What is a Service Mesh?

And Do I Need One When Developing Cloud Native Systems?

Daniel Bryant

@danielbryantuk

SpectoLabs

Cloud Native Apps: Expectations versus reality

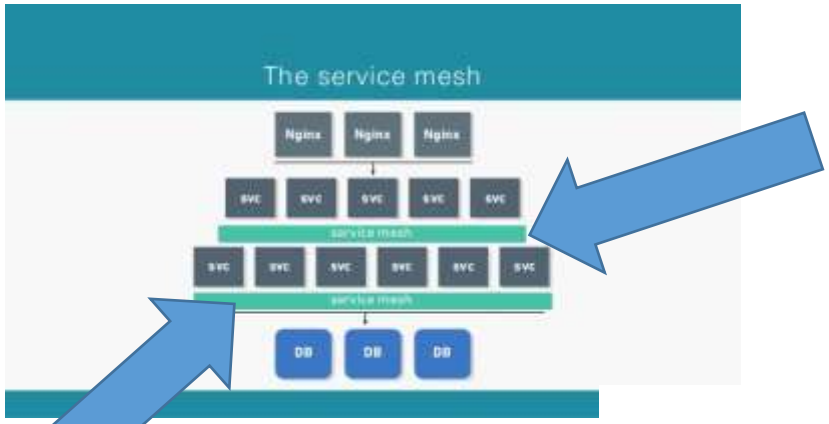
tl;dr – Service Meshes

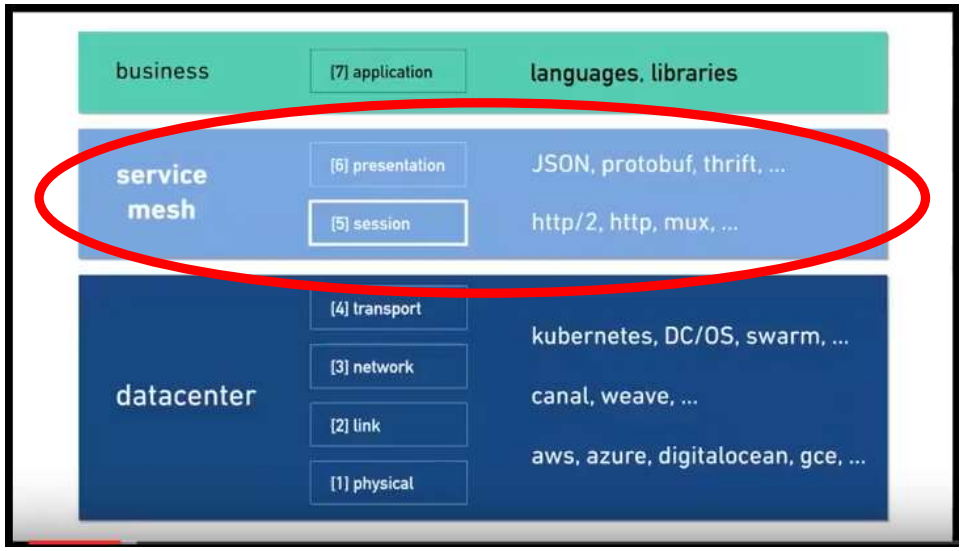
- A service mesh is a dedicated infrastructure layer for making service-to-service communication safe, fast, reliable, and (operator) configurable

