Zurich University
of Applied Sciences

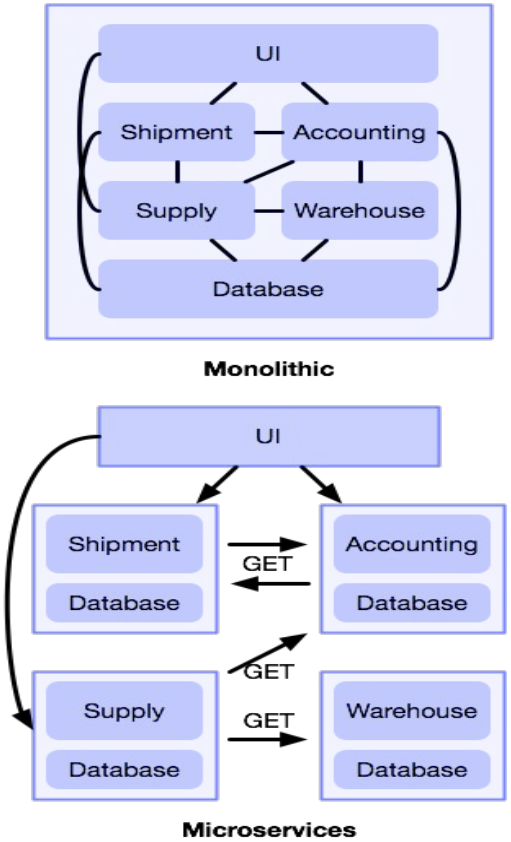
School of
Engineering
InIT Institute of Applied
Information Technology

MESH - Service Meshes

Prof. Dr. Thomas M. Bohnert
Christof Marti

Drawback of Microservice Architectures

- Implementing Cloud Native Applications using a distributed microservices architecture has advantages (see CNA1).
- But also some drawbacks
  - **Complexity is externalized** – overall system is more complex
  - Each component needs to implement a new set of **cross cutting concerns** (network communication, asynchronous requests, distributed state, security, …
    → large effort to implement seamless and polyglot
    → tendency to uniform frameworks (spring, .net, …)
  - Coordination of APIs, Protocol versions, migration, …
  - Additional **operations overhead** (monitoring, logging, debugging, certificate management, …)



Eight fallacies of distributed computing

(False) assumptions when writing distributed applications:

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology does not Change
6. There is one Administrator
7. Transport cost is zero
8. The network is homogenous

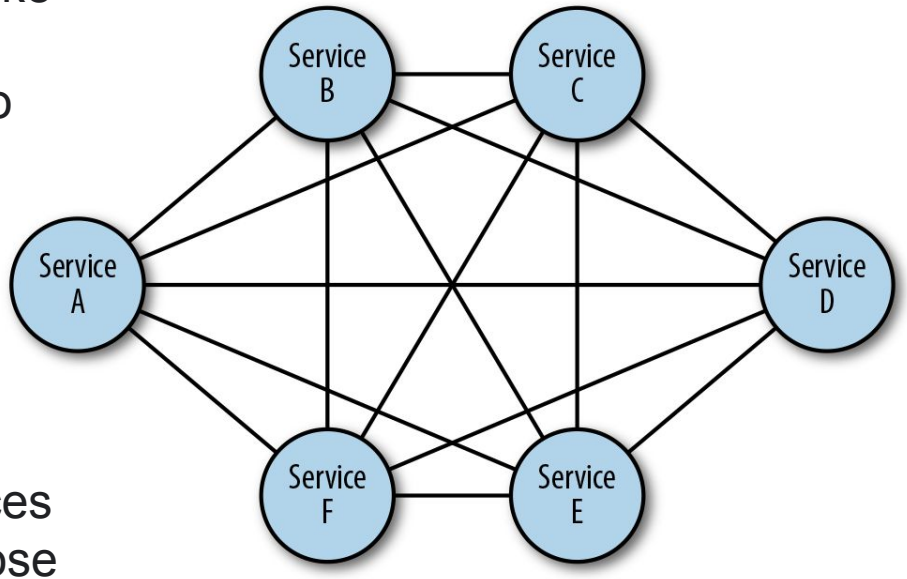
1994 L.Peter Deutsch, 1997 James Gosling, Sun Microsystems

Challenges of Microservice Architectures

- Network resilience / Traffic management
  - In distributed architectures, services can become bottlenecks or dependencies for other services.
  - Network policies managing quotas and rate limits for all services can ensure that a rogue service making too many calls does not overload the services it calls.
  - To effectively control services, create policies that specify which services can and can't make calls.
- Security
  - In a monolithic application, all function-to-function calls are secure inside the monolith.
  - Microservices need to authenticate, authorize, encrypt, and communicate.
  - Additional auditing tools needed to trace service-to-service communication.
- Observability
  - Observability is more important in microservice-based architectures. In monolithic applications, log files are sufficient to identify the source of an issue.
  - In a microservices architecture, multiple services can span a single request. Latency, errors, and failures can happen in any service within the architecture.
  - Developers need logging, network metrics, and distributed tracing and topology to investigate problems and pinpoint their location.

Exponential growth of complexity

- Simple to address cross cutting concerns like security, network resilience, policy, and observability for a few applications split into microservices,
- However, enterprises might have dozens, hundreds, or thousands of microservices.
- Therefore, any solution must scale.
  If not done correctly, the complexity of applications and the amount of microservices creates a greater dependence between those services.

