TRANSFORM NETWORK AND TRAFFIC WITH ISTIO

Table of Contents

01 Network, traffic, and microservices trends

B2B organizations nowadays are adopting the subscription-based, SaaS business model. It ensures a more predictable revenue stream and helps build customer loyalty, compared to the traditional one-time purchase model.

As more and more enterprises adopt subscription-based business models, they see a rise in traffic to their applications with an increasing number of subscribers. To handle the traffic and meet the demand, they build applications in a microservices pattern rather than a monolithic one.

In a microservices architecture, the business logic of the applications is developed, deployed, and scaled independently. This gives true agility for organizations compared to building a monolith. Microservices suit most enterprises serving high traffic as they are more resilient and flexible. They reduce application downtime and provide a better customer experience.

However, microservices architecture brings its own set of challenges. One of the primary challenges is the network complexity & security. Typically, enterprises deploy services in container orchestration tools like Kubernetes, across multiple cloud data centers. And these services talk to each other over a network, making it difficult to secure, manage, and monitor the network.

The two challenges — increased traffic and network complexity — put IT teams such as Architects, DevOps engineers, and site reliability engineers (SREs) under much pressure. This ebook will explore those challenges and show how Istio service mesh can simplify the network complexity by making it resilient, secured and manageable.

03

Networking loss if not worked on time

SaaS applications are hosted in the cloud, which means that users can access them from any device with an internet connection. An application with a vast user base would have subscribers from around the globe. So enterprises have to ensure that the applications do not have downtime, 24*7. That is one way to improve customer experience, otherwise, it will be hard to retain existing customers.

With end-customers accessing the services anytime and anywhere, downtime will impact the revenue loss directly. Any SLA below 99.9% will severely affect customer experience. Below figures show the statistics of revenue loss due to network downtime, MTTR, and the average number of outages per year:

$300,000 per hour

91% of large American companies lose around $0.3M per one hour of network downtime.

84 mins

It takes ~84 mins for the Ops team to restore the service after an interruption.

34 outages per year

On an avg, every American company faces 19 IT brownouts and 15 service outages per year such as site freeze, crash or slowdown.

Source:
https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/
https://www.logicmonitor.com/it-downtime-mitigation-report

Challenges of DevOps and Architects

While starting off, most architects and DevOps engineers only have a handful of services to deal with. As the applications scale and the number of services increases, managing the network, traffic, and security becomes complex.

Tedious security compliance

Applications deployed in multiple clusters from different cloud vendors talk to each other over the network. It is essential for such traffic to comply with certain standards to keep out intruders and to ensure secure communication.

The problem is that security policies are typically cluster-local and do not work across cluster boundaries. This points to a need for a solution that can enforce consistent security policies across clusters.

Chaotic network management

DevOps engineers would often need to control the traffic flow to services — to perform canary deployments, for example. And they also would want to test the resiliency and reliability of the system by injecting faults and implementing circuit breakers.

Achieving such kinds of granular controls over the network requires DevOps engineers to create a lot of configurations and scripting in Kubernetes and the cloud environment.

Lack of visualization over the network

With applications distributed over a network and communications happening between them, it becomes hard for SREs to keep track of the health and performance of the network infrastructure.

Without a single plane for traffic visibility (north-south and east-west) and granular performance and behavioral details, it is challenging for SREs to troubleshoot and resolve issues quickly. This often leads to SLA breaches and service unavailability.

Istio solves all the above problems by abstracting the network and security logic from the application layer to its own infrastructure layer. Let us see what Istio service mesh is, its components, and how it can help in simplifying network challenges for architects, DevOps/cloud engineers, and SREs.

Introduction to Istio service mesh

Istio is an open-source, CNCF-graduated service mesh platform that simplifies and secures traffic between microservices. Istio provides a dedicated infrastructure for traffic management, security, and observability, to help developers handle the network of microservices in Kubernetes and multiple clouds, at scale.

Istio works by deploying Envoy proxy — an L4 and L7 layer proxy — alongside each microservice. The proxy intercepts and handles service-to-service traffic, and thus abstracts communication logic from the service/application layer into a dedicated infrastructure layer (refer to fig. A).

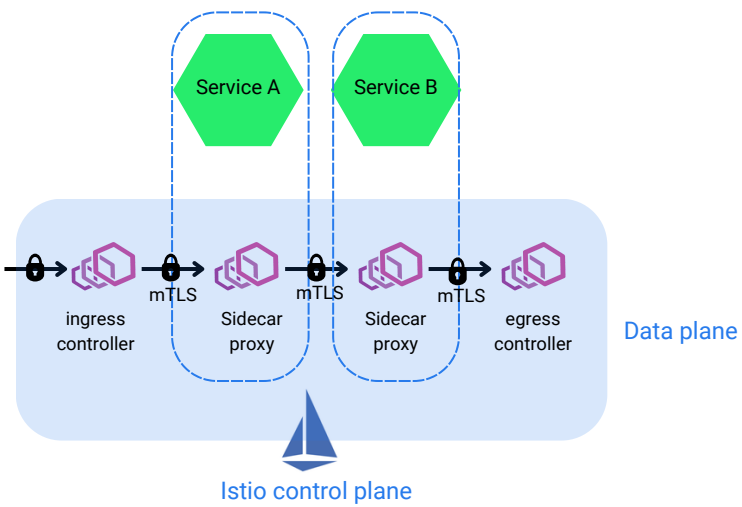


Fig. A – Service-to-service communication before and after deploying Istio service mesh in a Kubernetes cluster

Features of Istio service mesh

Istio provides several features for a variety of IT teams in an organization. Traffic management, security, and observability are the major ones.