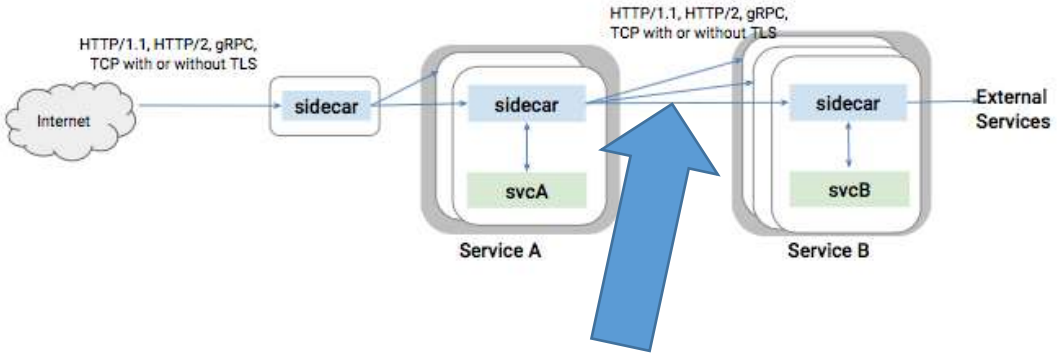
- Consists of control plane (“brains”, API, UI) and data plane (service proxies)
  - Some confusion on where the “service mesh” begins and ends

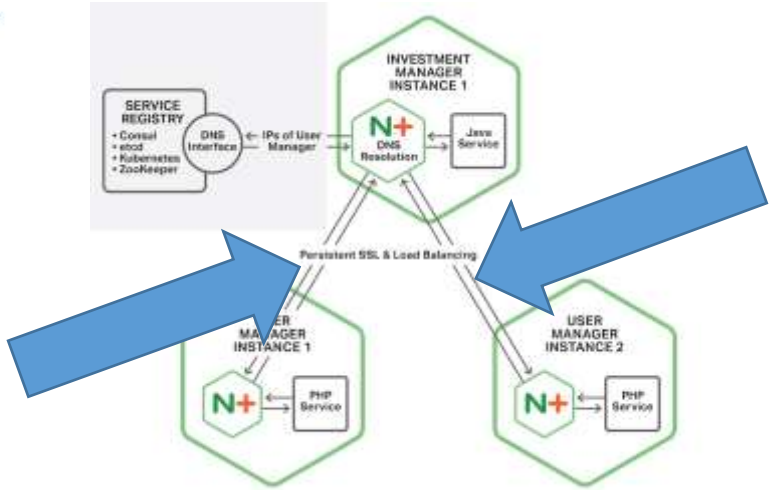
- Essential as we move from deployment of *complicated* monoliths/services to orchestration of *complex* cloud native microservices and functions

tl;dr – Service Meshes









@danielbryantuk

- Independent Technical Consultant, CTO at SpectoLabs
  - Architecture, DevOps, Java, microservices, cloud, containers
  - Continuous Delivery (CI/CD) advocate
  - Leading change through technology and teams

bit.ly/2jWDSF7

Setting the Scene

Complex
(Probe, Sense, Respond)

2010s

Microservices, functions, SaaS-all-the-things
Polyglot languages
Cloud and containers (Datacenter as a Computer)
Software-Defined Everything
Optimise for innovation (and Antifragility)
Business teams ("FinDev", SRE and Platform Team)

Complicated
(Sense, Analyse, Respond)

2000s

Monoliths, Coarse-grained SOA, SaaS
Frontend/backend language
"Co-lo" or private datacenters
Configuration management
Optimise for Recovery (MTTR)
Generalist teams (Full Stack and "DevOps")