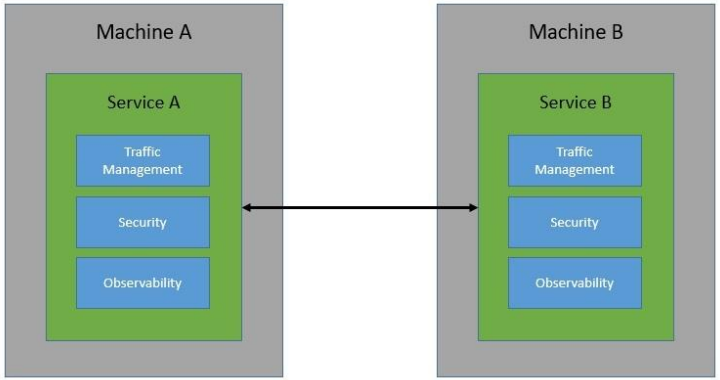


Source: O'Reilly service mesh fundamentals
https://www.oreilly.com/library/view/the-enterprise-path/9781492041795/ch01.html

Options to implement CNA cross cutting concerns

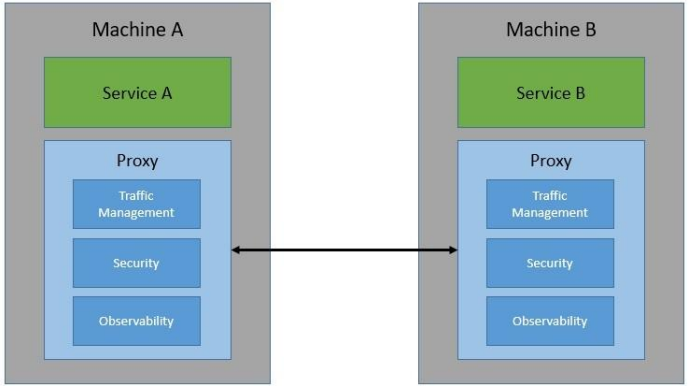
Each Service is implementing the CCC

- Using Libraries and dedicated services
  - e.g. Netflix (Eureka Service Registry, Ribbon LB, Hystrix Circuit-Breaker, …)
  - AAA*) frameworks
  - Tracing instrumentation (Zipkin, Jaeger, Datadog, dynatrace, …
- Limited interoperability between languages / platforms, → uniform environments (spring, .net, …)



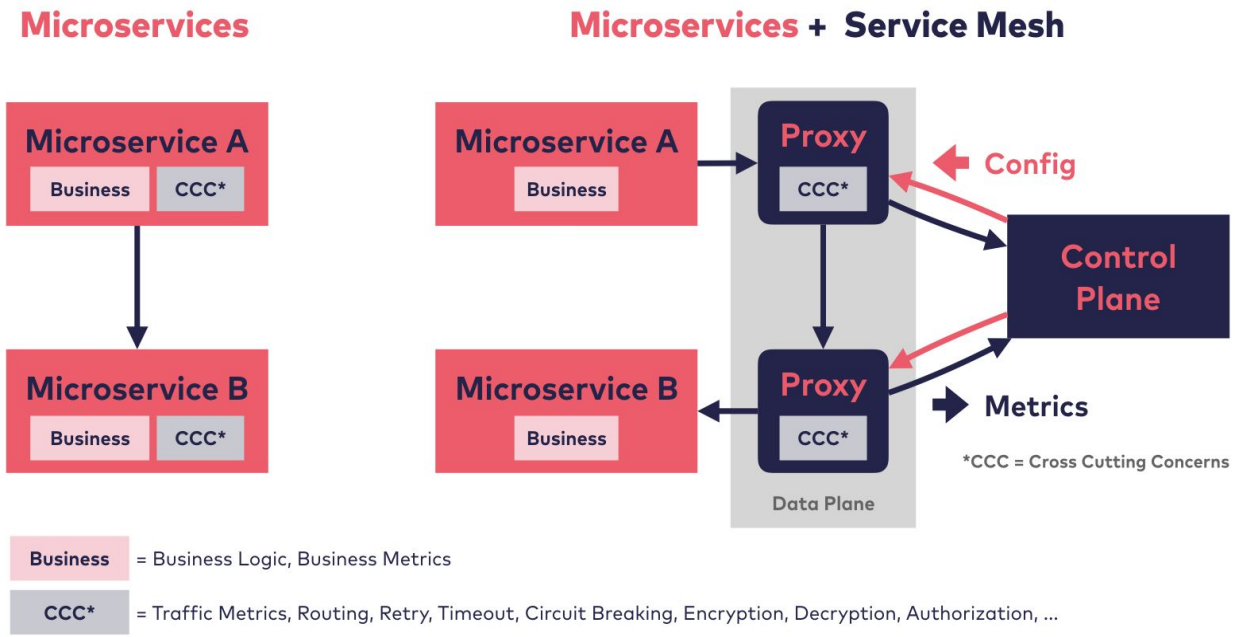
Outsourcing of CCC to proxy(ies)

- one or multiple separate proxy process(es)
- using API or intercepting network communication
- Language independent
- Service focuses on business functionality



*) Authentication, Authorization, Auditing

Service Mesh



Without a service mesh

- each microservice implements business logic and cross cutting concerns (CCC) by itself.

With a service mesh

- Many CCCs like traffic metrics, routing, and encryption are moved out of the microservice and into a **proxy**.
- Business logic and business metrics stay in the microservices.
- Incoming and outgoing requests are transparently routed through the proxies, which build the **data plane**.
- a service mesh adds a so-called **control plane**. It distributes configuration updates to all proxies and receives metrics collected by the proxies for further processing

Service Mesh

A service mesh is a platform layer on top of the infrastructure layer that enables managed, observable, and secure communication between individual services.

- Service meshes factor out all the *cross cutting concerns* of running a service, like monitoring, networking, and security.
- Service developers and operators can focus on *creating and managing applications* instead of worrying about implementing measures to address challenges for every service.
- Service meshes are transparent to the application. It monitors all traffic through a proxy. The **proxy** is deployed by a *sidecar pattern* to the microservices.
- This pattern decouples application or business logic from network functions.