- Traffic Management
- Network Security
- Observability

Traffic Management

Istio automatically detects all the endpoints of respective services in the mesh and stores them in its internal service registry. It helps Istio to manage traffic by load balancing between replicas. Istio supports applying fine-grained control over traffic splitting between services, like routing a percentage of traffic to a specific service.

Besides, Istio provides the following network resilience and testing methods to ensure the reliability of the applications:

- **Timeouts:** It is the timeframe within which a service call succeeds or fails. Timeouts are useful so that services do not hang due to the Envoy proxy waiting for replies forever.

- **Retries:** It is the number of times Envoy should try to connect to a service when the initial connection attempt fails.

- **Circuit breakers:** It is the threshold for calls to specific hosts within a service. Once the threshold has been reached, further connections to the host are prevented.

- **Fault injection:** It is a testing method for systems where errors are deliberately introduced to verify their ability to recover from error conditions.

(Watch video on advanced traffic management with Istio: Advance traffic management using Istio | Kubernetes | Demo)

Network Security

Istio helps to secure the network of microservices by facilitating granular authentication, authorization, and access control policies.

- **Authentication:** Istio verifies the identity of users (humans or machines) by allowing peer-to-peer and request authentication policies. Peer authentication involves mTLS implementation, where communication between services is encrypted and authenticated using certificates issued for both the client and the server.Request authentication involves server-side verification, where the client has to attach JWT (JSON Web Token) to the request.

- **Authorization:** Verifying whether the authenticated user is allowed to access a server and perform specific action is done using authorization policies in Istio. Authorization policies can be set to allow, deny, or perform custom actions against an incoming request based on different parameters.

- **Access control:** Istio controls the access of authorized users to resources by implementing the least privilege policy and role-based access control (RBAC). Istio supports RBAC policies to be set on method, service, and namespace levels.

(Watch video on implementing mTLS with Istio: mTLS with Istio service mesh | Demo | IMESH)

Observability

Istio offers observability and real-time visibility into the performance and behavior of applications. It is done by providing detailed telemetry for traffic flow between services in the mesh. Istio generates the following types of telemetry:

- **Metrics:** Istio generates service metrics for real-time performance monitoring of services. They are based on the four “golden signals” of monitoring, which are latency, traffic, errors, and saturation.

- **Distributed traces:** Istio collects traces of activity across multiple services in a mesh to better understand service dependency and traffic flow.

- **Access logs:** Istio can produce a complete record of communication between services in the mesh, making it easier to understand the behavior of each workload.

(Watch video on configuring Istio with Prometheus: Configuring Istio with Prometheus | Grafana | Metrics Monitoring)

5 Ways to transform network and traffic with Istio

Istio helps enterprises deal with ever-increasing traffic and network complexity. Below are 5 ways that IMESH proposes with Istio, which DevOps and architects can heed to keep a resilient, scalable, and available infrastructure:

1. Istio for managing internal network
2. Istio to handle traffic at the edge
3. Chaos engineering with Istio
4. Istio for implementing mTLS
5. Istio for network observability

#1 - Istio for managing the internal network

Istio provides the following features to test the network resiliency and to ensure applications operate reliably:

- Circuit breaking
- Timeouts
- Retries
- Traffic mirroring

They can be configured using Istio configuration resources (KubernetesCRDs)called `VirtualService` and `DestinationRule`.

Circuit breaking

In a web application, circuit breakers set limits to concurrent connections to a service and prevent it from overloading with requests. It is highly useful for B2B and B2C SaaS applications where service responsiveness can make or break the customer experience.

In circuit breaking, if the maximum connection requests to an upstream service go over the specified limit, they will become pending in a queue. And if the number of pending requests breaches the limit, further requests are denied until the pending ones are processed.

The circuit breaker is tripped so that client requests fail quickly, without exhausting the services and cascading the failure to the overall system. Istio uses `DestinationRule` to configure circuit breakers.

Here is a sample `DestinationRule` with circuit breaker rules:

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: istio-support
spec:
  host: istio-support
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
    outlierDetection:
      consecutive5xxErrors: 1
      interval: 1s
      baseEjectionTime: 3m
      maxEjectionPercent: 100
```

- The above `DestinationRule` sets the number of maximum connections and pending requests to `istio-support` service to

- If `istio-support` receives 3 requests simultaneously for example, one request will establish cnnection, one will be in the queue, and the third one or any additional requests will be denied until the pending one is processed (see Fig.B).

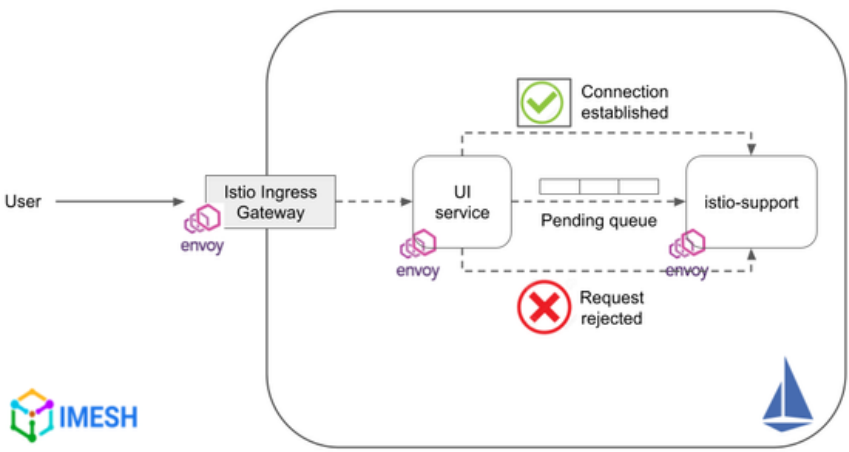


Fig. B - Circuit breaking with Istio

- The `outlierDetection` section towards the end defines the rules to evict unhealthy pods out of the load balancing pool. It means that if any pod of `istio-support` triggers a 5xx (or server) error, the pod will be considered an outlier or unhealthy. It will then be ejected out for 3 minutes before being allowed to rejoin the load balancing pool.

Timeouts

Timeout refers to the amount of time the Envoy proxy of the source should wait for a response from the destination service (see Fig.C). Timeouts fail or succeed a call within a specific timeframe, which ensures that services do not wait indefinitely for a response.

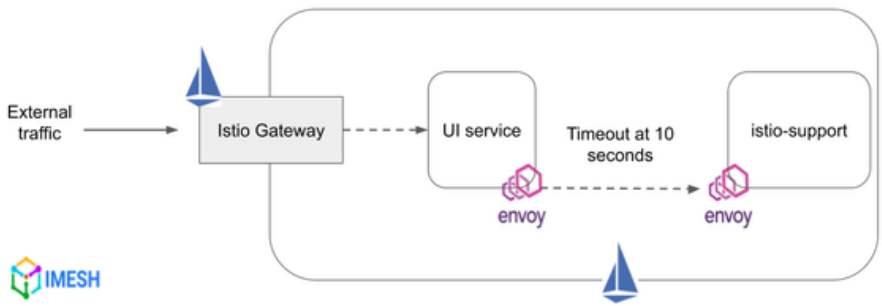


Fig. C - Timeout with Istio

You can implement timeouts in your environment using Istio. Istio allows creating timeout policies and applying them at the source Envoy sidecars.

Below is a sample `VirtualService` that configures a 10-second timeout for requests to `istio-support` service. In other words, calls to `istio-support` service will either fail or succeed in 10 seconds.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: istio-support
spec:
  hosts:
  - istio-support
  http:
  - route:
    - destination:
        host: istio-support
    timeout: 10s
```

- Timeouts should not be short or too long. Short timeouts will result in unnecessarily failed requests, especially when upstream services face transient issues, such as a temporarily overloaded network. Timeouts that are too long cause increased latency, especially if the call waits for a response from a failed service.

- If timeouts have to be configured on traffic to a destination outside the mesh, the destination service should first be added to Istio's internal service registry using `ServiceEntry` resource. Virtual service rules can then be applied to that traffic.

- With Istio, you can easily configure timeouts on traffic to any specific service/subset in the runtime.

Retries

Retry setting specifies the number of times an Envoy proxy should attempt to connect to a service if the initial request fails (refer to Fig.D). It helps to improve service availability when services face temporary issues, like resource contention, network problems, etc.

