Service Mesh with Istio

Cloud Native Italy @ 127.0.0.1 - November 2020

Diego Braga, *Solution Architect* @ Kiratech

Lorenzo Pistone, *DevOps Engineer* @ Kiratech

AGENDA

1	THE PROBLEM
2	THE SOLUTION: SERVICE MESH
3	WHAT SERVICE MESH ADDRESSES
4	WHAT SERVICE MESH DOES NOT ADDRESS
5	ISTIO ARCHITECTURE
6	ISTIO AUTHENTICATION & AUTHORIZATION

THE PROBLEM

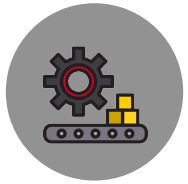
Who can deal with whom?
Are there any hidden dependencies?

What about TLS and certificates?

How many seconds does every call last?
Is there network latency? Where?

What if a microservice is not reachable or it answers with error?

THE SOLUTION: SERVICE MESH

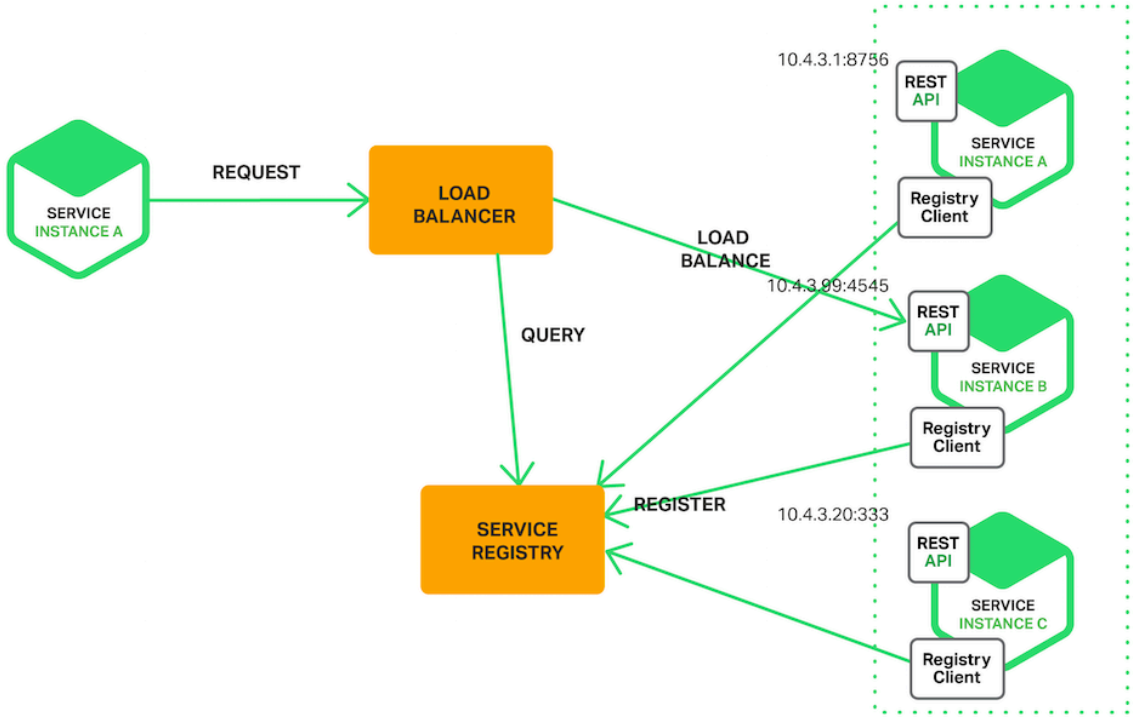


A service mesh is a configurable, low-latency infrastructure layer designed to handle a high volume of network-based interprocess communication among application infrastructure services using application programming interfaces (APIs) - ***a service mesh is like software-defined networking for the application layer***.

A service mesh ensures that communication among containerized and often ephemeral application infrastructure services is fast, reliable, and secure.

The mesh provides critical capabilities including ***service discovery, load balancing, encryption, observability, traceability, authentication and authorization***, and support for the circuit breaker pattern.

WHAT SERVICE MESH ADDRESSES (1/3)



Most orchestration frameworks already provide Layer 4 (transport layer) load balancing. A service mesh implements more sophisticated Layer 7 (application layer) load balancing, with richer algorithms and more powerful traffic management. ***Load-balancing parameters can be modified via API, making it possible to orchestrate blue-green or canary deployments.***

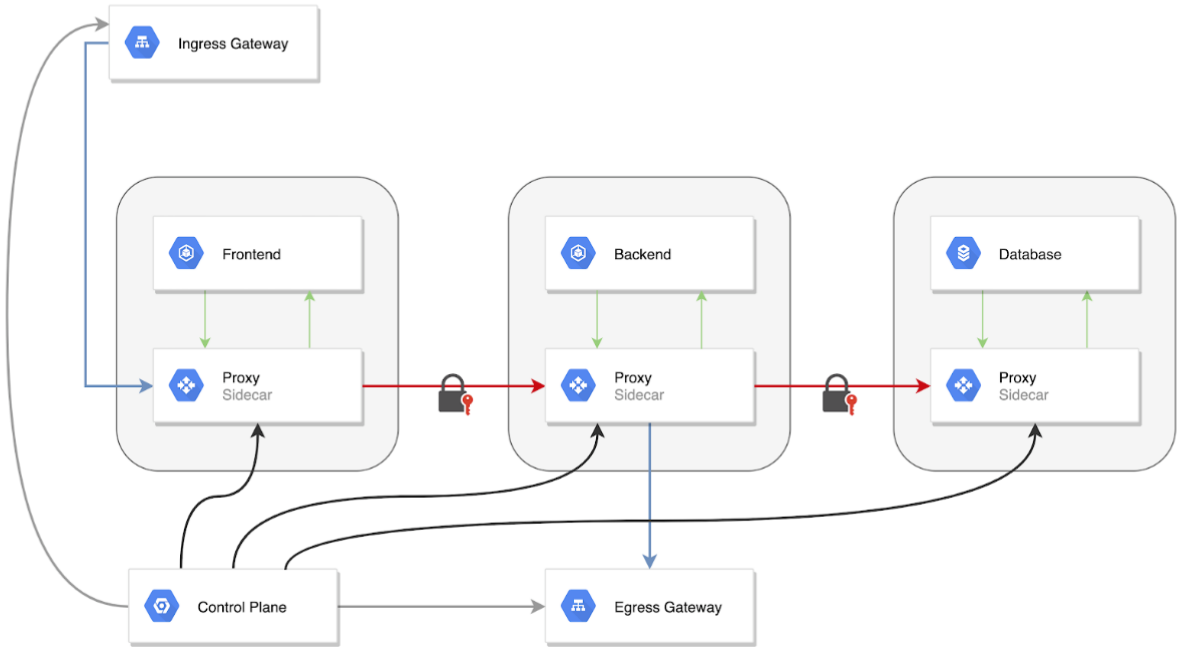
SERVICE DISCOVERY

The container orchestration framework keeps a list of instances that are ready to receive requests and provides the interface for DNS queries. ***Service mesh works with a service discovery protocol to detect services as they come up***.

LOAD BALANCER

Service Mesh keeps track of which instances of various microservices distributed across the infrastructure are "healthiest" and ***keep track of which instances are responding slowly to service requests and send subsequent requests to other instances***. The service mesh can do similar work for network routes, noticing when messages take too long to get to their destination, and take other routes to compensate.

WHAT SERVICE MESH ADDRESSES (2/3)



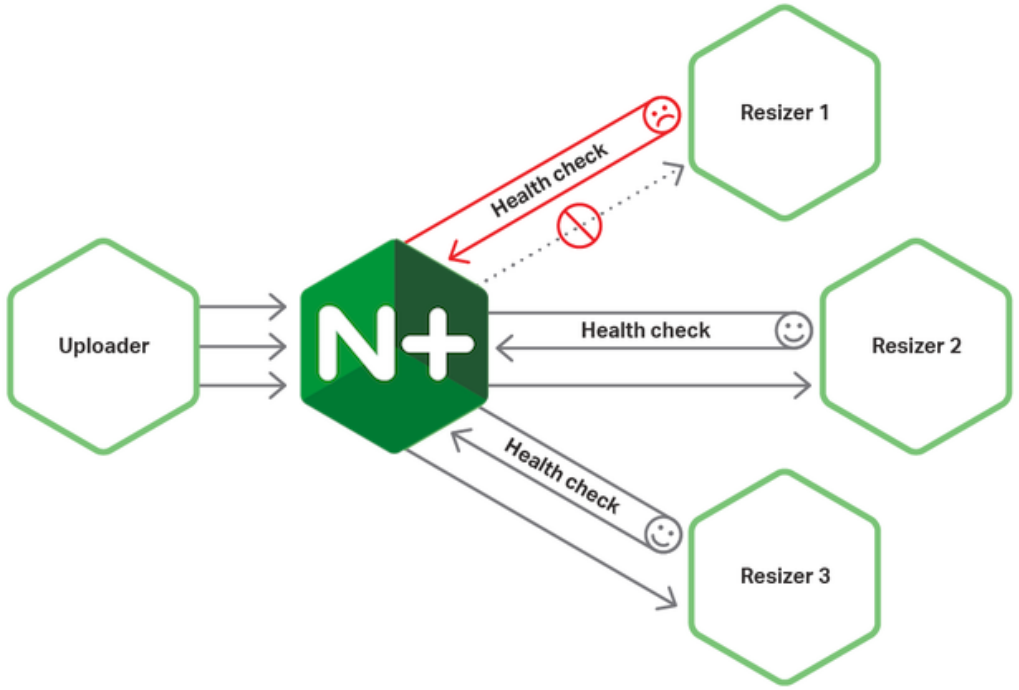
ENCRYPTION

Service mesh can encrypt and decrypt requests and responses. It can also improve performance by prioritizing the reuse of existing, persistent connections, which reduces the need for the computationally expensive creation of new ones. The most common implementation for encrypting traffic is mutual TLS (mTLS)

AUTHENTICATION AND AUTHORIZATION

Service mesh can authorize and authenticate requests made from both outside and within the app, sending only validated requests to instances.

WHAT SERVICE MESH ADDRESSES (3/3)



SUPPORT FOR THE CIRCUIT BREAKER PATTERN

The service mesh can support the circuit breaker pattern, which isolates unhealthy instances, then gradually brings them back into the healthy instance pool if warranted.