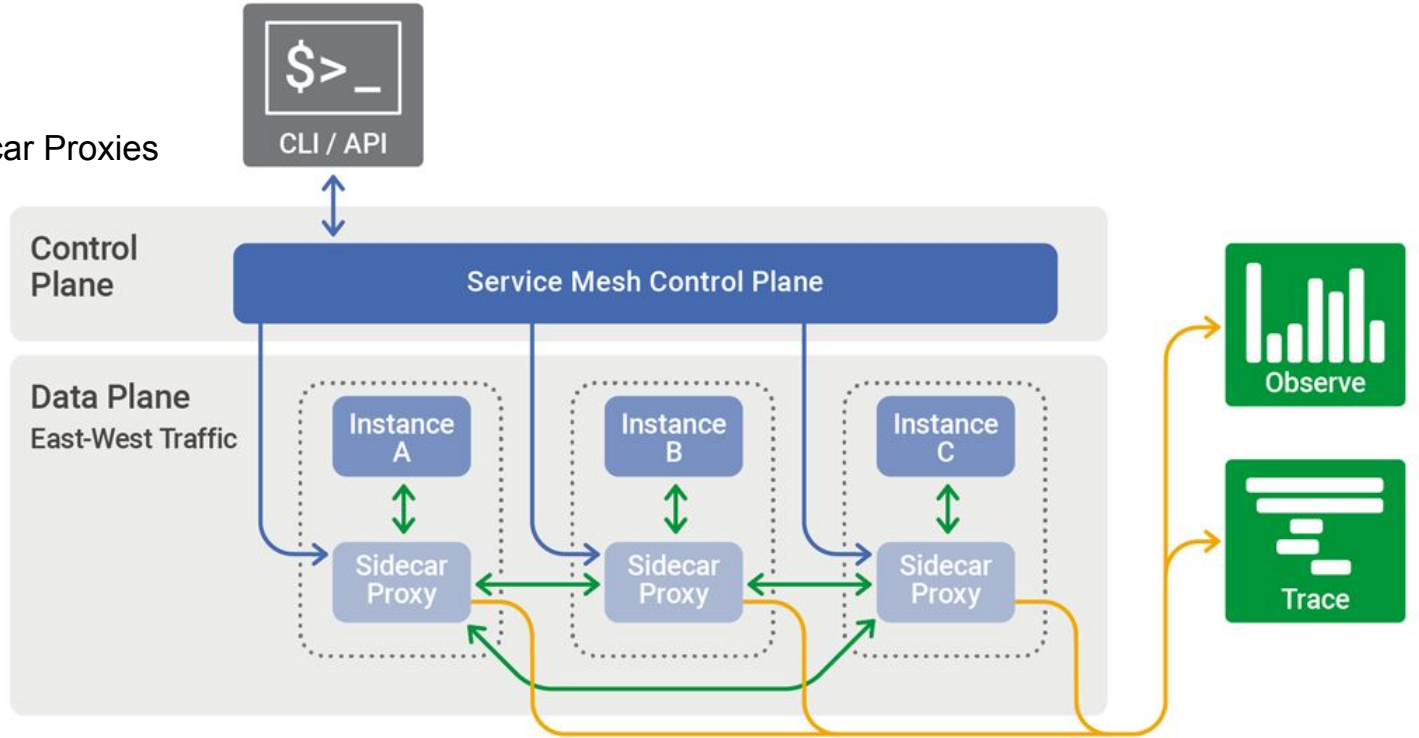
Data-Plane & Control Plane

Control Plane:

- Manages Policies,
- Injects and controls Sidecar Proxies
- Collects Metrics and Logs

Data Plane:

- Sidecar Proxies
- Intercept traffic from Service-Container
- Handles traffic between Services
- Executes Policies from Control Plane
- Collects Service metrics and forwards them to the control plane



Data-Plane & Control Plane

Data-Plane

In a service mesh, deployments are modified to include a *sidecar proxy*. Instead of calling services directly over the network, each service calls its local sidecar proxy, which in turn encapsulates the complexities of the service-to-service exchange. This interconnected set of proxies (or sidecars) in a service mesh represents its data plane.

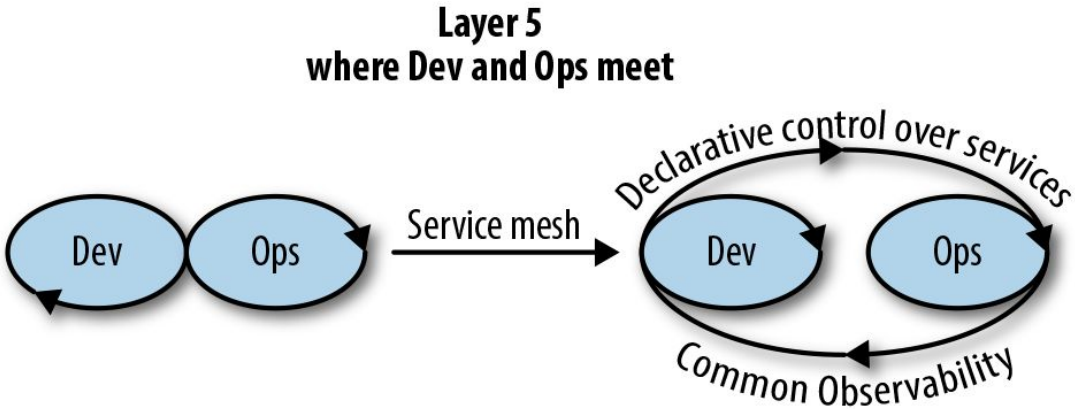
Control-Plane

All service mesh proxies are centrally managed by a control pane. This is quite useful when supporting *service mesh policies* such as access control, observability, service discovery etc.