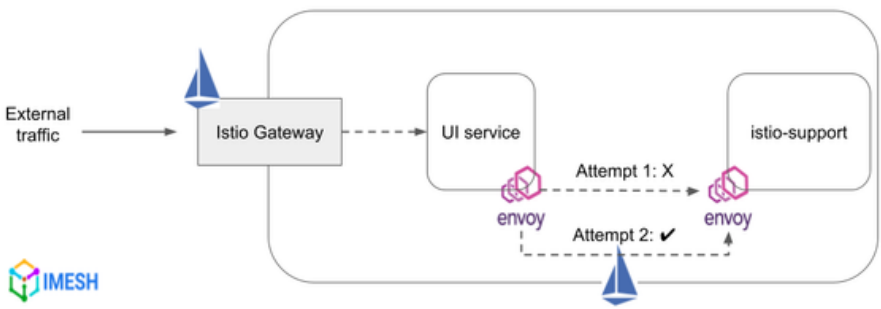


Fig. D - Retries with Istio

The following `VirtualService` sets the maximum number of retries to 4, while calling `istio-support` service.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: istio-support
spec:
  hosts:
  - istio-support
  http:
  - route:
    - destination:
        host: istio-support
    retries:
      attempts: 4
      perTryTimeout: 2s
```

- In the above resource, `attempts` represents the maximum number of retries allowed for a given request. Exceeding `attempts` will activate a circuit breaker.

- `perTryTimeout` defines the timeout per attempt, including the initial call and any retries.

- `retryOn` subfield can be added under `retries` field, which will help to set conditions under which retry takes place. The conditions should be valid HTTP status, and there can be one or more conditions or policies. For example, `retryOn: connect-failure,refused-stream,503` means that Istio will initiate a retry if the upstream service returns any of these HTTP status codes (connect-failure, refused-stream, 503).

Traffic mirroring

Traffic mirroring refers to sending a copy of the live traffic to a mirrored service (see Fig.E). It is useful in testing, monitoring, and analyzing a newly deployed application, before releasing and routing production traffic to it.

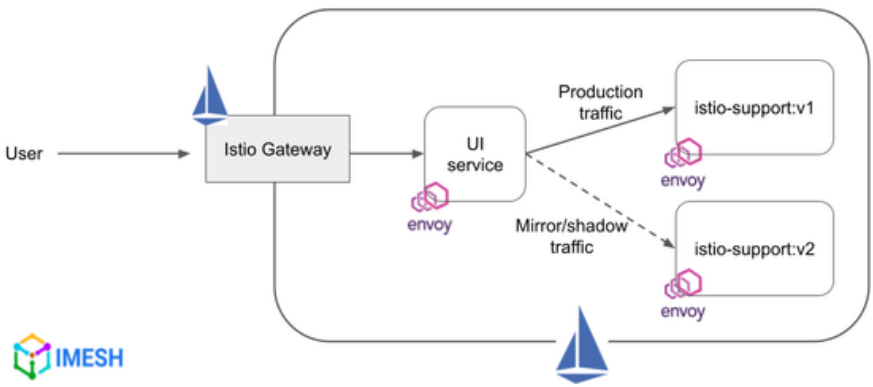


Fig. E - Traffic mirroring with Istio

The mirrored traffic does not affect the performance of the primary service, as it is separate from the main flow of requests served by the primary service. Also, responses from mirrored services are discarded.

The following route rule sends 100% of the traffic to `v1` and then to `istio-support:v2`.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: istio-support-mirror
spec:
  hosts:
    - istio-support
  http:
  - route:
    - destination:
        host: istio-support
        subset: v1
      weight: 100
    mirror:
      host: istio-support
      subset: v2
    mirrorPercentage:
      value: 100.0

---

apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: istio-support
spec:
  host: istio-support
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

The `value` field under `mirrorPercentage` allows users to further control the traffic by sending a fraction of the requests, instead of mirroring all requests.

#2 - Handling traffic at the edge using Istio

Many architects and DevOps engineers keep on extending the capacity of API gateways to handle traffic at the edge. However, it is high time for them to think of a radical solution, like Istio, to replace gateways and simplify the network complexity and achieve zero trust security. Istio can do a much better job managing the API gateway and act as an Ingress controller.

There can be several implementation but we propose two important ones:

1. Managing your existing API gateway with Istio.
2. Use Istio Ingress Gateway as the default API gateway.

Managing API gateway with Istio

API gateway with Envoy proxy to manage multicloud workloads

Let say you are using or invested into API gateway such as Mulesoft, Kong or Gravitee API gateway solution. To manage the API gateway, along with all the workloads, from a central plane, you need to enable Istio to handle the 3rd party API gateway resource as well.

An Envoy proxy can be attached to the API gateway (see Fig.F below). The idea is that the API gateway receives the traffic and processes it, but when the traffic goes out from the API gateway and into the cluster services, it goes through an Envoy proxy attached to it.

With the sidecar proxy, all the traffic from the edge to the services can be secured with authentication and authorization policies in Istio, and also the tracing of traffic logs can be collected or visualized in the central plane.

When to use this architecture	Pros	Cons
Early phase of Istio implementation.	With Envoy proxies, S2S communication can also be secured with mTLS in just seconds from the Istio control plane.	Configuration can be very tricky.
API gateway handles a lot of traffic.	Traffic management functionalities can be distributed- let API gateway perform advanced traffic management (API management, billing, etc) and Istio performs L7 routing.	
Heavily invested into 3rd party API gateway.	Istio supports Kubernetes natively and also supports VMs.	
	All the east-west and north-south traffic can be managed and secured easily from Istio.	
	End-to-end observability in a single plane	

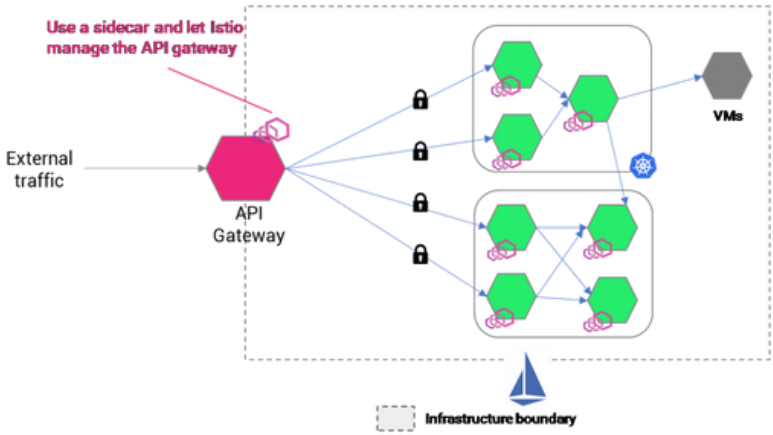


Fig F - Envoy proxy attached to API gateway to manage multicloud workloads

In the above scenario, Istio is used to inject sidecars to all services inside a cluster and apply security and network logic selectively from the central plane. We can set up Istio to also get mutlicluster visibility in terms of access logs, tracing, and performance metrics of each service.

(Read more about the implementation architecture of Istio and API gateway: Implementation Architecture of Istio and API gateway.)

Istio Ingress Controller as an API gateway

Istio Ingress Gateway can be used as an application load balancer and also API gateway. Istio Ingress supports Kubernetes workloads natively and It can be extended to handle complicated networking functions for legacy workloads as well (see Fig.G). Once you deploy it, Istio creates a network load balancer that will distribute the load evenly among the nodes. Once the networking load balancers parse the HTTP requests, the L7 processing (such as filtering, routing rules, etc) happens at the Istio Ingress Gateway.

Istio Ingress Gateway provides virtual services and destination rules to implement advanced networking functions such as retries, timeouts, circuit breakers, failovers, etc. You can also use Istio to implement a canary deployment strategy with Argo Rollouts.

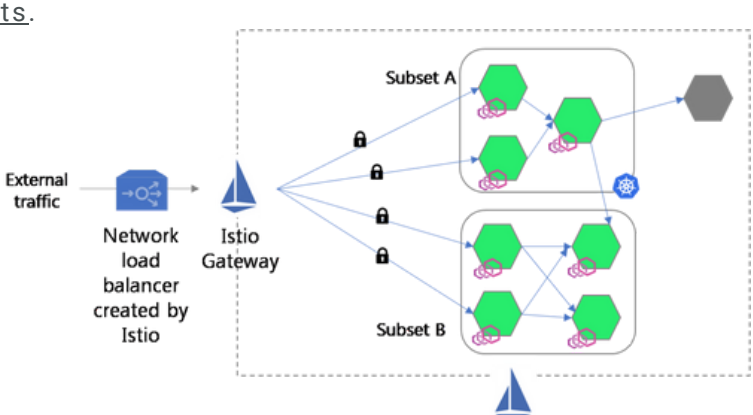


Fig G: Istio Ingress Gateway implementation as a simple application load balancer or API gateway.

(Read more about using Istio as application load balancer: How to use Istio Ingress Gateway as Application Load Balancer (ALB))

#3 - Chaos engineering with Istio

Chaos engineering is a discipline that helps teams build resilient systems by proactively injecting failures into production environments. The goal of chaos engineering is to identify and mitigate weaknesses in a system before they cause outages or other problems in production.

Istio helps users perform chaos engineering through fault injection. Fault injection is a testing method that includes introducing errors while forwarding HTTP requests to the destination specified in a route. Istio lets DevOps and SREs test the resiliency and failure recovery capacity of applications by injecting faults.

With Istio, faults can be injected at the application layer. That is, more relevant failures can be injected, such as HTTP error codes, instead of tinkering in L4 or L3 layers such as killing pods, delaying packets, or corrupting packets at the TCP layer. Istio lets users inject two types of faults using `VirtualService` resource:

- **Delays**: Used to delay requests to upstream services and simulate network latency or an overloaded upstream service (see Fig.H).

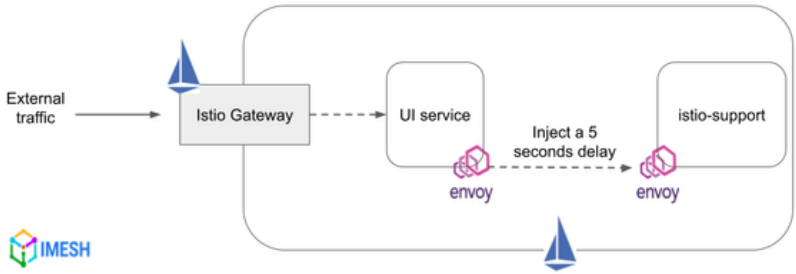


Fig H - Fault injection by delaying requests using Istio

- **Aborts**: Used to abort HTTP request attempts and return error codes to downstream service, in order to simulate a faulty upstream service (refer to Fig.I).

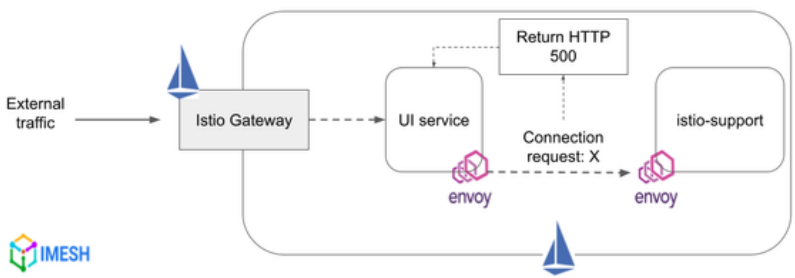


Fig I - Fault injection by aborting request forwarding using Istio

The following `VirtualService` will inject a 5 seconds delay on all the requests going to `istio-support` service.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: istio-support-delay
spec:
  hosts:
  - istio-support
  http:
  - fault:
      delay:
        percentage:100
        fixedDelay: 5s
```