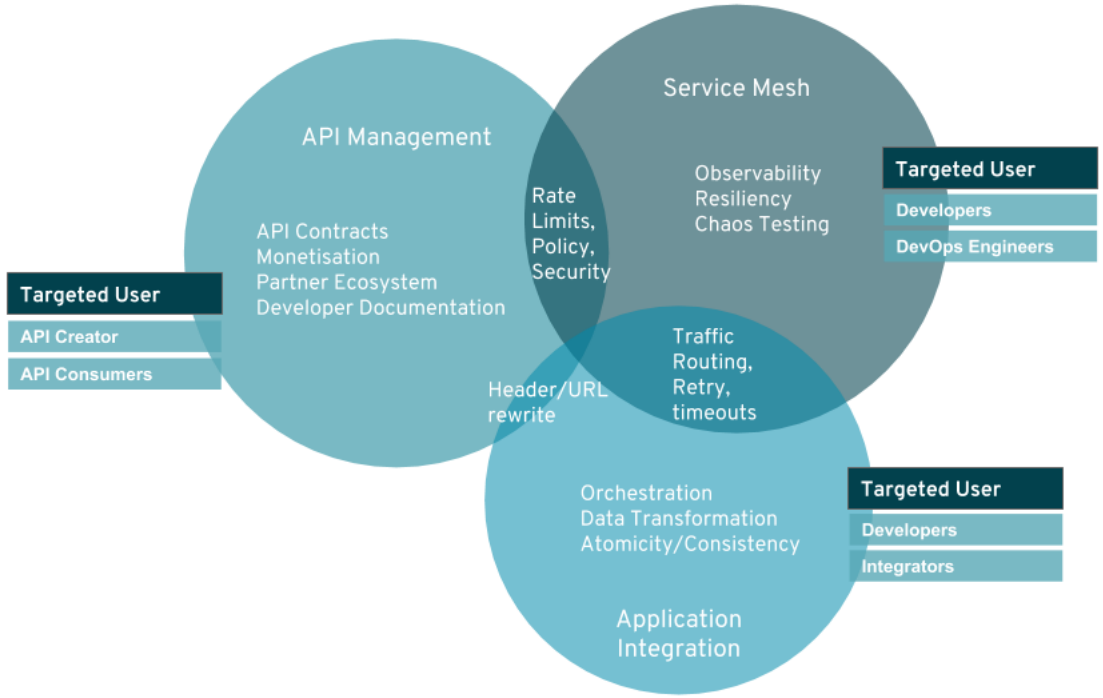
WHAT SERVICE MESH DOES NOT ADDRESS

API GATEWAY

A service mesh's primary purpose is to manage internal service-to-service communication, while an API Gateway is primarily meant for external client-to-service communication.

TRACING

You will need to add logic in your application to propagate tracing headers from incoming to outgoing requests to gain full benefit from service mesh distributed tracing capability.

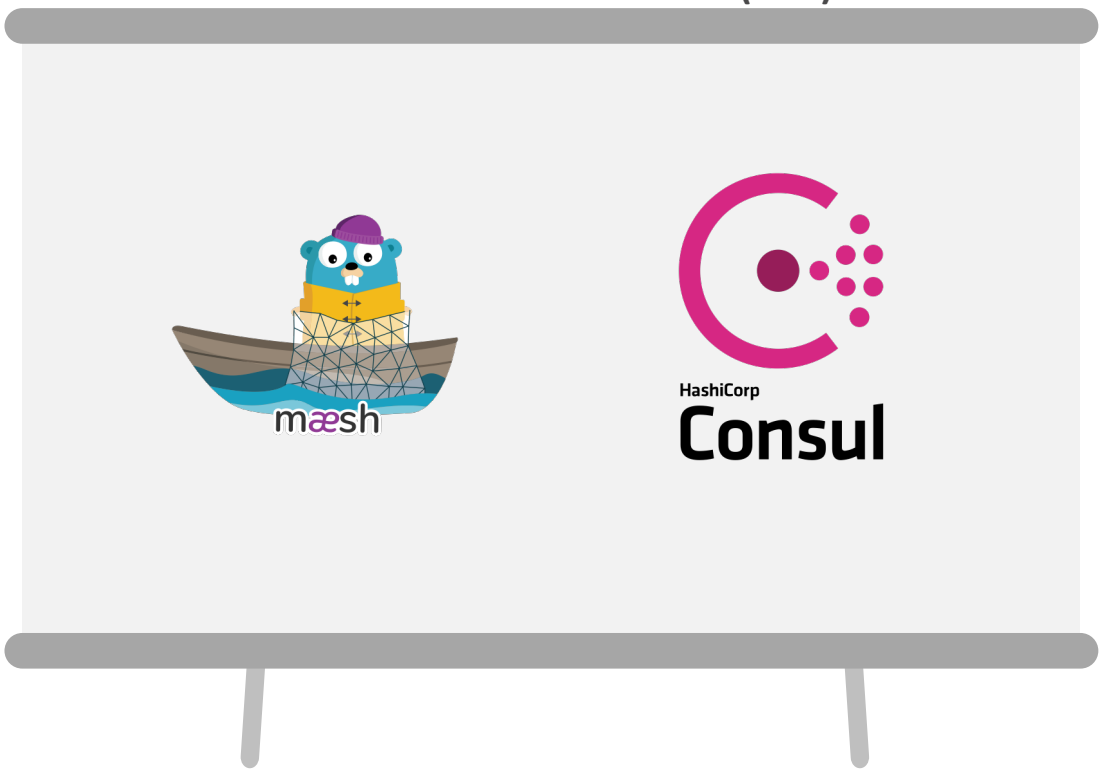


SERVICE MESH SOLUTIONS

Many tools implement Service Mesh using different approaches

SIDECAR CONTAINER

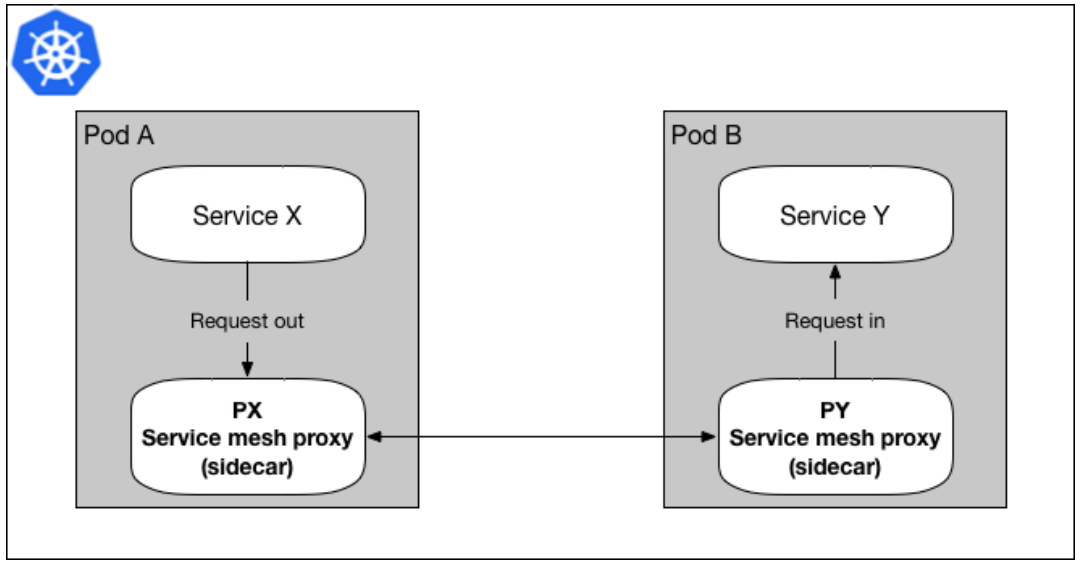
SERVICE MESH INTERFACE (SMI)



SERVICE MESH SOLUTIONS – SIDECAR PROXY

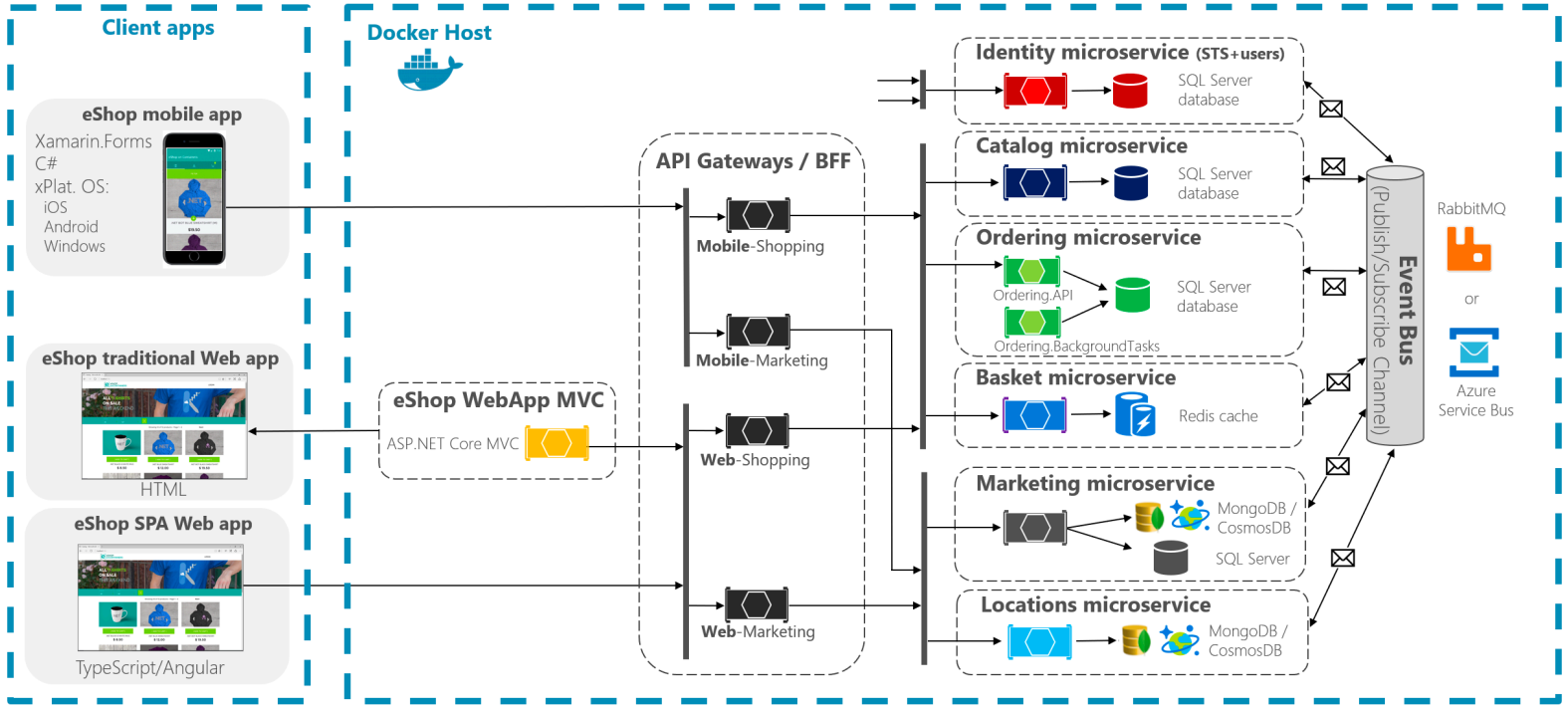
ARCHITECTURE

The proxy is placed in the same pod as the application container is running. From outside the pod, the application and the sidecar will share the same networking namespace and their IP. Inside the pod, they're still two separate containers so they have separate file system and process level isolation so this gives them the right kind of isolation and uniformity from the network where you can have your application built and deployed separately and thus proxy have its own build lifecycle.