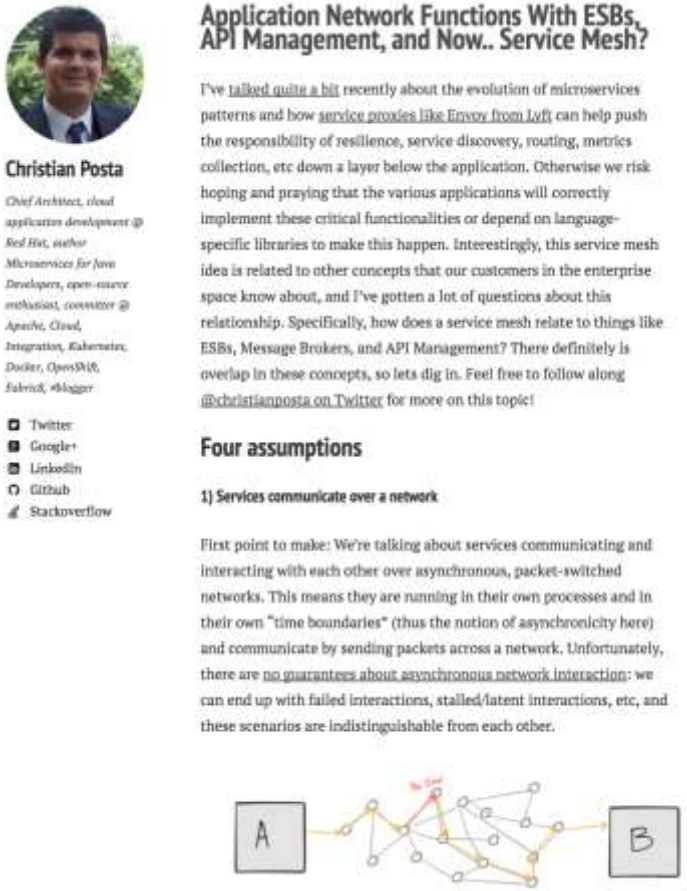
Simple
(Sense, Categorise, Respond)

1990s

Monoliths
Single language
In-house hardware (servers, SAN, networks)
Manual config and scripting
Optimise for Stability (MTBF)
Specialist staff/departments

What do ”cloud native” comms look like?

- Services communicate over a network
- These interactions are non-trivial
- Lot of value in understanding the network
- The application is ultimately responsible



Christian Posta

Chief Architect, cloud application development @ Red Hat, author Microservices for Java Developers, open-source enthusiast, committer @ Apache, Cloud, Integration, Kubernetes, Docker, OpenSWAN, Fabric8, #blogger

Twitter
Google+
LinkedIn
Github
Stackoverflow

Application Network Functions With ESBs, API Management, and Now.. Service Mesh?

I've talked quite a bit recently about the evolution of microservices patterns and how service proxies like Envoy from Lyft can help push the responsibility of resilience, service discovery, routing, metrics collection, etc down a layer below the application. Otherwise we risk hoping and praying that the various applications will correctly implement these critical functionalities or depend on language-specific libraries to make this happen. Interestingly, this service mesh idea is related to other concepts that our customers in the enterprise space know about, and I've gotten a lot of questions about this relationship. Specifically, how does a service mesh relate to things like ESBs, Message Brokers, and API Management? There definitely is overlap in these concepts, so lets dig in. Feel free to follow along @christianposta on Twitter for more on this topic!