Similarly, the below `VirtualService` resource configures `istio-support` service to return an HTTP 500 error for each received request.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: istio-support-500
spec:
  hosts:
    - istio-support
  http:
    - route:
        - destination:
            host: istio-support
      fault:
        abort:
          percent: 100
          httpStatus: 500
```

Note: A fault rule must have a delay or abort or both. Also, simultaneously specifying delay and abort faults does not create any dependencies between them.

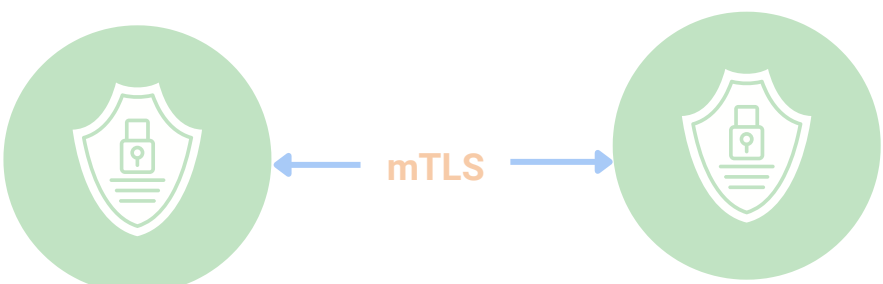
#4 - Istio for implementing mTLS

Authenticating inbound traffic is crucial in the current dynamic threat landscape. With applications deployed and distributed over the cloud, data is prone to cyber attacks, such as man-in-the-middle (MITM), IP spoofing, packet sniffing, denial-of-service (DoS), etc. And this is why a strong authentication mechanism like mTLS is essential. Istio not only helps DevOps engineers implement mTLS, but it can also perform certificate management and rotation at scale.

What is mTLS?

Mutual Transport Layer Security (mTLS) is a cryptographic protocol designed to authenticate two parties and secure their communication in the network. mTLS protocol is an extension of TLS protocol where both the parties- web client and web server- are authenticated. The primary aim of mTLS is to achieve the following:

- Authenticity: To ensure both parties are authentic and verified
- Confidentiality: To secure the data in the transmission
- Integrity: To ensure the correctness of the data being sent



How to enable mTLS and certificate rotation using Istio service mesh

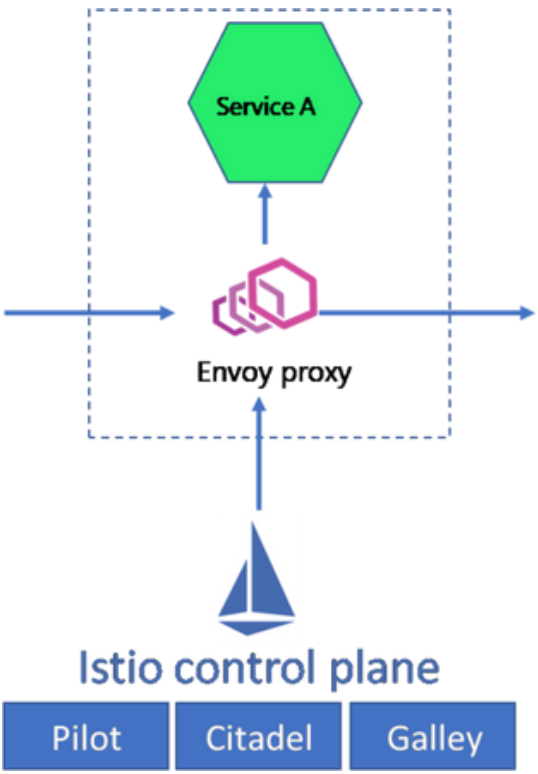


Fig - mTLS implementation in Istio

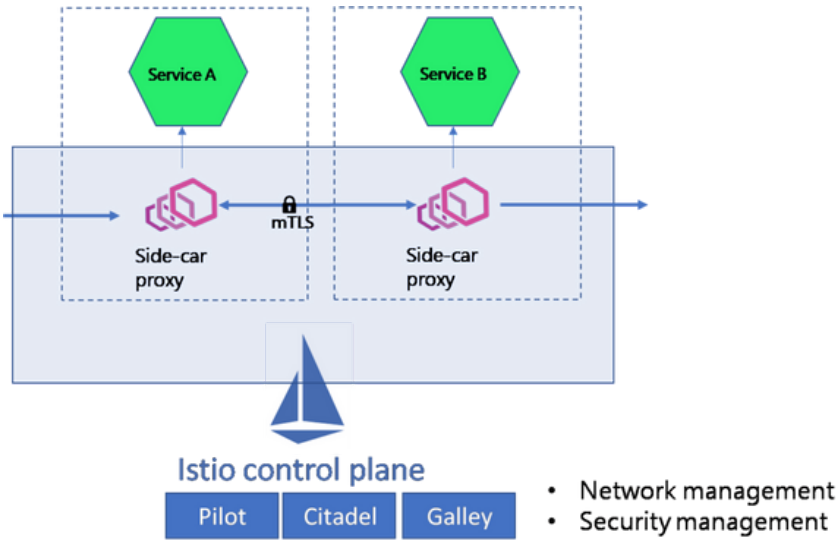
Though Istio supports multiple authentication types, it is best known for implementing mTLS to applications hosted over the cloud, on-prem, or Kubernetes infrastructure. The Envoy proxy acts as Policy Enforcement Points (PEP); you can implement mTLS using the peer-to-peer (p2p) authentication policy provided by Istio and enforce it through the proxies at the workload level.

Example of p2p authentication policy in Istio to apply mTLS to `demobank` app in the 'istio-nm' namespace:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "mTLS-peer-policy"
  namespace: "istio-nm"
spec:
  selector:
    matchLabels:
      app: demobank
  mtls:
    mode: STRICT
```