SERVICE MESH – MICROSERVICES HETEROGENEITY

eShopOnContainers reference application
(Development environment architecture)



POLYGLOT APPLICATIONS

Most of the microservices are built using polyglot languages since microservices encourage developers to write applications using languages that make it more efficient and easier to maintain.
That means you will have applications running Java / Go / NodeJS / Python. If security is implements using libraries, the complexity is growing because security is embedded in the applications, patching must be applied to all the frameworks. The application developer is responsible for enforcing security.

SERVICE MESH – MICROSERVICES SECURITY – AUTHENTICATION

DEFINITION

Act of verifying or validating your end-user or the client who is currently making a request.

In microservices it's the combination of both because the client which is connecting to you might be a microservice in your network, but it might be making a request on behalf of a end-user request.
Normally authentication is made by creating a login and a password (most of the modern applications delegate that to identity management solutions) where the login password goes to the solution and it gives back a JWT token or a JSON token and that token acts as the identity.
As the request moves along the cluster, the token is the identity which verifies the user

SERVICE MESH – MICROSERVICES SECURITY – AUTHORIZATION

DEFINITION

Act of verifying or validating if the end-user or the client is allowed to make the request or has the right permission to access that resource.

In a multi-tenant environment, if the application has multiple users or quality of services, normally you need some sort of authorization.
Authorization is the step which follows closely once you have authenticated the user.

ISTIO

MATURE

Started 3 years ago

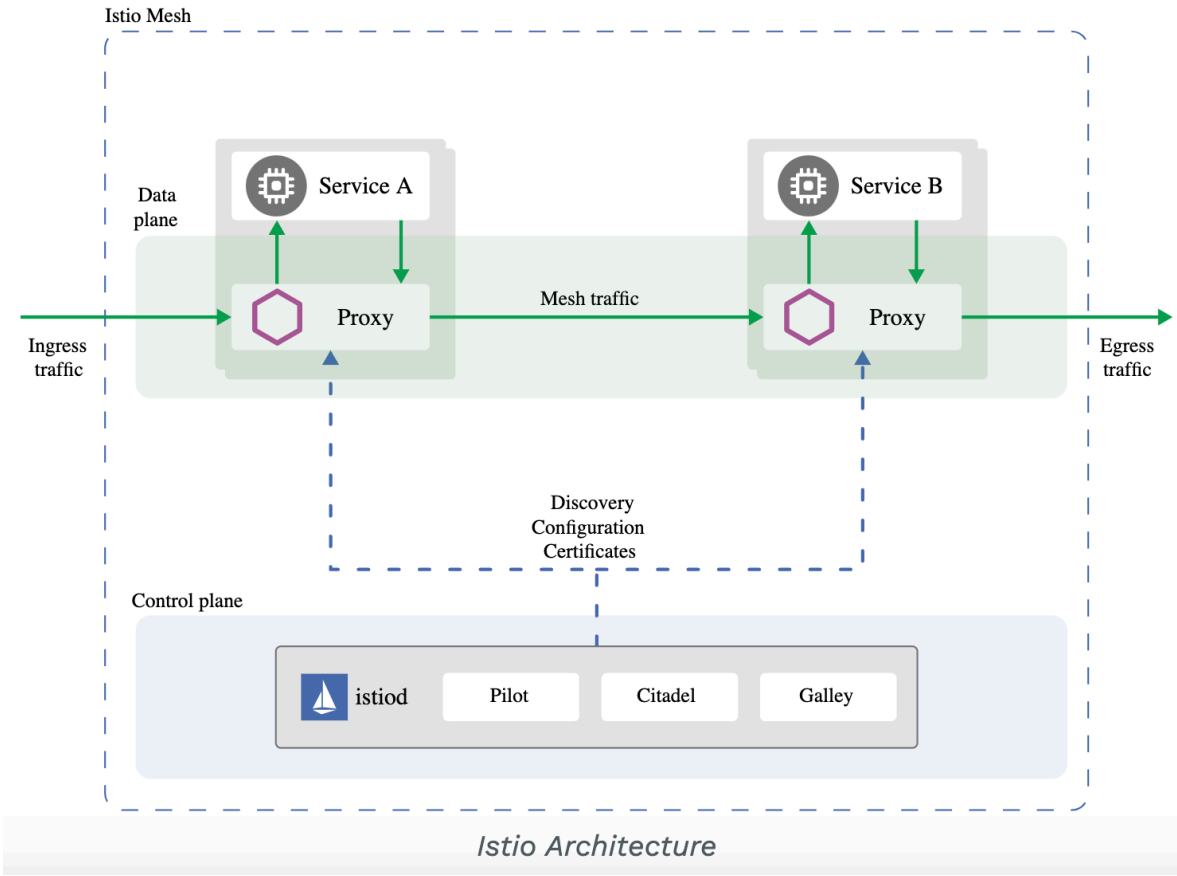
STABLE

At v1.7.4, battle tested

ENTERPRISE GRADE

Chosen by Google: https://cloud.google.com/istio/?hl=it
And RedHat: https://www.redhat.com/it/topics/microservices/what-is-istio

ISTIO ARCHITECTURE (1/2)



Istio Architecture

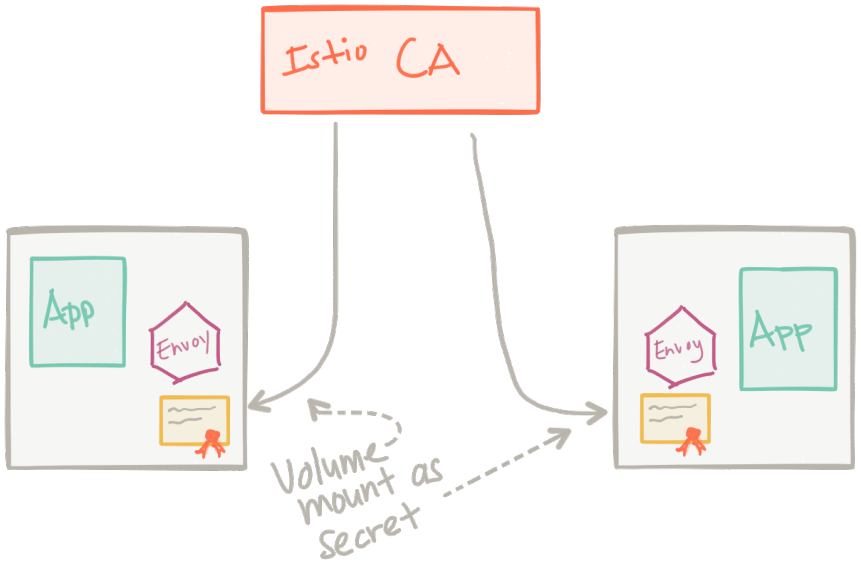
DATA PLANE

Touches every packet/request in the system. Responsible for service discovery, health checking, routing, load balancing, authentication/authorization, and observability.

CONTROL PLANE

Provides policy and configuration for all of the running data planes in the mesh. Does not touch any packets/requests in the system. The control plane turns all of the data planes into a distributed system.

ISTIO ARCHITECTURE (2/2)



NETWORK SECURITY

Istio's Citadel component in the control plane handles getting the certificates and keys onto the application instances. **Citadel can generate the certificates and keys needed for each workload to identify itself, as well as rotate certificates periodically** so that any compromised certificates have a short life. Using these certificates, Istio-enabled clusters have automatic mutual TLS. You can plug in your own CA provider root certificates as needed as well.

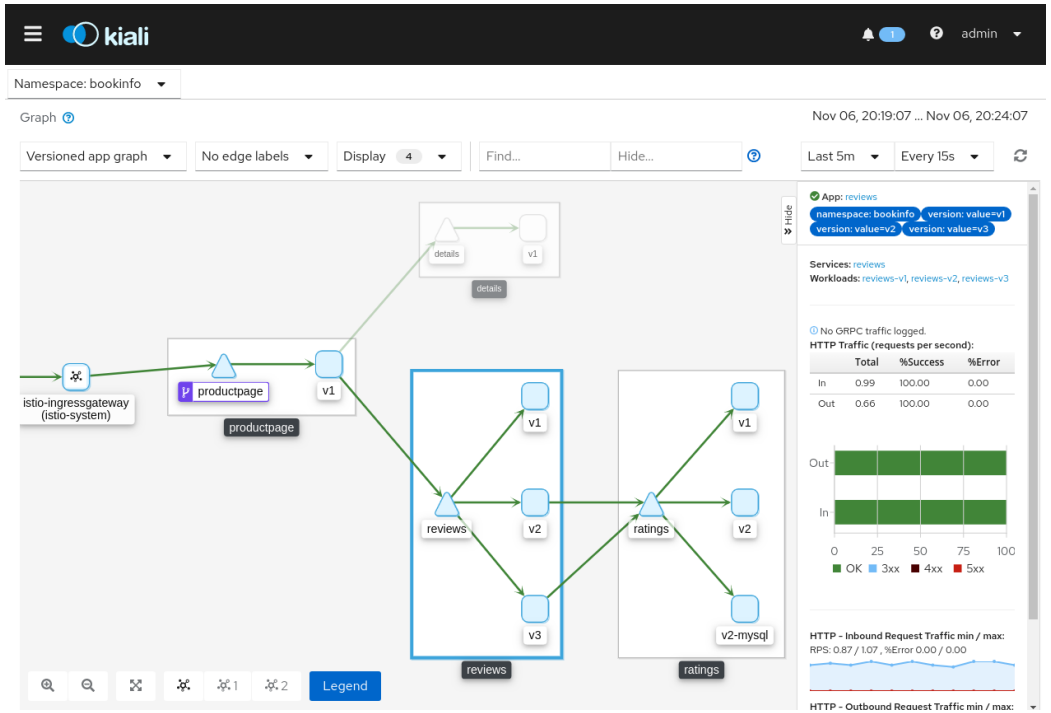
Citadel is exposing a secret discovery service API which allows to fetch certificates and secrets so you don't have to mount those certificates on the file system.