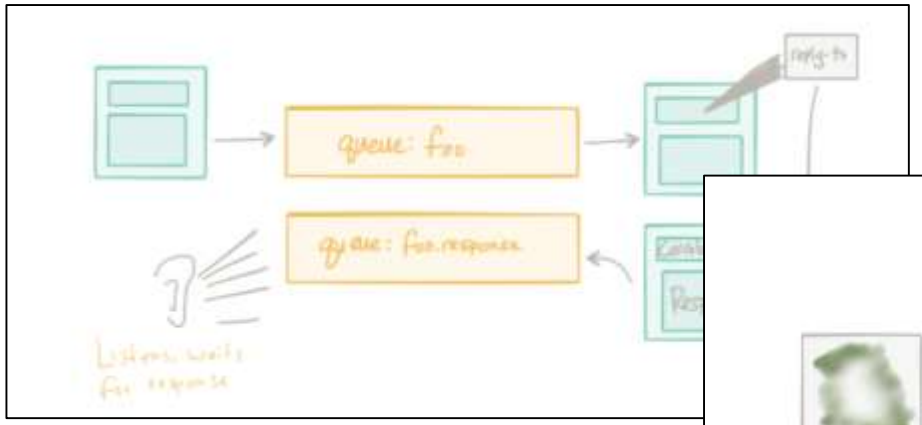
Four assumptions

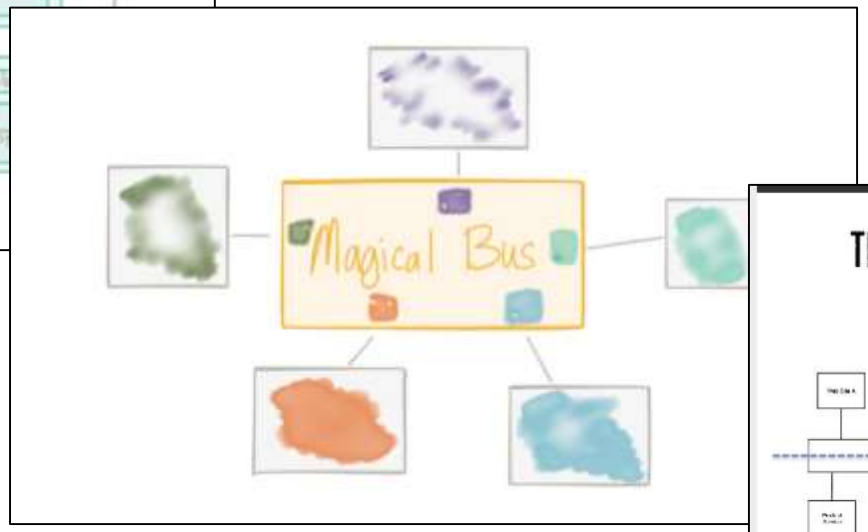
1) Services communicate over a network

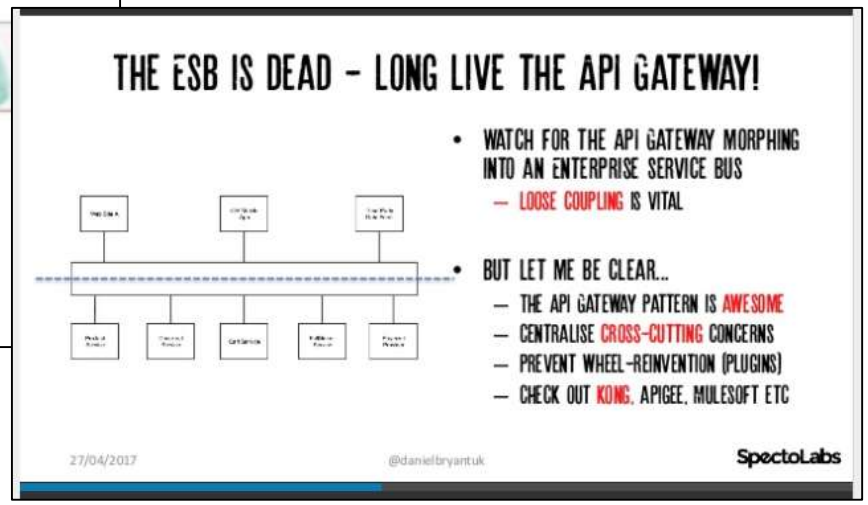
First point to make: We're talking about services communicating and interacting with each other over asynchronous, packet-switched networks. This means they are running in their own processes and in their own "time boundaries" (thus the notion of asynchronicity here) and communicate by sending packets across a network. Unfortunately, there are no guarantees about asynchronous network interaction: we can end up with failed interactions, stalled/latent interactions, etc, and these scenarios are indistinguishable from each other.

blog.christianposta.com/microservices/application-network-functions-with-esbs-api-management-and-now-service-mesh/