The working mechanism of mTLS authentication in Istio is as follows:

1. At first, all the outbound and inbound traffic to any application in the mesh is re-routed through the Envoy proxy.
2. So the mTLS happens between the client-side Envoy proxy and the server-side Envoy proxy.
3. The client-side Envoy proxy would try to connect with the server-side Envoy proxy by exchanging certificates and proving their identity.
4. Once the authentication phase is completed successfully, a TCP connection between the client and service side Envoy proxy is established to carry out encrypted communications.

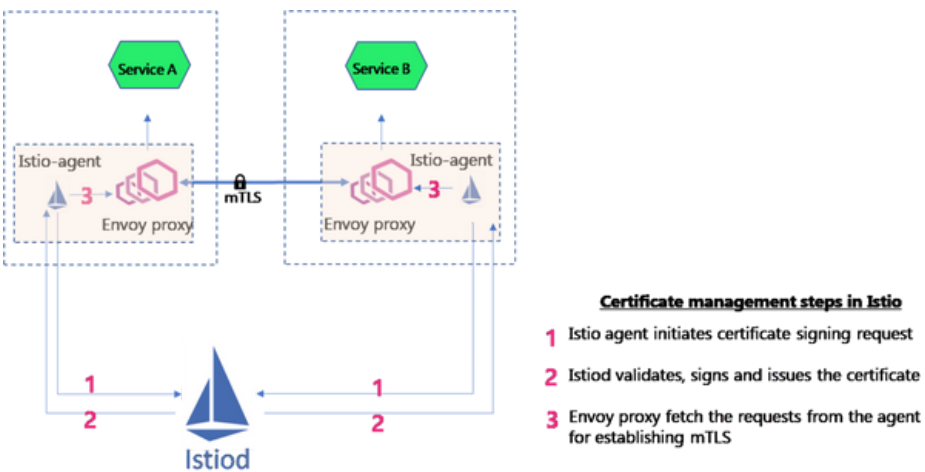


Note that mTLS with Istio can be implemented at all levels- application, namespace, or mesh-wide.