ISTIO OBSERVABILITY

OBSERVABILITY

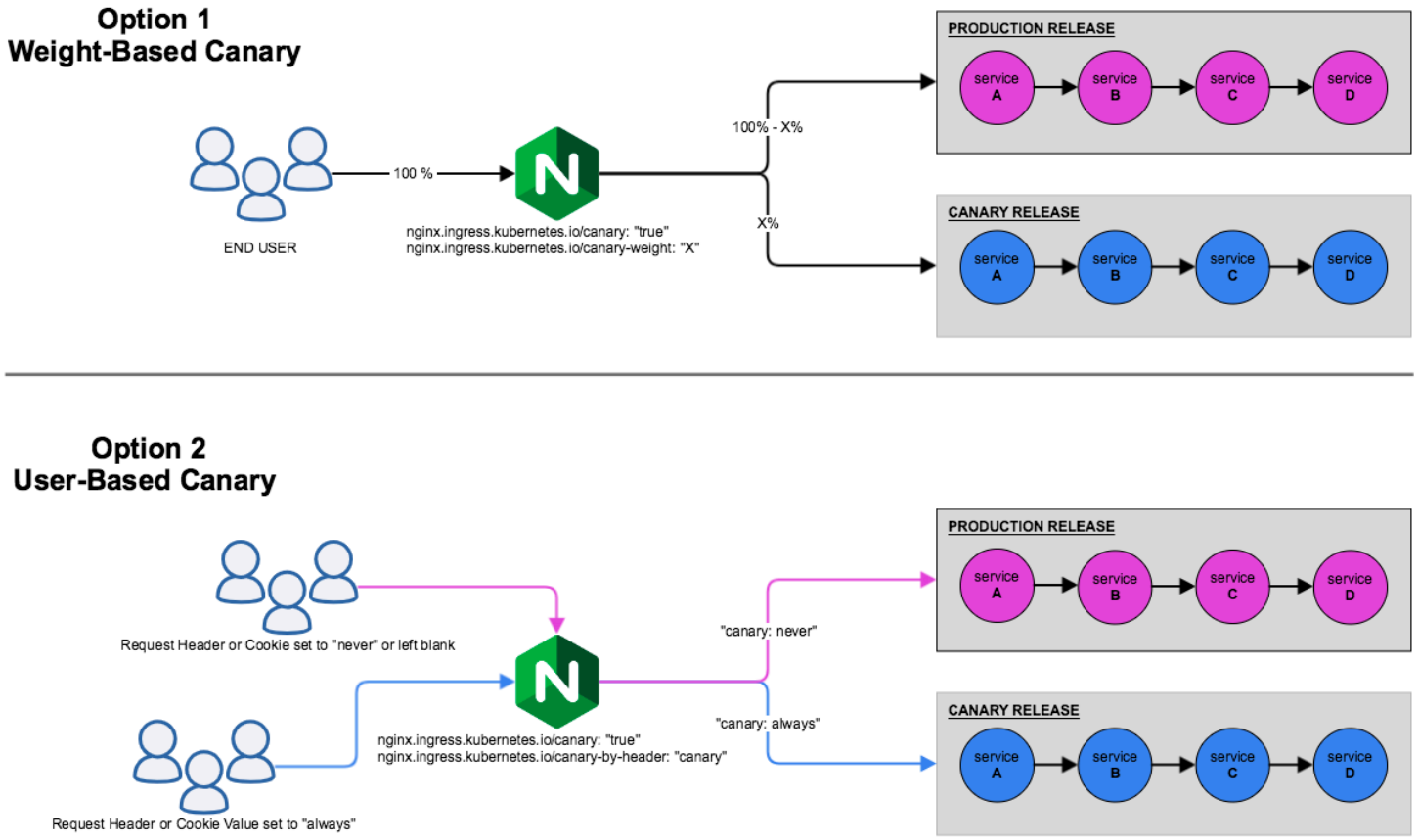
Istio generates the following types of telemetry in order to provide overall service mesh observability:

- **Metrics**: based on the four "golden signals" of monitoring (latency, traffic, errors, and saturation).
- **Distributed Traces**: Istio generates distributed trace spans for each service, providing operators with a detailed understanding of call flows and service dependencies within a mesh.
- **Access Logs**. As traffic flows into a service within a mesh, Istio can generate a full record of each request, including source and destination metadata. This information enables operators to audit service behavior down to the individual workload instance level.



Kiali is an observability console for Istio with service mesh configuration capabilities. It helps you to understand the structure of your service mesh by inferring the topology, and also provides the health of your mesh. Kiali provides detailed metrics, and a basic Grafana integration is available for advanced queries. Distributed tracing is provided by integrating Jaeger.

ISTIO TRAFFIC MANAGEMENT – CANARY RELEASE



One of the benefits of the Istio project is that it provides the control needed to deploy canary services.

The idea behind canary deployment (or rollout) is to ***introduce a new version of a service by first testing it using a small percentage of user traffic***, and then if all goes well, increase, possibly gradually in increments, the percentage while simultaneously phasing out the old version.

If anything goes wrong along the way, ***we abort and rollback to the previous version***. In more sophisticated schemes the rollout it can be based on the region, user, or other properties of the request.

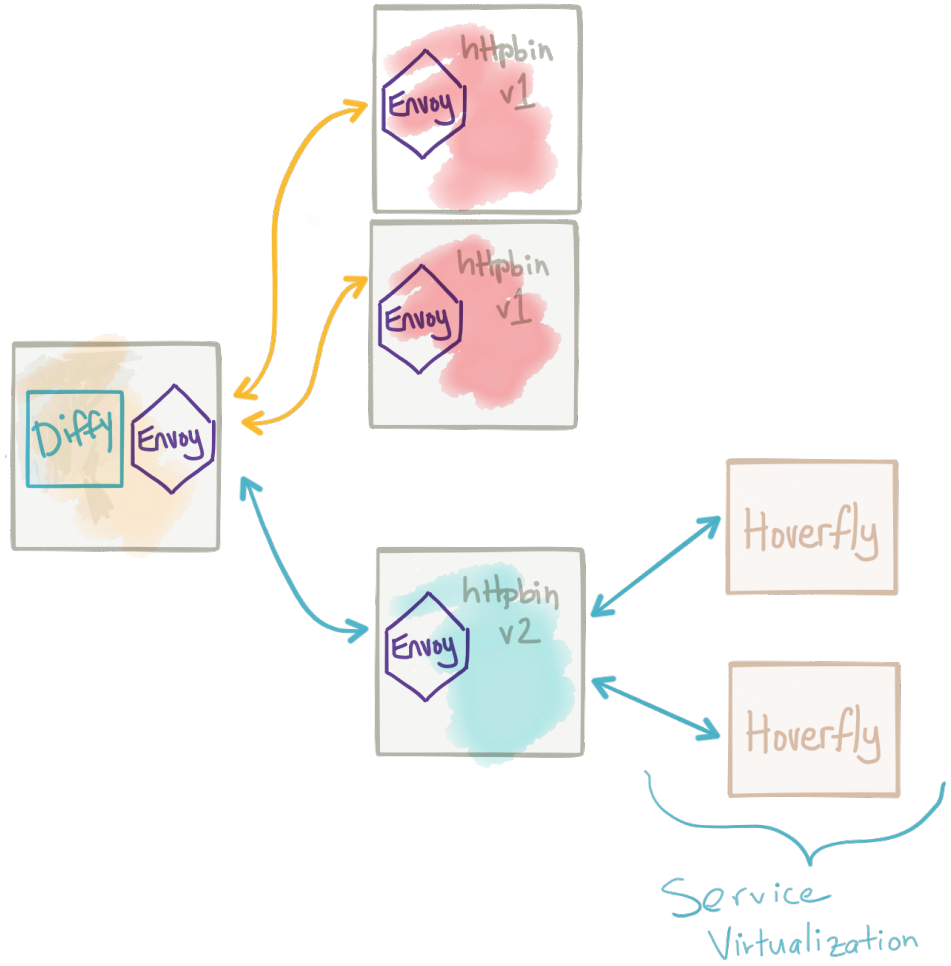
ISTIO TRAFFIC MANAGEMENT – MIRRORING

TRAFFIC MIRRORING

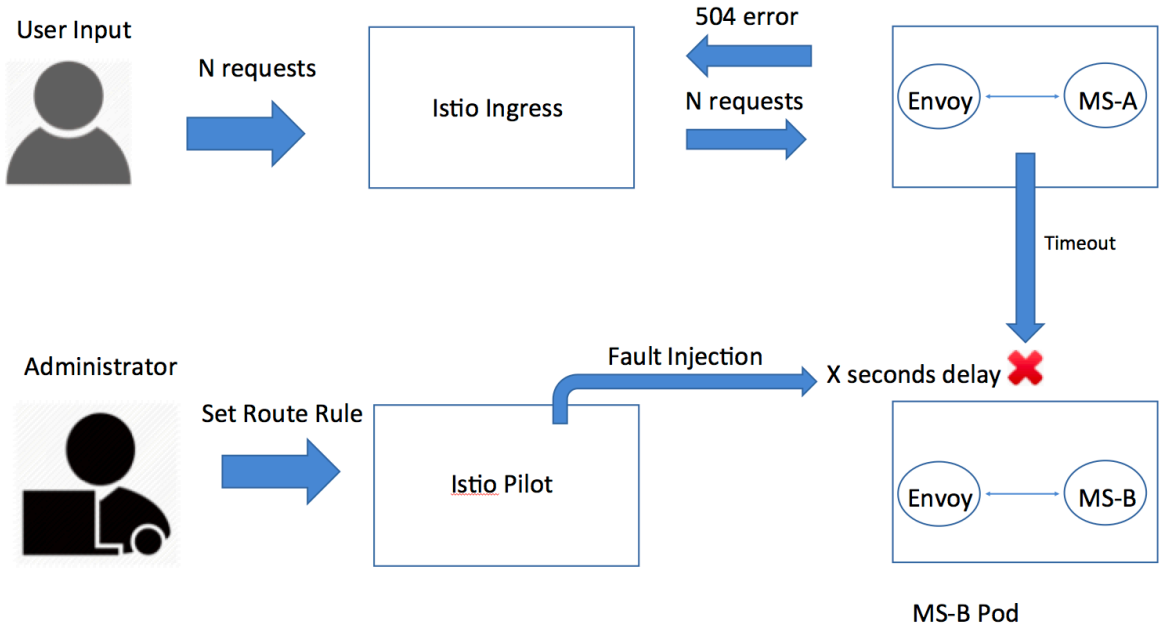
Traffic mirroring, also called shadowing, is a powerful concept that allows feature teams to bring changes to production with as little risk as possible.