But we've been here before…



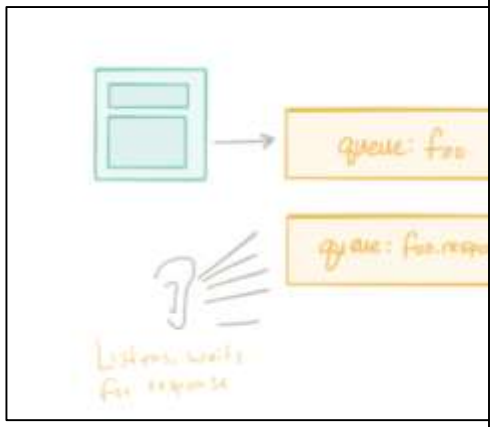




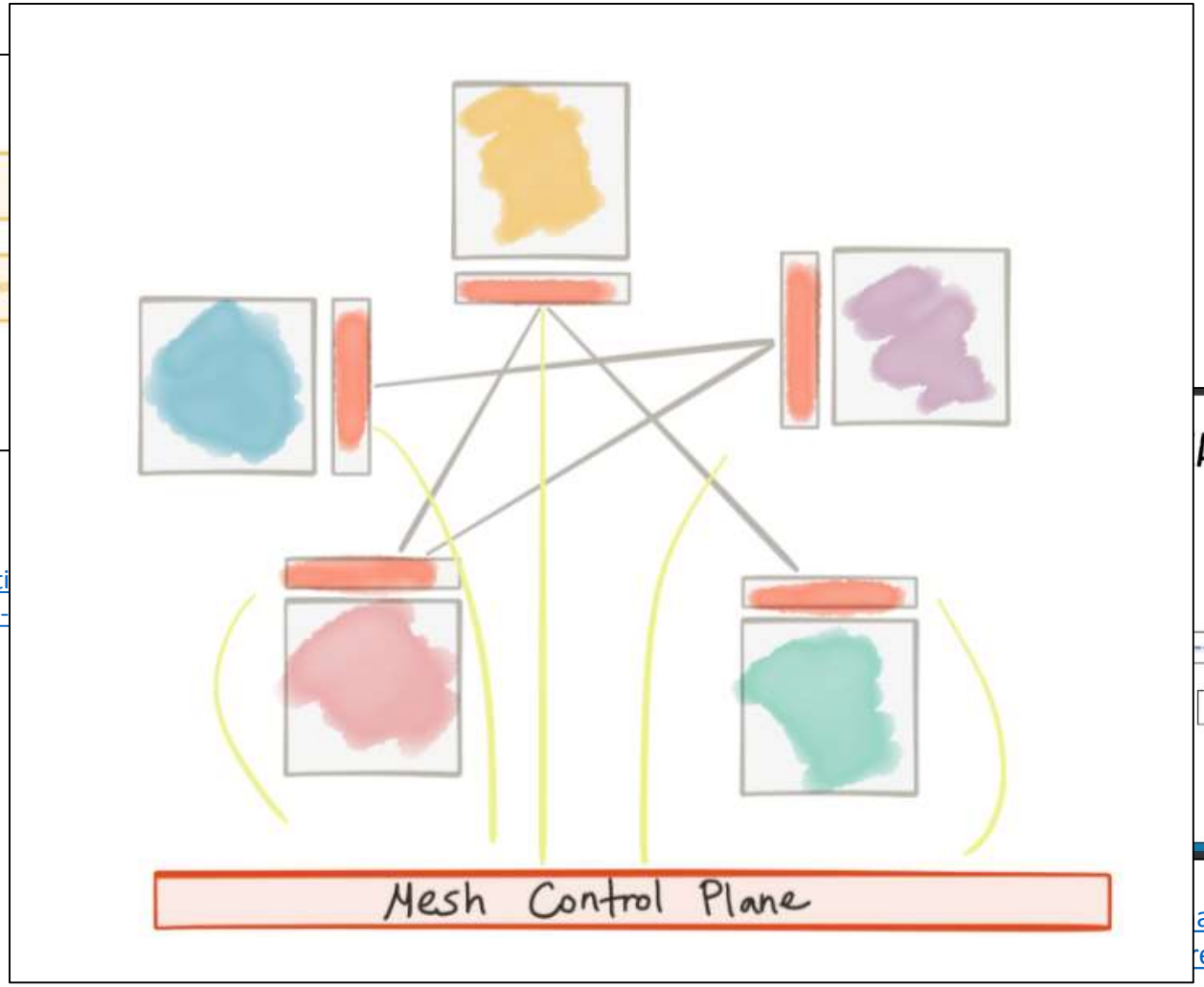
blog.christianposta.com/microservices/application-network-functions-with-esbs-api-management-and-now-service-mesh/