Dev / Ops decoupling

- Service meshes help you to avoid bloated service code, based on infrastructure concerns.
- Service meshes insert a dedicated infrastructure layer between dev and ops, separating cross cutting concerns of service communication by providing independent control over them.
- The service mesh is a networking model that sits at a layer of abstraction above TCP/IP.

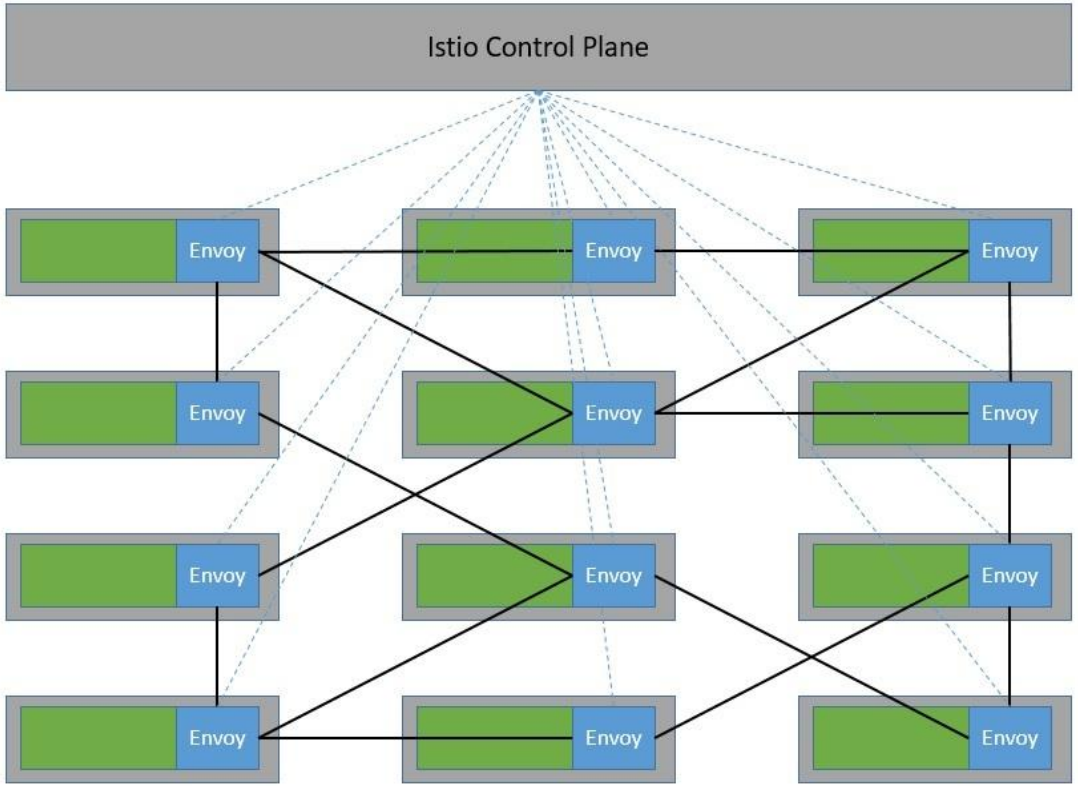
- Operators don't necessarily need to involve Developers to change how many times a service should retry before timing out.
- Customer Success teams can handle the revocation of client access without involving Operators.
- Product Owners can use quota management to enforce price plan limitations for quantity-based consumption of particular services.
- Developers can redirect their internal stakeholders to a canary with beta functionality without involving Operators.



Source: O'Reilly service mesh fundamentals
https://www.oreilly.com/library/view/the-enterprise-path/9781492041795/ch01.html

Scalability for large deployments

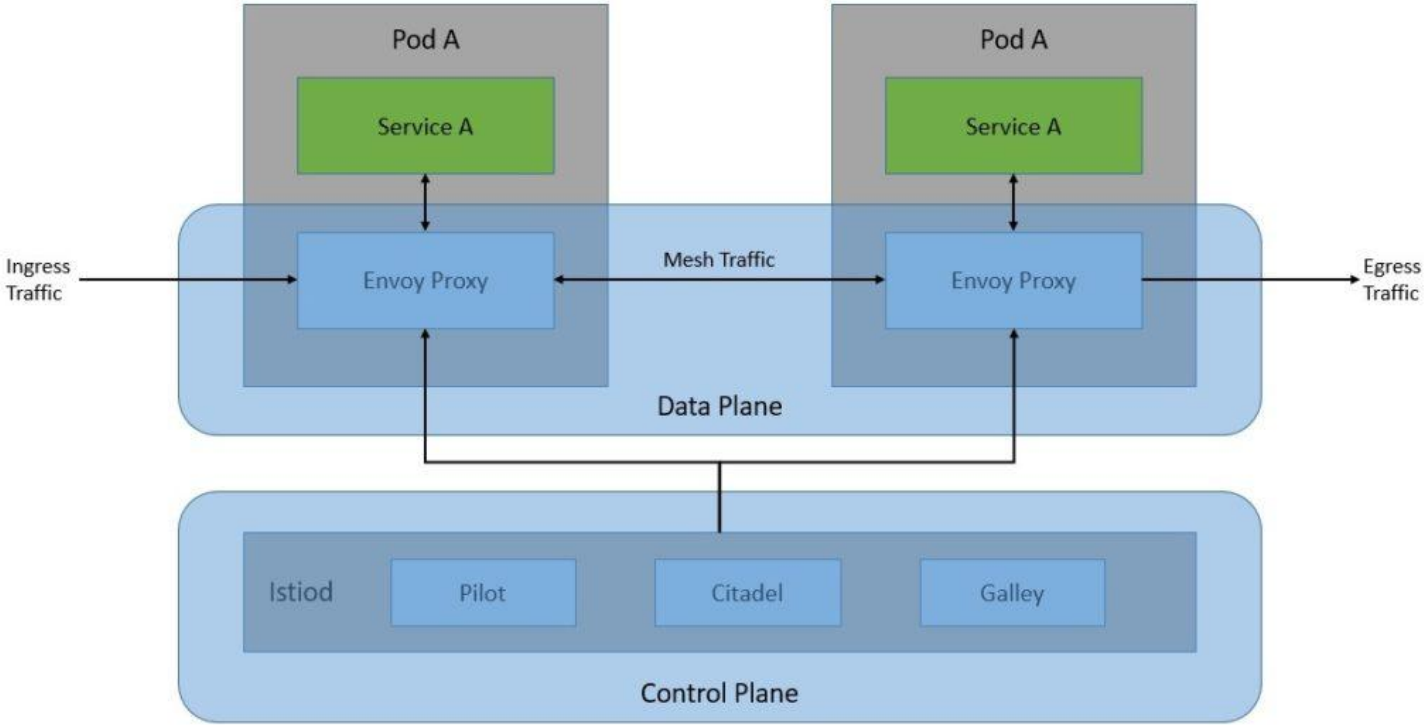
- Single Installation of control plane managed by platform operator.
- Policies can be applied
  - globally
    - network policies
    - security policies
  - tenant / namespace / application specific
    - Failure handling
    - Traffic routing
    - Metrics



