambassador. She has worked on the Istio service mesh since the beginning of the project in 2017 and serves on the Istio Steering Committee and the Technical Oversight Committee. Previously, she was a senior technical staff member and master inventor at IBM for 15+ years. She coauthored *Istio Ambient Explained* (O'Reilly) and has more than 200 patents to her name.

Christian Posta (@christianposta) is VP, global field CTO at Solo.io. He is a coauthor of *Istio in Action* (Manning) as well as many other books on cloud native architecture. He 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 and commercial enterprises as well as web-scale companies and now helps organizations create and deploy large-scale, cloud native, resilient, distributed architectures. He enjoys mentoring, training, and leading teams to be successful with distributed systems concepts, microservices, DevOps, and cloud native application design.