Mirroring sends a copy of live traffic to a mirrored service. The mirrored traffic happens out of band of the critical request path for the primary service, in production ("Dark Production") or to test clusters.

The traffic is mirrored asynchronously and out of band from the production traffic. Any responses are ignored.



ISTIO TRAFFIC MANAGEMENT – FAULT INJECTION



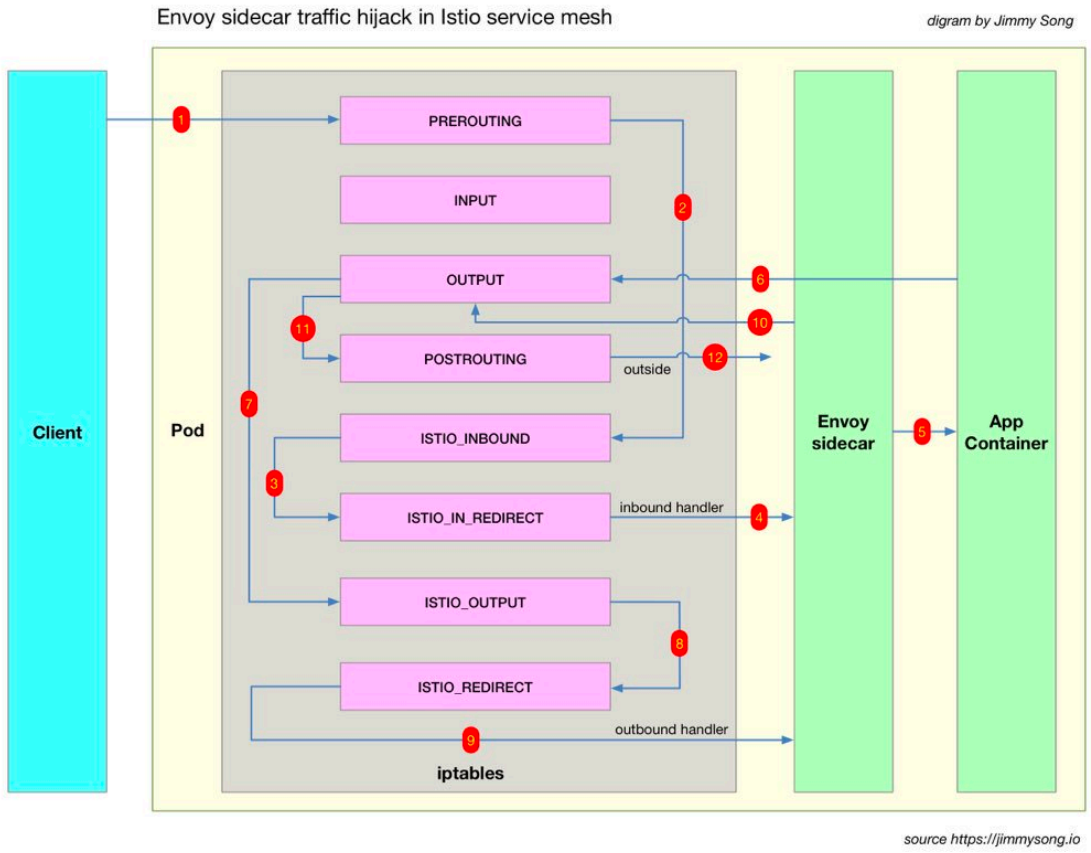
Fault injection is a system testing method which ***involves the deliberate introduction of faults and errors into a system.***

It can be used to identify design or configuration weaknesses and to ensure that the system is able the handle faults and recover from error conditions.

With Istio, failures can be injected at the application layer to test the resiliency of the services.

You can configure faults to be injected into requests that match specific conditions to simulate service failures and higher latency between services.

ISTIO PERFORMANCE AND SCALABILITY



The Istio load tests mesh consists of

- 1000 services
- 2000 sidecars
- 70,000 mesh-wide requests per second.

After running the tests using Istio 1.7.4, there are the results:

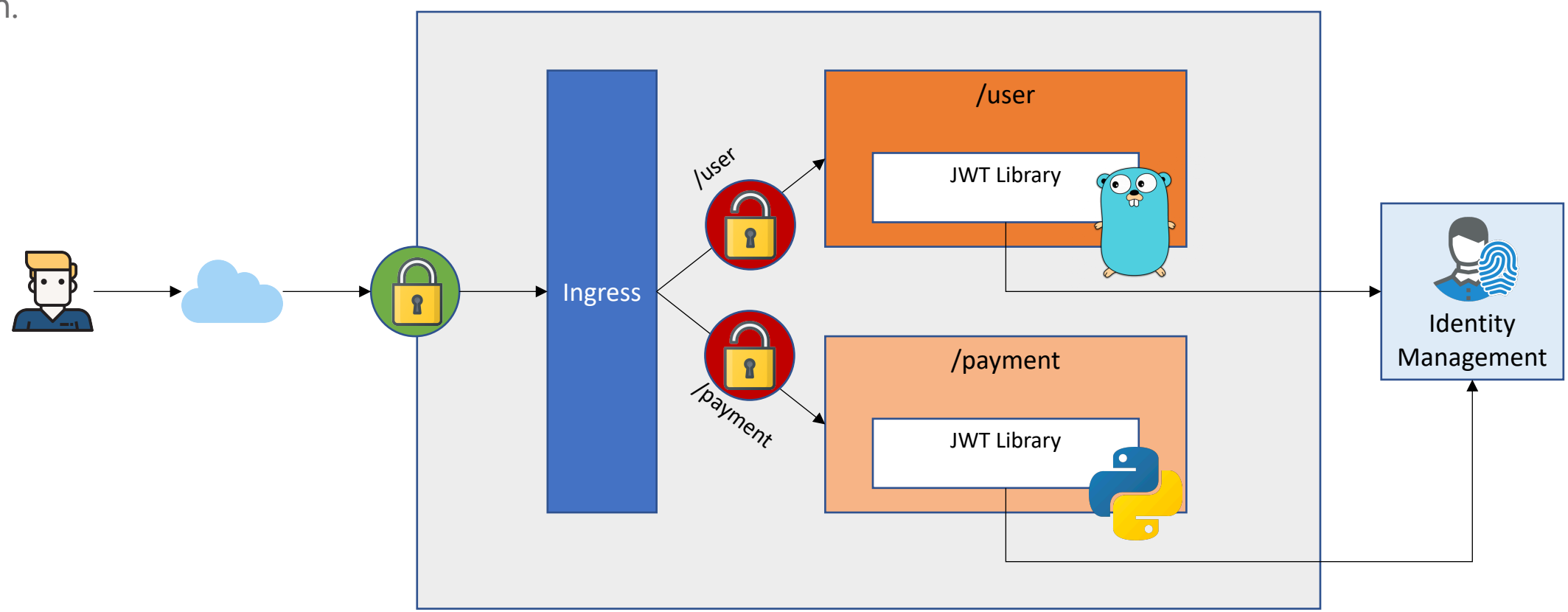
- The Envoy proxy uses **0.5 vCPU and 50 MB memory** per 1000 requests/second going through the proxy.
- Istiod uses **1 vCPU and 1.5 GB of memory**.
- **The Envoy proxy adds 2.76 ms to the 90th percentile latency**.

Inside the mesh, a request traverses the client-side proxy and then the server-side proxy. This two proxies on the data path add about 6.3 ms to the 90th percentile latency at 1000 requests per second. The server-side proxy alone adds 1.7 ms to the 90th percentile latency.

SERVICE MESH SIMPLIFIES SECURITY - AUTHENTICATION

OPTION 1 – JWT VALIDATION IN APPLICATIONS

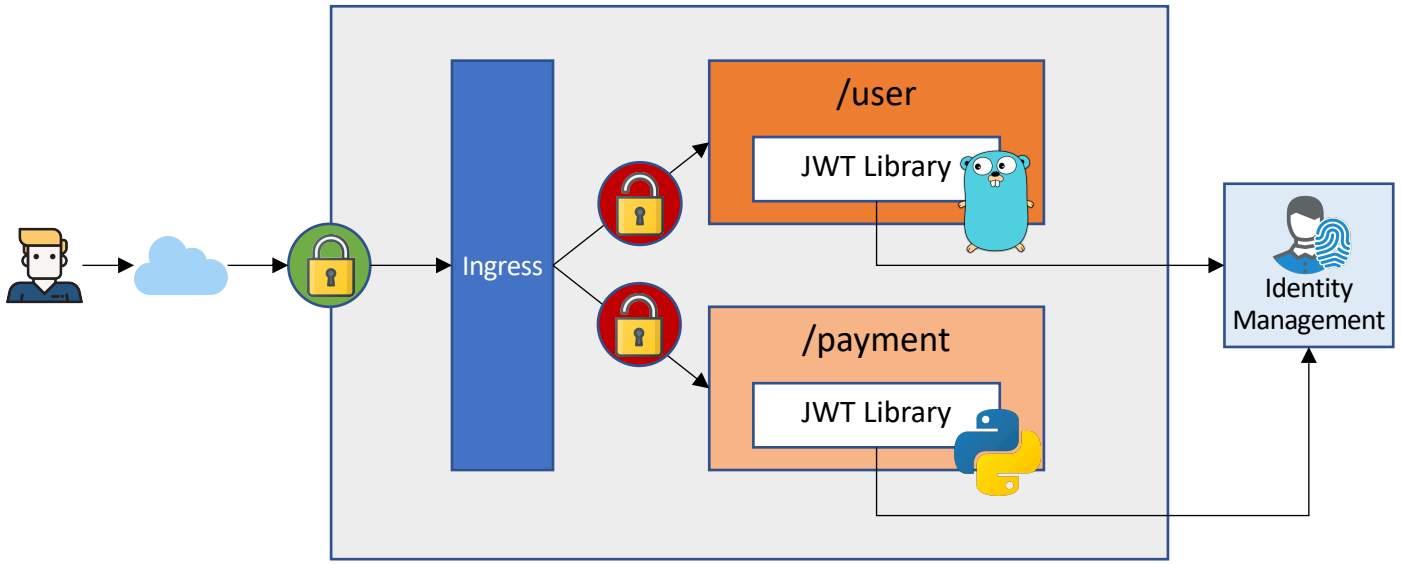
The application itself is validating the JWT token.