Certificate management and rotation in Istio service mesh

Istio provides a stronger identity by issuing X.509 certificates to Envoy proxies attached to applications. The certificate management and rotation are done by an Istio agent running in the same container as the Envoy proxy. The Istio agents talk to the Istiod- the control plane of Istio- to effectively circulate the digital certificates with public keys. Below are the details phases of certificate management in Istio:

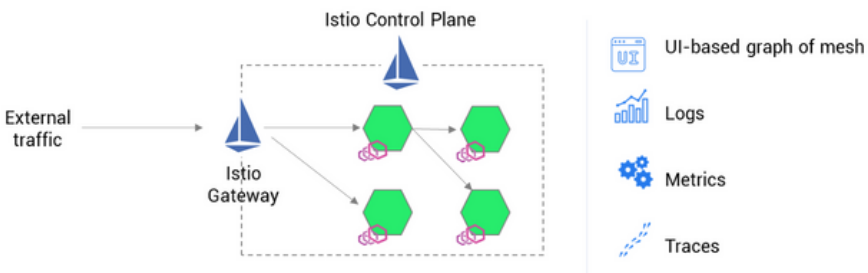
1. Istio agents generate public key pairs (private and public keys) and then send the public key to the Istio control plane for signing. This is called a certificate signing request (CSR).
2. Istiod has a component (earlier Galley) that acts as the CA. Istiod validates the public key in the request, signs, and issues a digital certificate to the Istio agent.
3. When mTLS connection is required, Envoy proxies fetch the certificate from the Istio agent using Envoy secret discovery service (SDS) API.
4. The Istio agent observes the expiration of the certificate used by the Envoy. Upon the certificate's expiry, the agent initiates a CSR to Istiod.



(Read more about mTLS and how to implement it with Istio: What is mTLS | How to implement it using Istio?)

#5 - Network and traffic observability using Istio

One of the primary responsibilities of SREs in large organizations is to monitor the golden metrics of their applications, such as CPU utilization, memory utilization, latency, and throughput. Istio helps DevOps engineers and SREs gain visibility over the network topology by emitting a lot of metrics, logs, traces about the application performance, network and traffic (refer the image below. Istio helps SREs in improving their ability for incident response and troubleshooting in case of infrastructure anomalies.



Istio supports open source monitoring, logging, and tracing systems for observability

You can use Istio to observe the performance and behavior of all your microservices in your infrastructure. Since Istio supports integrations with many open source and closed source monitoring and logging systems, it is easy for SREs to plugin the metrics, logs and traces from Istio for further analysis.

One of the common usecase which many SREs and Ops team like to implement is integration of Istio with Prometheus and Grafana. Istio supports Prometheus and Grafana natively.

Prometheus and Grafana are in the addon folder in the Istio directory and it is extremely easy to configure them. You just need to go to the path ***istio-(your_version)/samples/addons*** and enter the following commands to deploy them:

```
kubectl apply -f prometheus.yaml
kubectl apply -f grafana.yaml
```

The add-on YAMLs are applied to istio-system namespace by default.

To see a full tutorial on configuring Istio, Prometheus, and Grafana for monitoring, read the blog: How to configure Istio, Prometheus and Grafana for monitoring.

Network topology with Istio

Istio uses Kiali to visualize the mesh architecture and communication between services within the mesh. Kiali provides a web-based GUI to visualize the traffic flow between pods, and helps DevOps and SREs understand the dependency and topology graph (see image below).

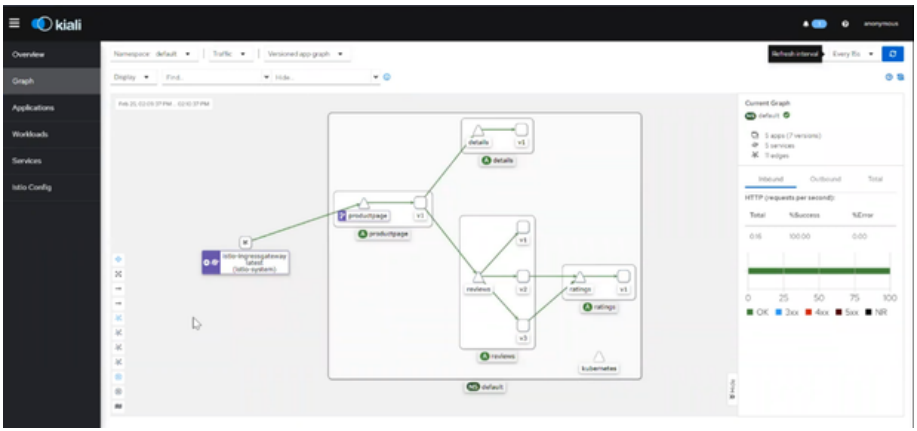


Fig J - Service graph in Kiali

You can deploy Kiali by following the same method we described above — for Prometheus and Grafana:

```
kubectl apply -f kiali.yaml
```

Once they are deployed, use the below command to open the Kiali dashboard:

```
istioctl dashboard kiali
```