Istio

- Istio is an open-source implementation of the service mesh originally developed by IBM, Google, and Lyft.
- It can layer transparently onto a distributed application and provide all the benefits of a service mesh like traffic management, security, and observability.
- It's designed to work with a variety of deployments, like on-premise, cloud-hosted, in Kubernetes containers, and in servicers running on virtual machines.
- Although Istio is platform-neutral, it's quite often used together with microservices deployed on the Kubernetes platform.

Istio components: Control Plane & Data Plane



Istio

Istio Data Plane – Envoy Proxy side-car

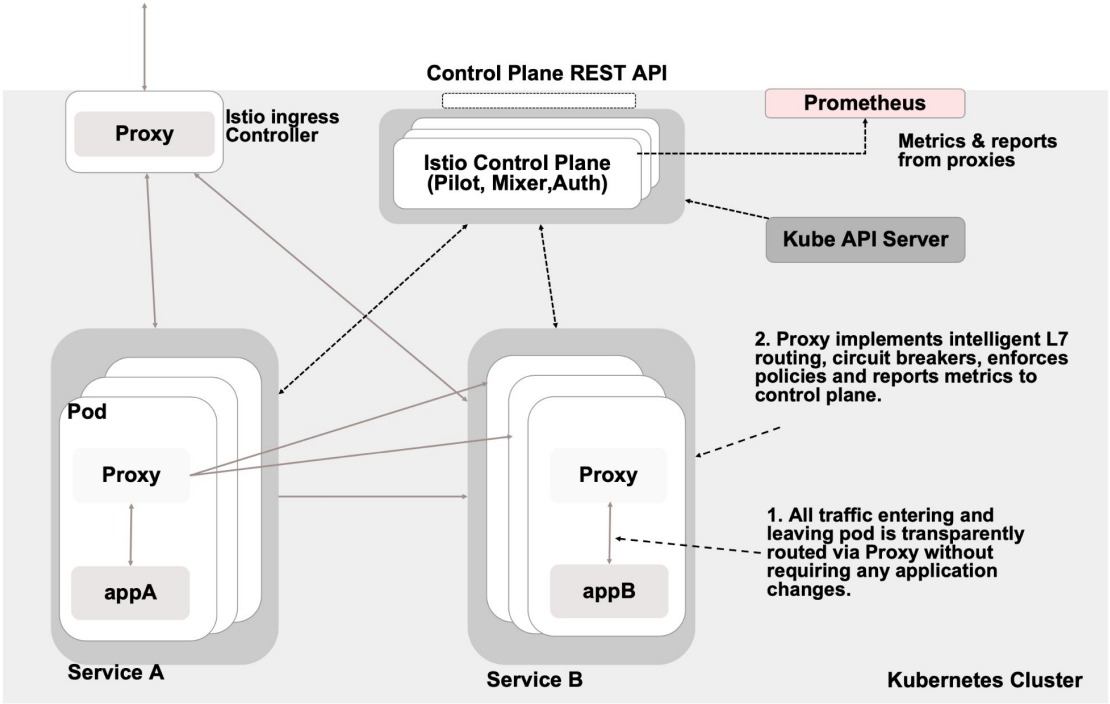
- High performanc proxy written in C++
- Injected as sidecar container in Kubernetes Pods
- Providing dataplane features
  - Dynamic Service Discovery
  - Communication
    - TCP, UDP, HTTP, DNS, L3&L4 filter, connection limiting, gRPC
    - HTTP routing, filter, dynamic proxy
    - Rate / Bandwith limiting,
  - Security
    - TLS
    - Authentication, Authorization, RBAC
  - Observability
    - Statistic, Access logging
    - Tracing
  - Patterns
    - Service Discovery
    - Load Balancing
    - Circuit Breaking
    - Health Checking
    - Connection Pooling