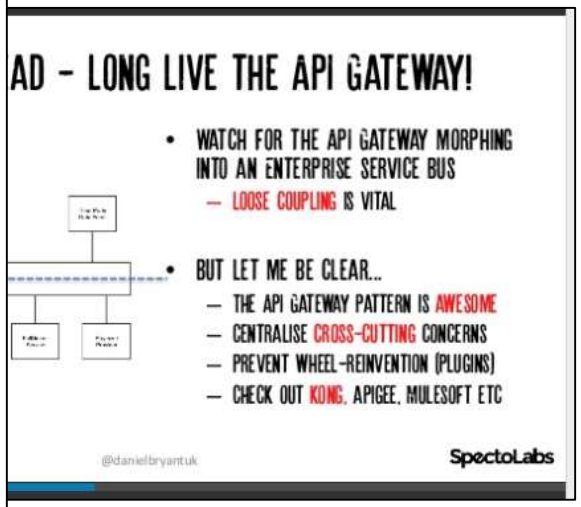
www.slideshare.net/dbryant_uk/goto-chicagocraftconf-2017-the-seven-more-deadly-sins-of-microservices

But we've been here before…



blog.christianposta.com/microservices/applicati
functions-with-esbs-api-management-and-now-





are.net/dbryant_uk/goto-chicagocraftconf-
e-deadly-sins-of-microservices

Let’s go unicorn spotting…

- Netflix
  - Karyon + HTTP/JSON or RxNetty RPC + Eureka + Hystrix + …
- Twitter
  - Finagle + Thrift + ZooKeeper + Zipkin
- Google
  - Stubby (gRPC) + GSLB + GFE + Dapper
- AirBnB
  - HTTP/JSON + SmartStack + ZooKeeper + Charon/Dyno

So, from a technology perspective…

- Deploying cloud native services/functions to a “platform” is essential
  - Abstracts underlying resources and provides runtime foundations
- Need clear collaboration zones for dev/ops/platform
  - Must also cultivate “mechanical sympathy”
- Managing lots of out-of-process communication going “over the wire”
  - We must not treat local and remote calls the same (dev, observability etc)

So, from a technology perspective…

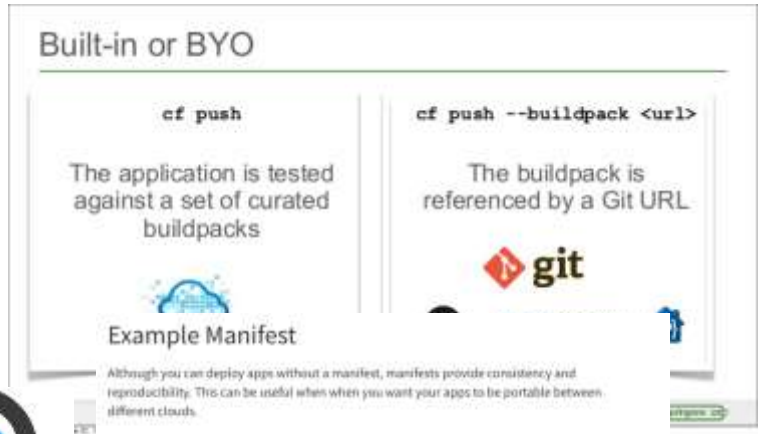
- **Deploying cloud native services/functions to a “platform” is essential**
- Need clear collaboration zones for dev/ops/platform
- Managing lots of out-of-process communication going “over the wire”