SERVICE MESH SIMPLIFIES SECURITY - AUTHENTICATION

OPTION 1 - THE PROCESS

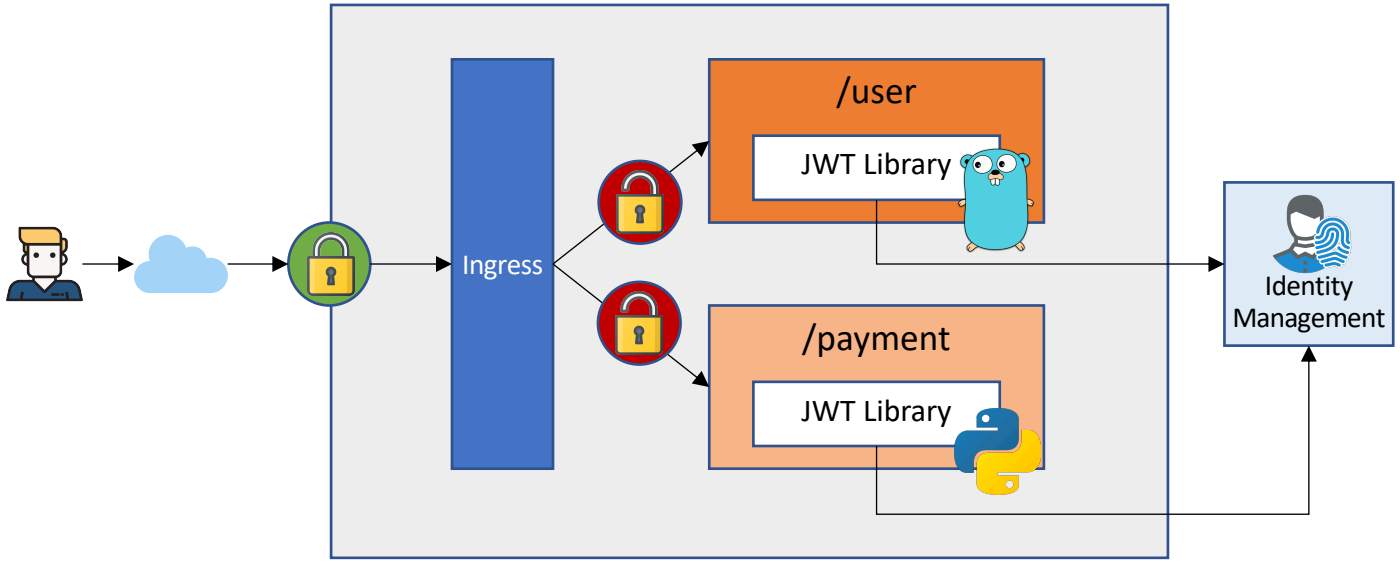
1. The end user is making a request which goes to the ingress and that request is encrypted and it has a JWT token in it as the authorization header
2. The ingress will do the TLS termination for requests
3. Ingress will forward the request to the services (i.e. based on paths)
4. The request which contains the JWT token is passed in plain text from ingress to the backend
5. The backend uses some library to validate the JWT token
6. If the request is valid, then the service processes the request; it not, it send authentication reject response to the client



SERVICE MESH SIMPLIFIES SECURITY - AUTHENTICATION

OPTION 1 - THE PROBLEM

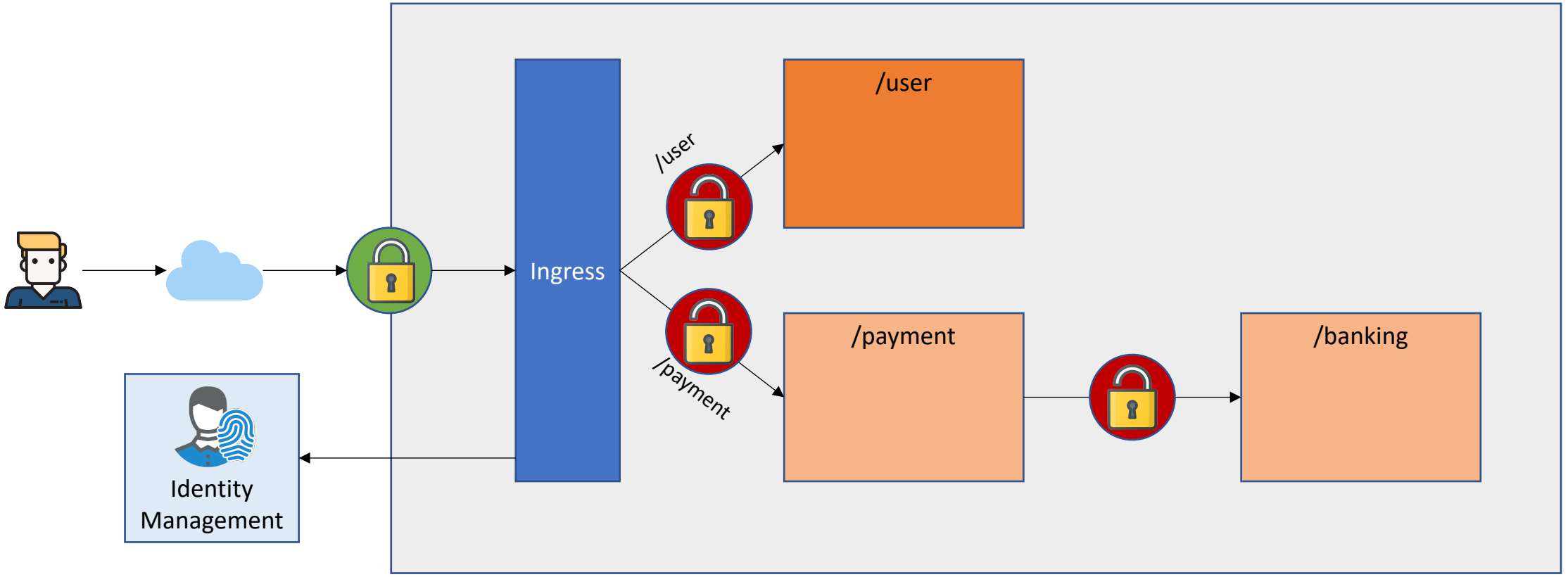
1. The link between the ingress and the backend is unencrypted – anyone that intercept the token can assume the identity of the end user
2. Application code to implement validation – if you have to change the Identity Management tool, all the libraries must be replaced



SERVICE MESH SIMPLIFIES SECURITY - AUTHENTICATION

OPTION 2 – JWT VALIDATION IN INGRESS

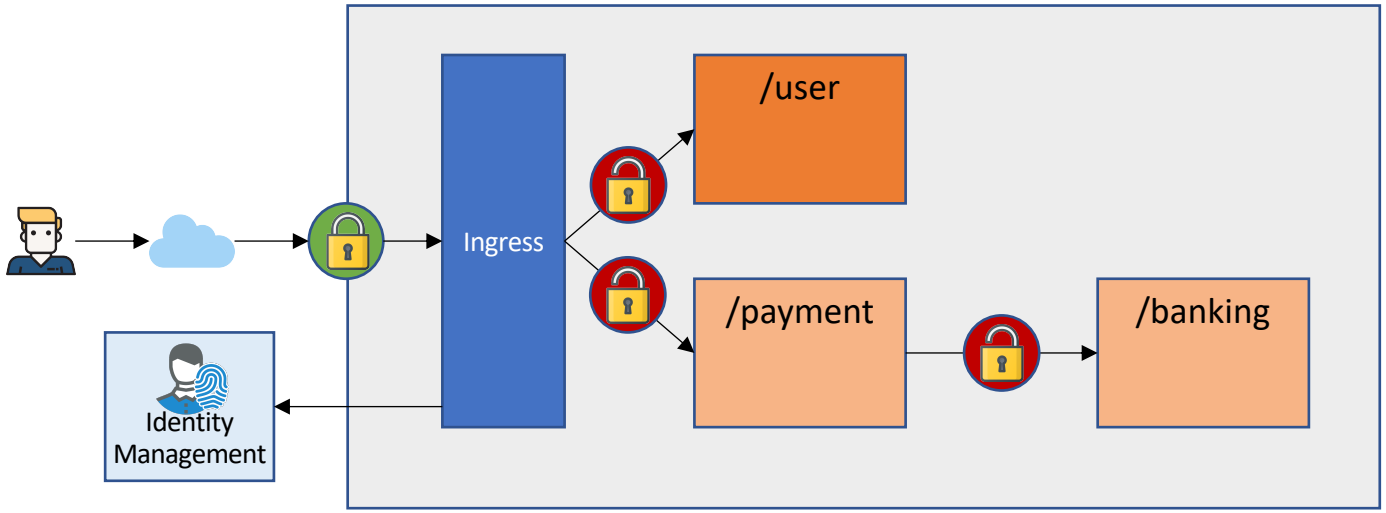
The ingress is responsible for communication with Identity Management solution.