https://www.envoyproxy.io/

Istio Control Plane – Components

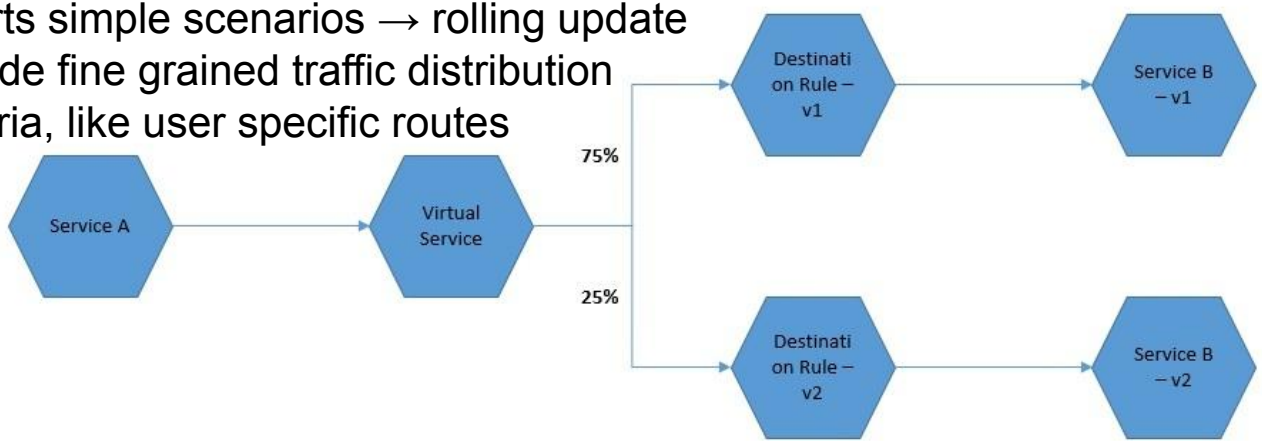
All functionality is provided by a single daemon (**Istiod**)
In previous releases the functionality was split into multiple services:

- *Pilot*: Service Discovery (managing Envoy sidecars),
  Managing Routing and Resilience Rules
  Traffic Management (Routing, Ingress, A/B Tests, Canary Rollouts)
  Resilience (Timeout, Retry, Circuit Breaker)
  Platform adapter (K8S, Mesos, …)
- *Citadel*: Security, Service-to-Service / end-user authentication, credential / certificate-management
- *Galley*: configuration management, injection

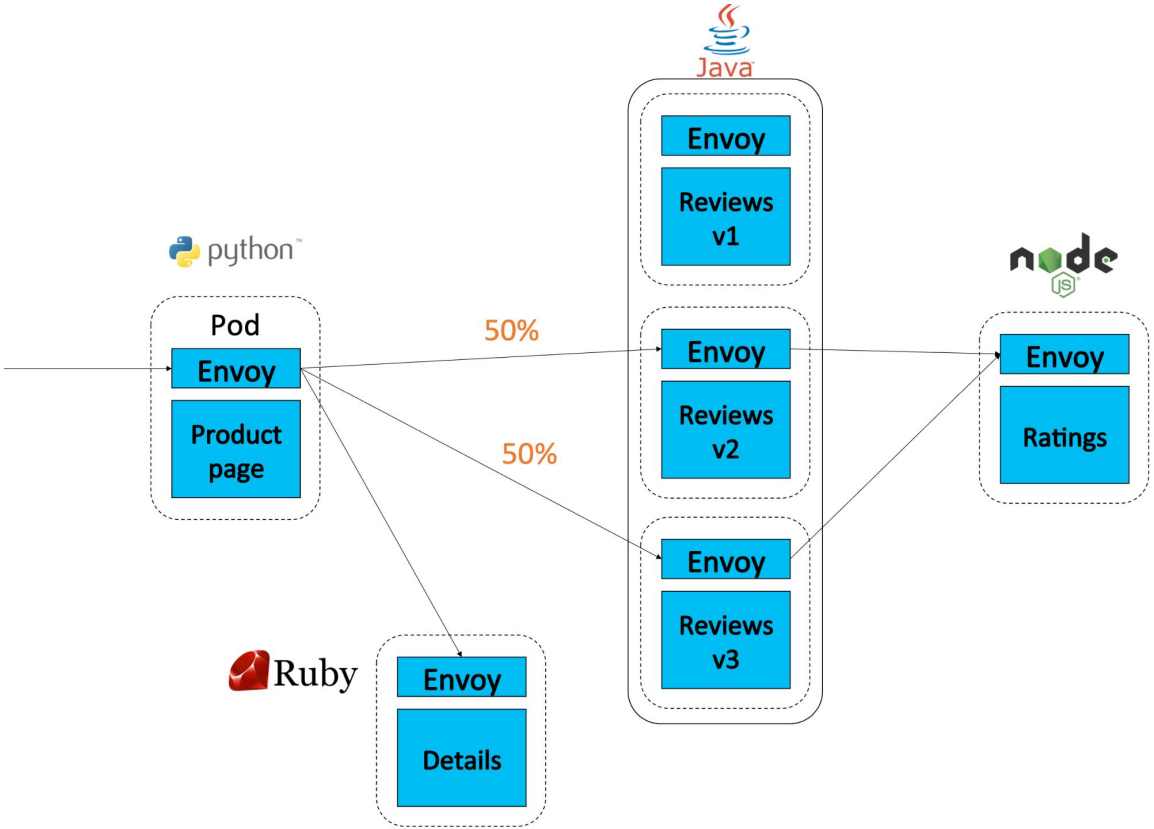

Control Plane REST API
Proxy
Istio ingress Controller
Istio Control Plane (Pilot, Mixer,Auth)
Prometheus
Metrics & reports from proxies
Kube API Server
2. Proxy implements intelligent L7 routing, circuit breakers, enforces policies and reports metrics to control plane.
Pod
Proxy
appA
Service A
Proxy
appB
Service B
1. All traffic entering and leaving pod is transparently routed via Proxy without requiring any application changes.
Kubernetes Cluster
Proxy. Based on Envoy, a high performance L7 proxy from Lyft, currently being used at large scale in production. https://github.com/lyft/envoy
Istio control plane traffic. Request routing rules, resilience configuration (circuit breakers, timeouts, retries), policies (ACLs, rate limits, auth), and metrics/reports from proxies.
User/application traffic. HTTP/1.1, HTTP/2, gRPC, TCP with or without TLS