The above code will return a URL as you see below, which can access the dashboard:

```
C:\D_Drive\imesh\webinar\istio\istio-1.17.1>istioctl dashboard kiali
http://localhost:20001/kiali
```

Accessing the URL from the web browser will open the dashboard (see Fig.K) with all the namespaces:

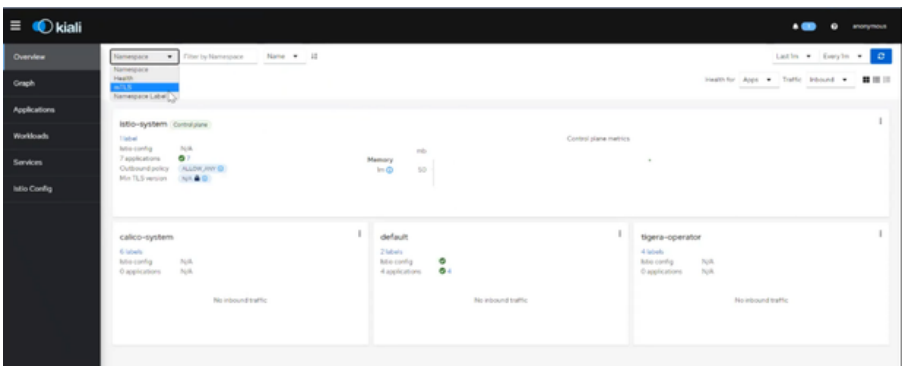


Fig K - Kiali dashboard UI

To see the service graph and visualize the traffic flow between services, you can go to “Graph” and then select the “default” namespace where we have deployed the applications.

It will open a service graph where you can see the request coming from ingress to the product page, which then flows to other microservices:

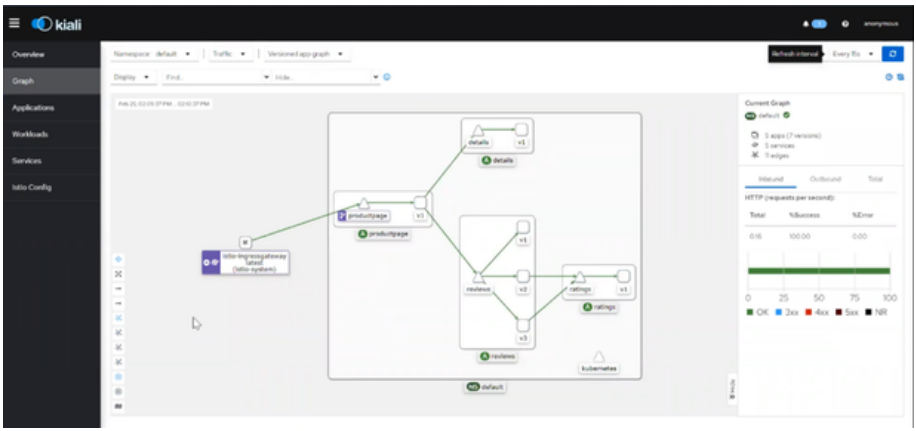


Fig - Kiali service graph

Integrations of Istio

Istio is open source and integrates with almost all the open source providers in the CNCF landscape. The community is also active in providing support towards the integrations.

Security	Networking	Observability	DevOps	Infrastructure
Google SSO, SAML, Okta, OAuth 2.0	Ingress controller	Prometheus, Datadog, Skywalking	Argo CD	Kubernetes
Vault	Envoy gateway	Splunk, Grafana Loki	Spinnaker	VMs
Lets Encrypt, AWS Cert manager	API gateways	Jaeger	Jenkins	Google/Azure/AWS/IBM cloud

Conclusion

The complexity of managing the traffic and cloud network will become more complex for DevOps and architects as the applications scale to meet demands. Extending API gateways beyond their capacity will eventually cease to become the ideal solution. They will need a powerful platform like Istio to replace gateways and to transform the network and traffic in the cloud.

Istio is complex, like Kubernetes. If you are deploying Istio in a production environment, it is highly recommended to talk to an Istio expert beforehand. Because many subtle errors can creep in while configuring service definitions or setting up Istio, particularly when you have applications deployed across multiple namespaces, written by developers with varying levels of experience.

At IMESH, we help enterprises safely deploy Istio into the production environment. We take care of all the operational hassles, such as Istio lifecycle management and optimization. Book a free pilot if you are interested in knowing more.

Authors

Ravi Verma, CTO, IMESH

Ravi is the CTO of IMESH. Ravi, a technology visionary, brings 12+ years of experience in software development and cloud architecture in enterprise software. He has led R&D divisions at Samsung and GE Healthcare and architected high-performance, secure and scalable systems for Baxter and Aricent. His passion and interest lie in network and security. Ravi frequently discusses open-source technologies such as Kubernetes, Istio, and Envoy Proxy from the CNCF landscape.

Anas, Product Marketing Manager, IMESH

Anas is a curious marketer with a passion for exploring the cloud-native landscape. With a deep understanding of CI/CD, Kubernetes, and Istio, he attempts to make DevOps engineers and architects be aware and adopt open-source and cloud technologies. Anas spearheads the product marketing functions at IMESH and acts as a bridge between the Product and Marketing teams, facilitating the seamless adoption of cloud-native solutions. He loves to play football and try out tutorials in a demo cluster.

About IMESH

IMESH offers Kubernetes-native application network and security platform to manage multi-cloud and hybrid cloud environments. The IMESH platform is built on top of Istio service mesh and Envoy API gateway and helps cloud, platform and security teams to make Kubernetes application more secure, manageable, and reliable.

Visit: https://imesh.ai/
email: contact@imesh.ai