SERVICE MESH SIMPLIFIES SECURITY - AUTHENTICATION

OPTION 2 - THE PROBLEM

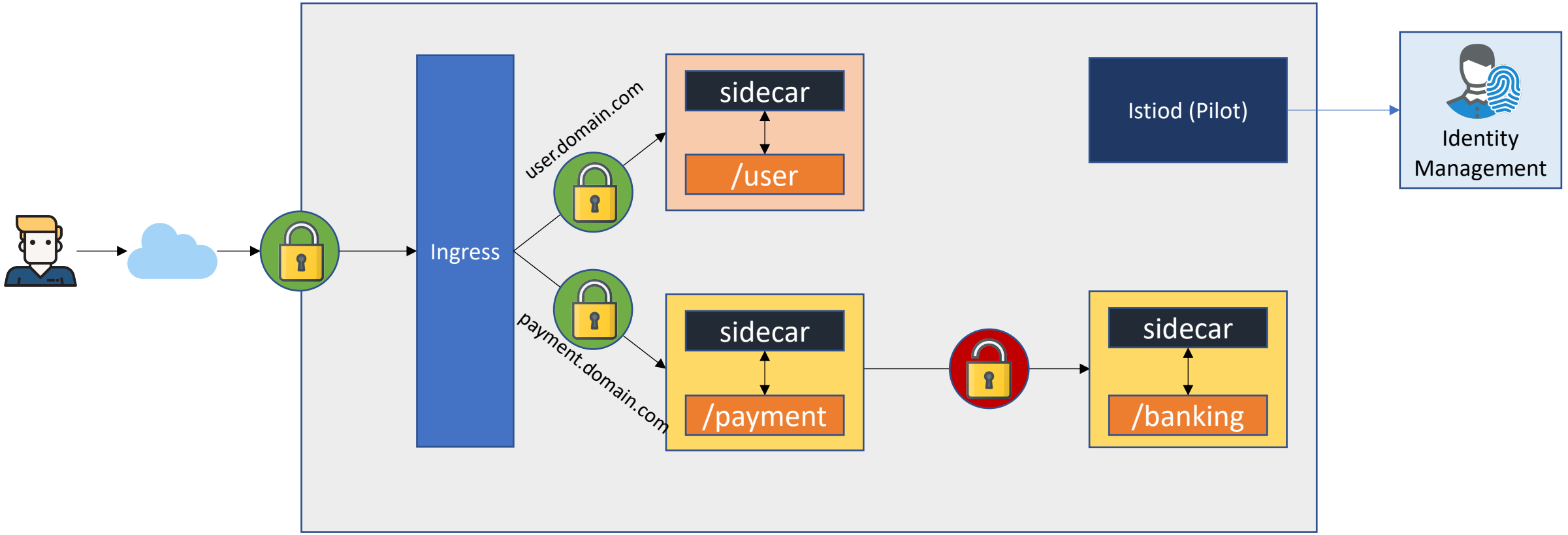
1. The link between the ingress and the backend is unencrypted – anyone that intercept the token can assume the identity of the end user
2. In a nested flow of calls – the payment service calling the banking service – there will be a propagation of the identity with no encryption between services



SERVICE MESH SIMPLIFIES SECURITY - AUTHENTICATION

OPTION 3 – ISTIO INGRESS TLS PASSTHROUGH + JWT VALIDATION BY SIDECARS

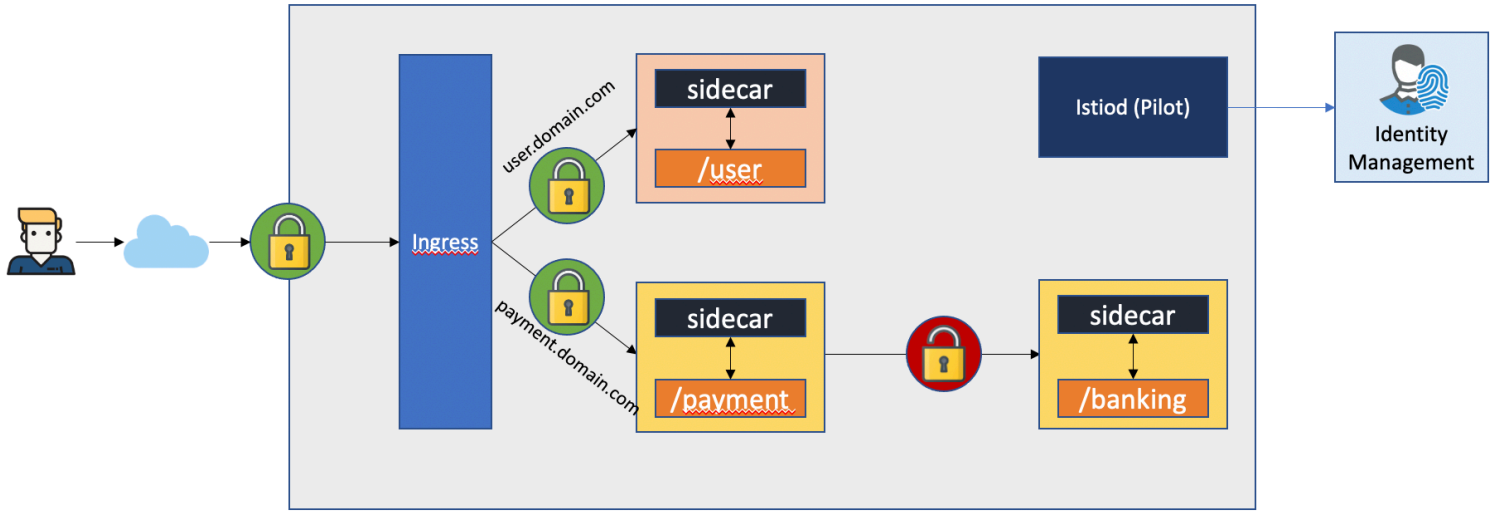
The ingress is responsible for TLS pass-trough.



SERVICE MESH SIMPLIFIES SECURITY - AUTHENTICATION

OPTION 3 – ISTIO INGRESS TLS PASSTHROUGH + JWT VALIDATION BY SIDECARS – THE PROCESS

1. The traffic comes encrypted to the ingress that via SNI (Server Name Indication) based routing
2. The sidecars are configured to do JWT validation – Istiod (Pilot) gets the configuration from the operator (JWT url, JWT attributes), it fetches the Public Key from Identity Management system and it configures the sidecars with the Public Key
3. When the sidecar receives the request, it uses the Public Key to validate the token.
4. If the token is valid, it will forward the entire token to the backend.



SERVICE MESH SIMPLIFIES SECURITY - AUTHENTICATION

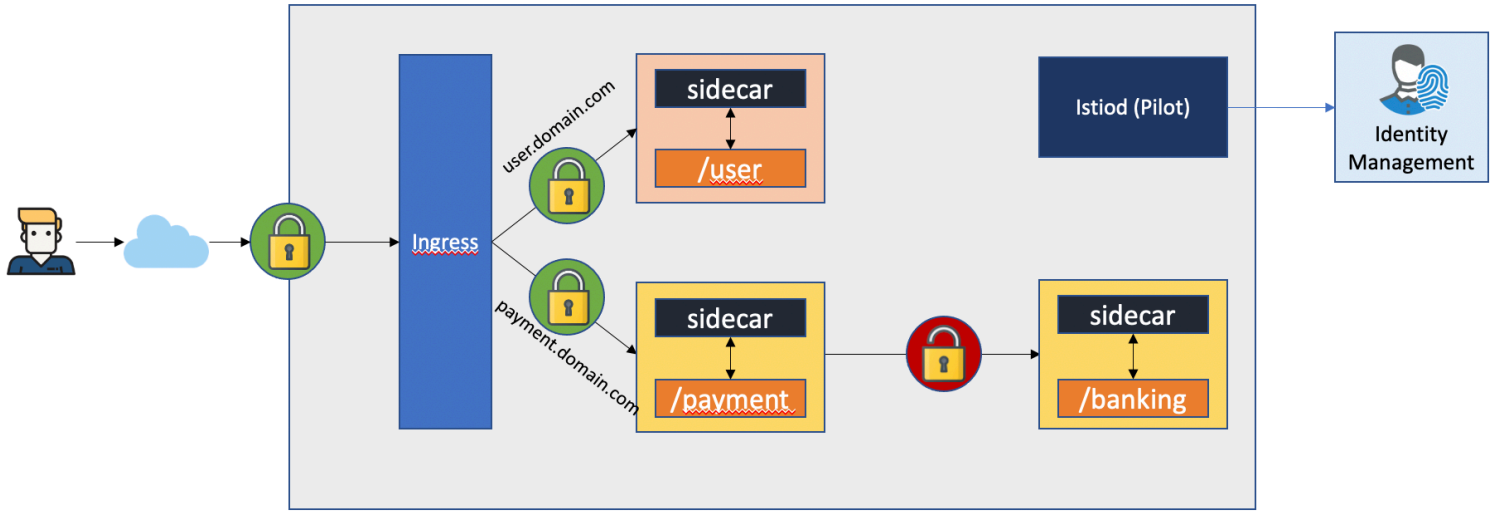
OPTION 3 – ISTIO INGRESS TLS PASSTHROUGH + JWT VALIDATION BY SIDECARS – THE PROCESS