Traffic Management

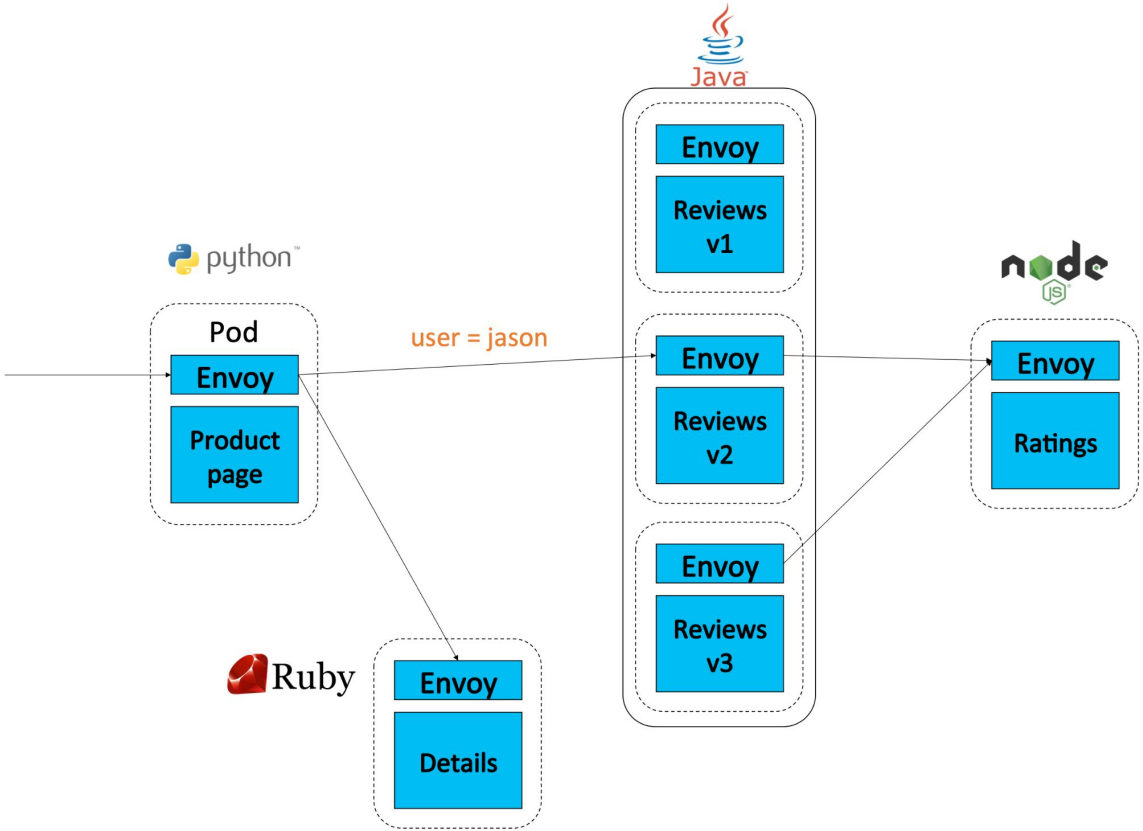
- Service Discovery
- Traffic routing
  - Ingress gateway (dynamic reverse proxy) & egress gateway
  - Basic Load Balancing (Round Robin, Random, Weighted least request)
  - AB-Testing (weighted routing, user specific)
  - Canary rollouts
    - Kubernetes supports simple scenarios → rolling update
    - service mesh provide fine grained traffic distribution and use other criteria, like user specific routes



Traffic Management - Weighted routing example



Traffic Management - User based routing example



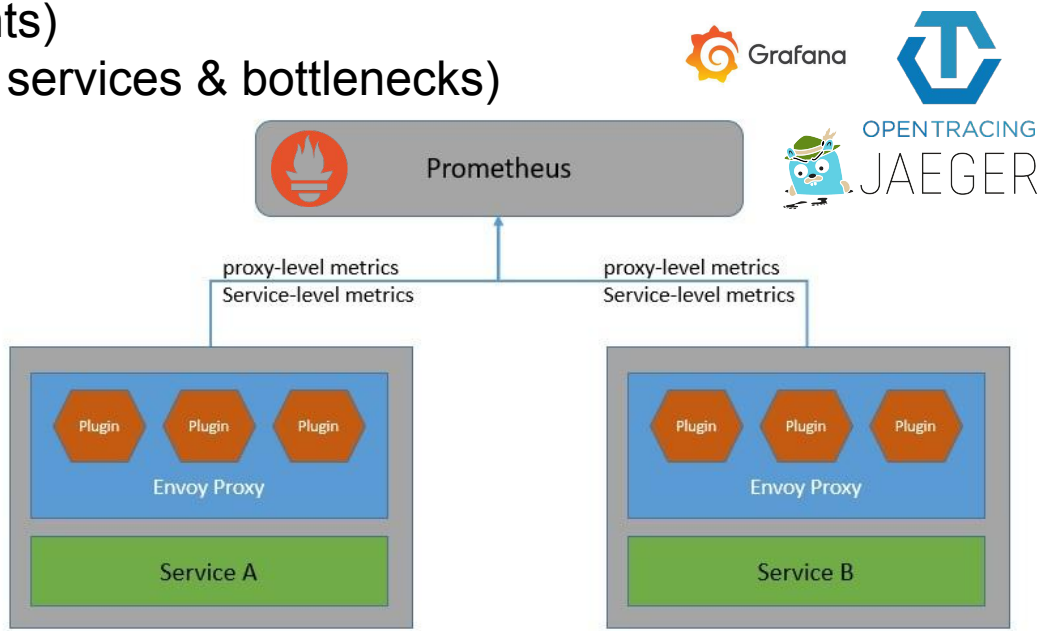
Traffic Management

- Failure Handling / Recovery
  - Timeouts
  - Rate / Bandwidth limiting
  - Circuit Breaker
  - Retry
  - Health Checks
- Fault injections
  test resilience, recovery capabilities of services without modifying or shutting down pods
  - delay (increased network latency, service overload)
  - failures (HTTP error codes, TCP connection failure)