THE PROS

- Backend only receives valid requests
- Encryption from ingress to backend

THE CONS

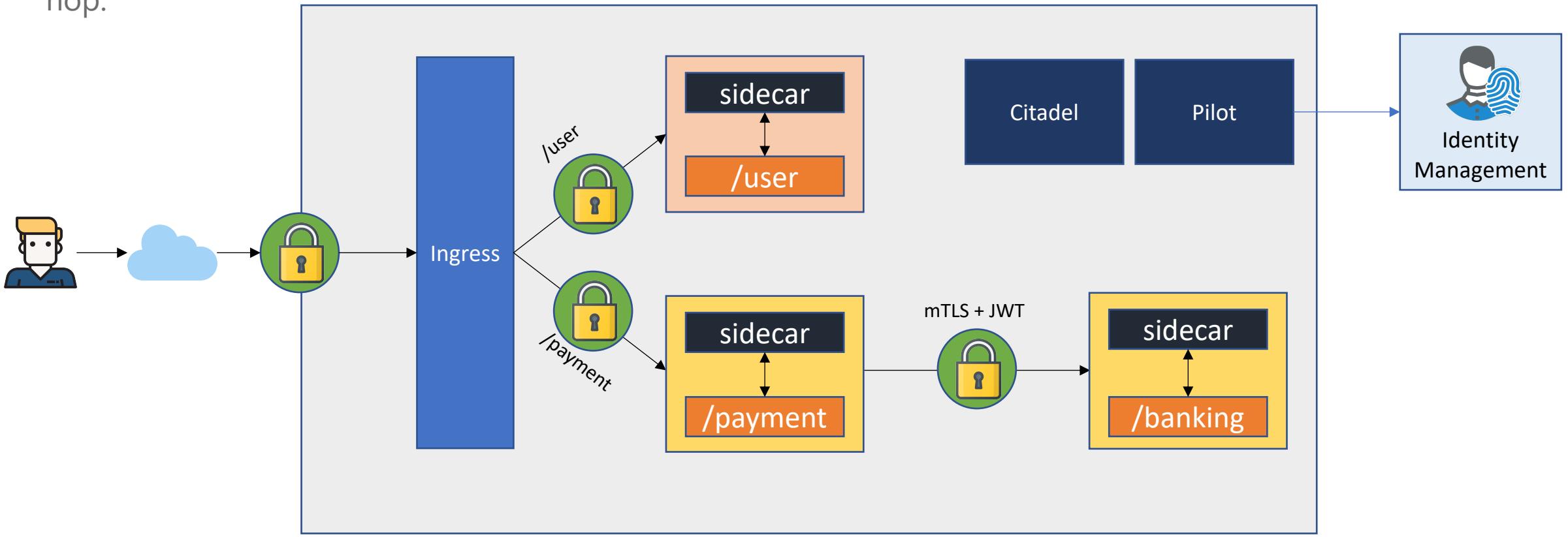
- With SNI routing, some of the advanced capability is lost
- Not attaching identities to each microservice: when the application propagate the JWT token, the communication is not encrypted
- By default Istio sidecars will propagate JWT token to one hop



SERVICE MESH SIMPLIFIES SECURITY - AUTHENTICATION

OPTION 4 – ISTIO mTLS + JWT VALIDATION BY SIDECARS

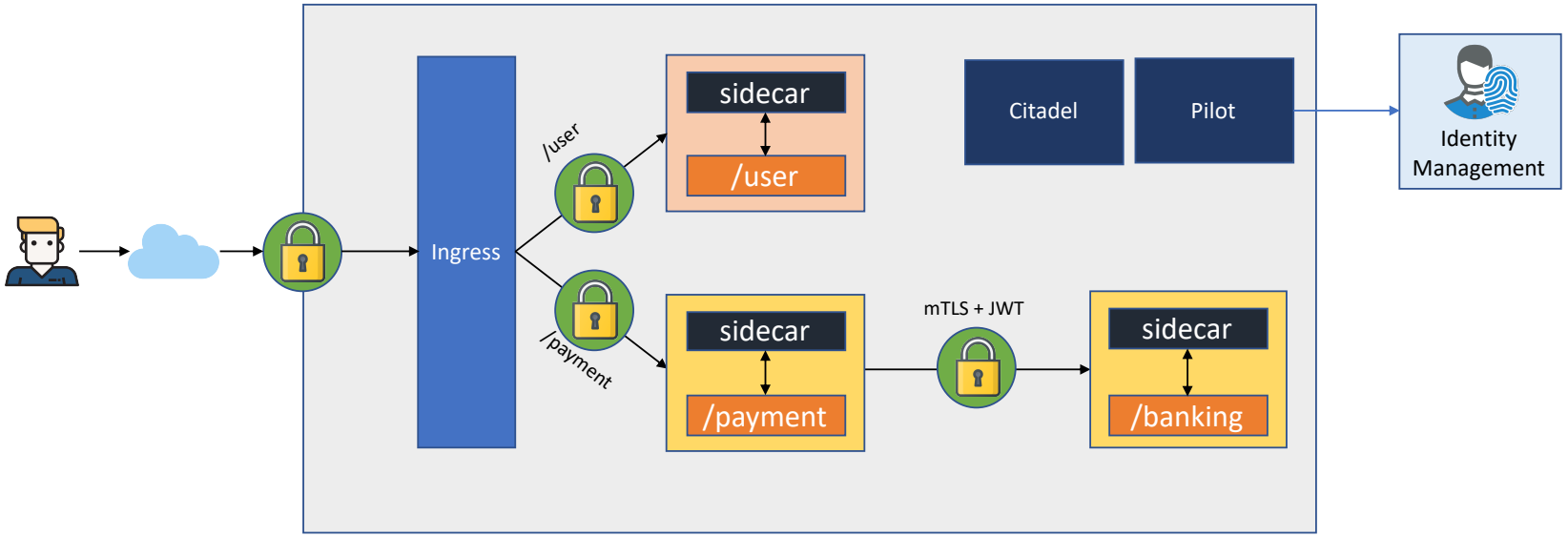
Istio has the capability to issue single unique identities for each service, securing traffic at each hop.



SERVICE MESH SIMPLIFIES SECURITY - AUTHENTICATION

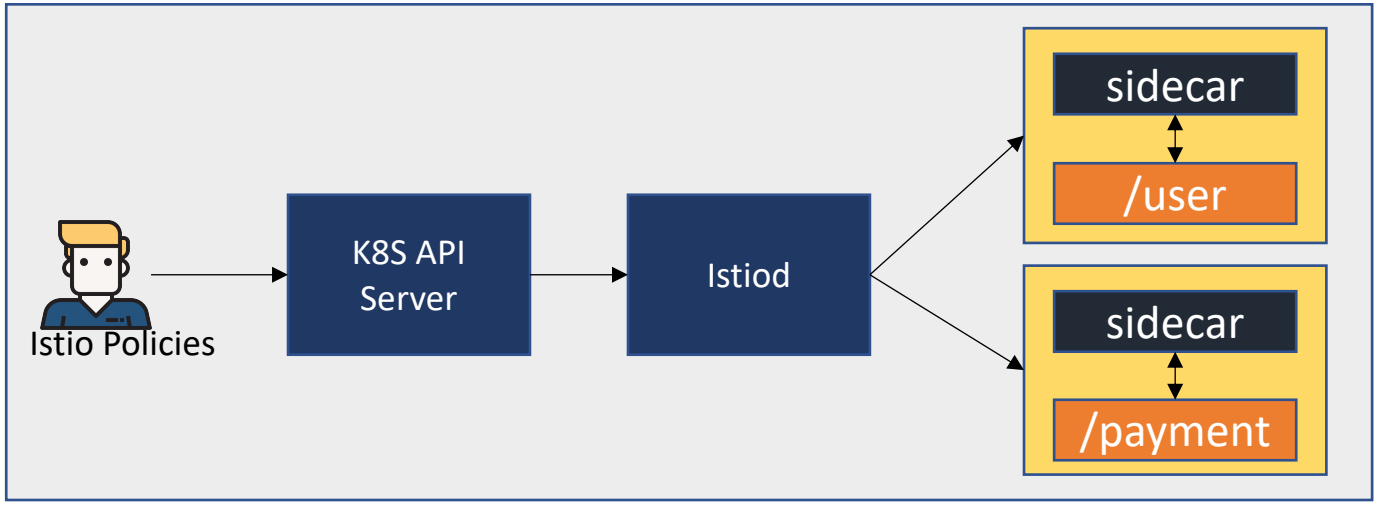
OPTION 4 – ISTIO mTLS + JWT VALIDATION BY SIDECARS

1. The user traffic is presented encrypted to the ingress
2. Ingress terminates the traffic and re-encrypts with mutual TLS with its own identity and the identity of the sidecar for the service
3. User service or payment service will do JWT validation.
4. Payment service communicates with banking service propagating the JWT
5. Citadel is responsible of creating certificates used for mTLS and rotating them

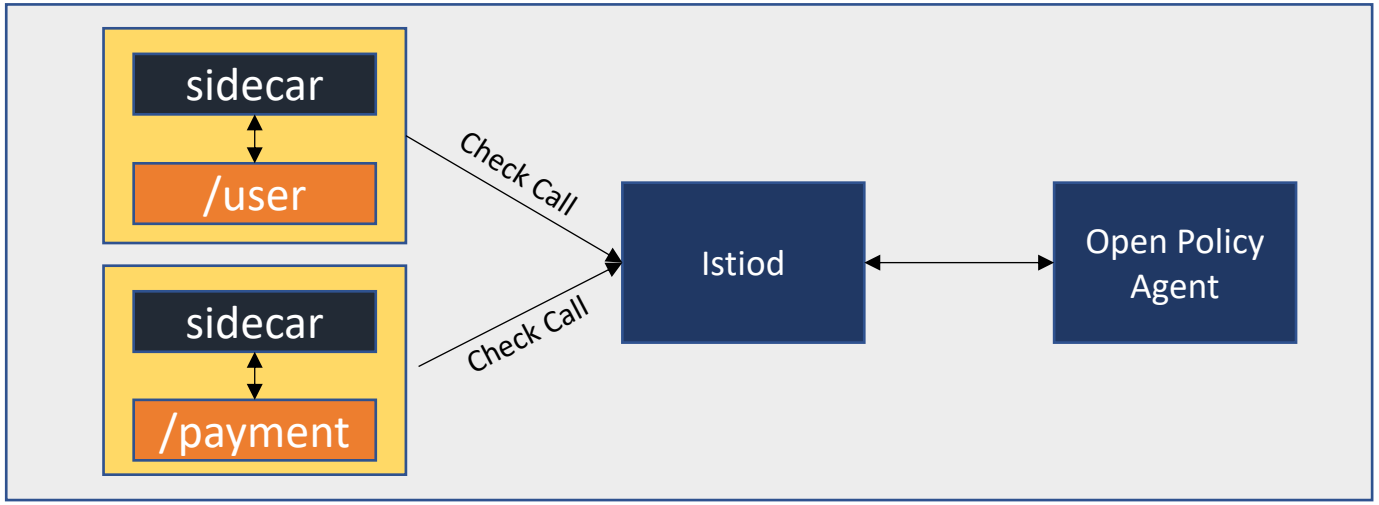


SERVICE MESH SIMPLIFIES SECURITY - AUTHORIZATION

ROLE BASED ACCESS CONTROL (RBAC)



ATTRIBUTES BASED ACCESS CONTROL (ABAC)



FOLLOW US

www.kiratech.it

Kiratech S.p.A.

@kiratech

Kiratech

kiratechspa

KiratechChannel