Observability

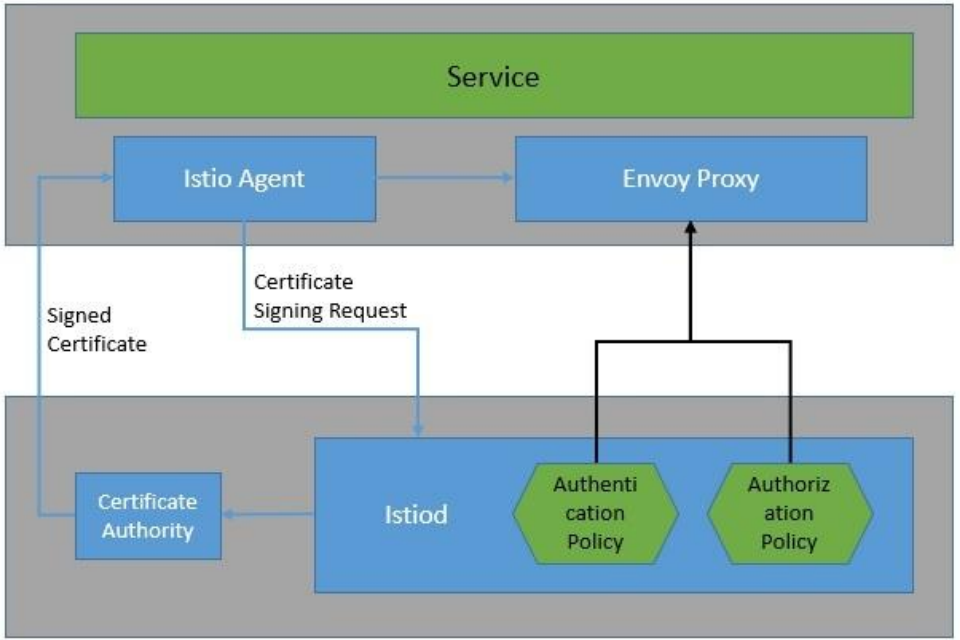
Istio generates detailed telemetry

- Metrics collection (latency, throughput, …) proxy level, service-level, control-plane
- Monitoring (service health, events)
- Distributed Tracing (detect slow services & bottlenecks)
  Integration of multiple backends
- Access logs



Security

- End-to-End Encryption
  - TLS
  - Certificate management (rollover)
- Authentication, Authorization
  - peer authentication (service-to-service)
  - request authentication (end-user)
- Access control (RBAC)

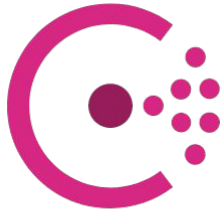


Service Mesh Implementations

- Many implementations
- Similar concepts, different components & technologies

How to choose a Service Mesh Implementation

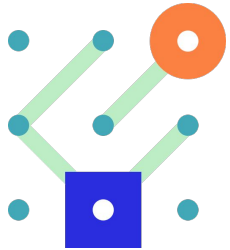
- Identify the most important problems to be solved by the service mesh. Keep in mind that libraries or adaptions to the architecture could be good alternatives in some cases (see below).
- Discuss requirements regarding simplicity/usability, performance, and compatibility.
- Identify top two or three implementations based on your feature and non-feature requirements. See table in comparision*)
- Try the implementations by executing their respective tutorials (links below) and possibly discard one candidate.
- Test the latency and resource overhead for your individual application. For each service mesh candidates, set up an identical test environment and install the service mesh. Set up an additional mesh-free environment. Install your application in all environments. Perform a load test in all and measure request latency, CPU and memory consumption by using tools like Locust or Fortio.



NGINX Service Mesh

*) Comparison: https://servicemesh.es/

SMI – Service Mesh Interface



- A standard interface for service meshes on Kubernetes
- Provider Agnostic
  - common portable set of service mesh APIs without binding to a specific implementation
- A basic feature set for the most common service mesh capabilities
  - Traffic policy – apply policies like identity and transport encryption across services
  - Traffic telemetry – capture key metrics like error rate and latency between services
  - Traffic management – shift traffic between different services
- Flexibility to support new service mesh capabilities over time
- SNCF sandbox project (https://smi-spec.io/)

Pros and Cons of Service Meshes

Pros

- Commodity features are implemented outside microservice code and they are reusable.
- Solves most of the problems in Microservices architecture which we used to have ad-hoc solutions: Distributed tracing, logging, security, access control etc.
- More freedom when it comes to selecting a microservices implementation language: You don’t need to worry about whether a given language supports or has libraries to build network application functions.

Pros and Cons of Service Meshes

Cons

- *Complexity*: Having a service mesh drastically increase the number of runtime instances that you have in a given microservice implementation.
- *Adding extra hops*: Each service call has to go through an extra hop (through service mesh sidecar proxy).
- *Service Meshes address a subset of problems*: Service mesh only address a subset of inter-service communication problems, but there are a lot of complex problems such as complex routing, transformation/type mapping, integrating with other services and systems, to be solved at your microservice's business logic.
- *Immature*: Service mesh technologies are relatively new to be declared as full production ready for large-scale deployments.