Service/function platforms

So, from a technology perspective…

- Deploying services/functions to a “platform” is essential
- **Need clear collaboration zones for dev/ops/platform**
- Managing lots of out-of-process communication going “over the wire”

Collaboration zones for deployment



So, from a technology perspective…

- Deploying services/functions to a “platform” is essential
- Need clear collaboration zones for dev/ops/platform
- **Managing lots of out-of-process communication going “over the wire”**

The Eight Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

OR

NETFLIX OSS

(~ 2013)

https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/

So, from a technology perspective…

- Deploying services/functions to a “platform” is essential
- Need clear collaboration zones for dev/ops/platform
- Managing lots of out-of-process communication going “over the wire”

But be careful, technology is seductive…

- Service meshes are an emerging and rapidly evolving space
- Only one part of cloud native solution
- For big picture and people aspects:
  - "Microservices: Org and People Impact"
  - "Seven Deadly Sins of Microservices"

https://twitter.com/KevinHoffman/status/887638576409837569

Service Mesh Functionality

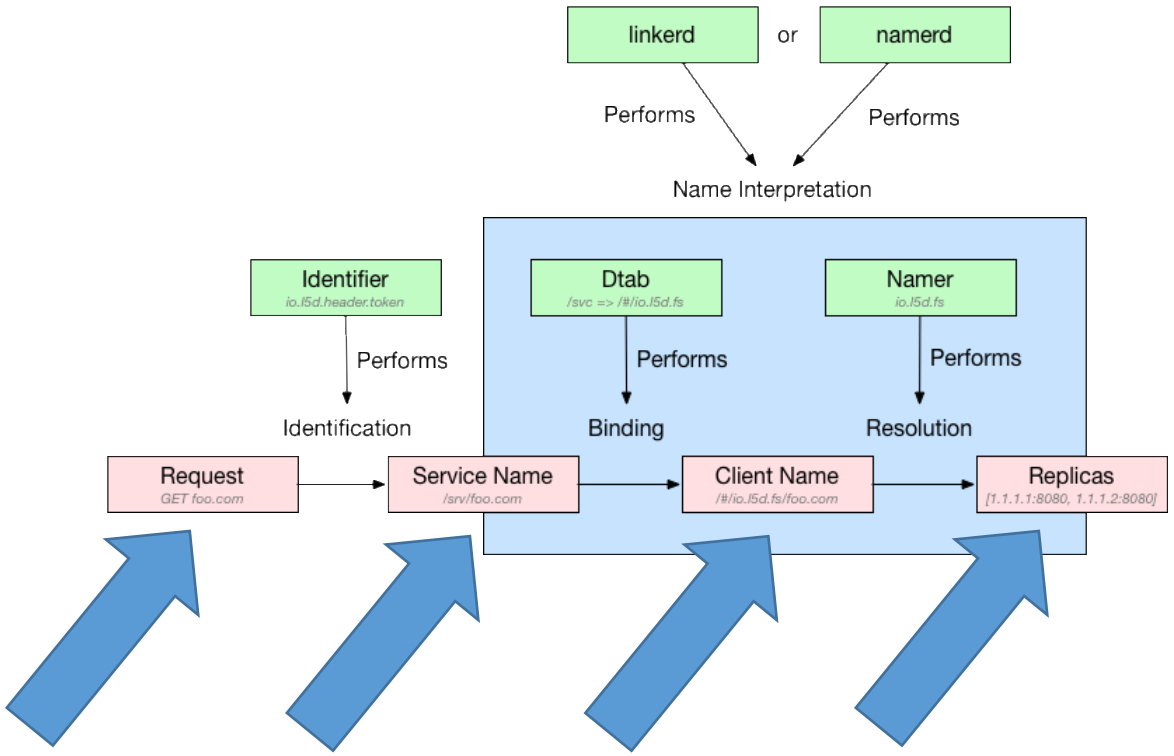
Service mesh features

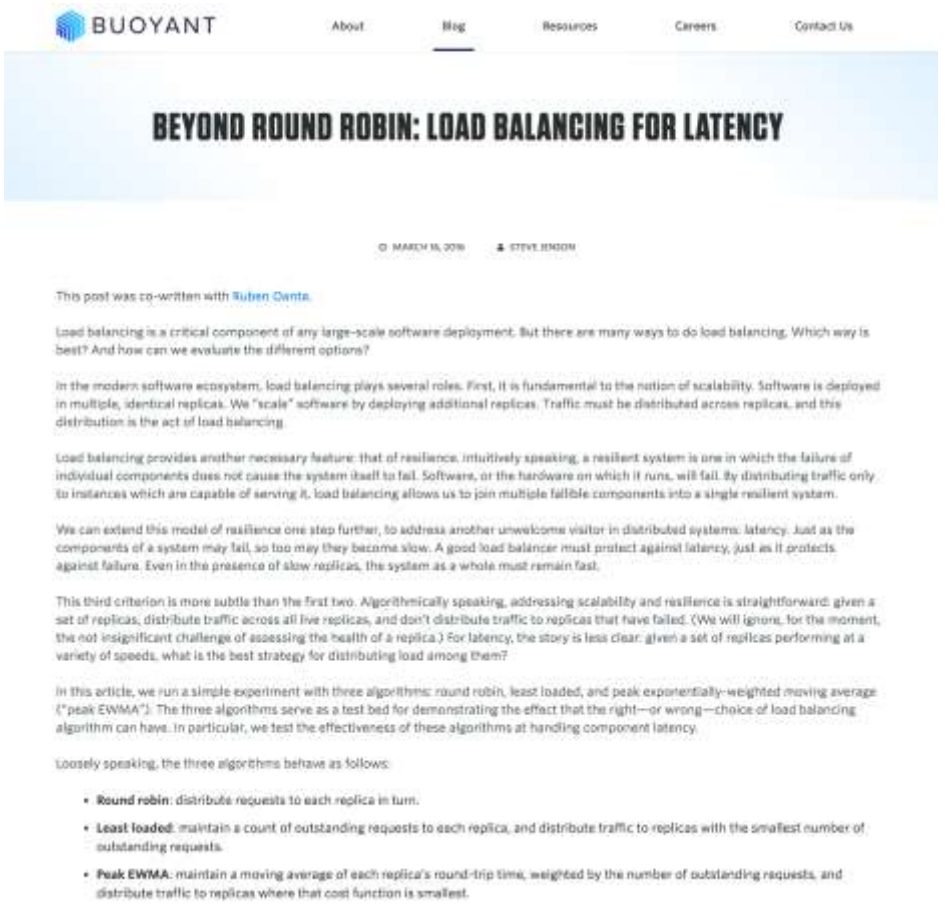
- Normalises naming and adds logical routing
  - user-service -> AWS-us-east-1a/prod/users/v4
- Adds traffic shaping and traffic shifting
  - Load balancing
  - Deploy control
  - Per-request routing (shadowing, fault injection, debug)
- Adds baseline reliability
  - Health checks, timeouts/deadlines, circuit breaking, and retry (budgets)

Service mesh features

- Increased security
  - Transparent mutual TLS
  - Policies (service Access Control Lists - ACL)
- Observability / monitoring
  - Top-line metrics like request volume, success rates and latencies
  - Distributed tracing
- Sane defaults (to protect the system)
  - With options to tune

Naming and load balancing




BUOYANT

About Blog Resources Careers Contact Us

BEYOND ROUND ROBIN: LOAD BALANCING FOR LATENCY

This post was co-written with Ruben Oanta.

Load balancing is a critical component of any large-scale software deployment. But there are many ways to do load balancing. Which way is best? And how can we evaluate the different options?

In the modern software ecosystem, load balancing plays several roles. First, it is fundamental to the notion of scalability. Software is deployed in multiple, identical replicas. We "scale" software by deploying additional replicas. Traffic must be distributed across replicas, and this distribution is the act of load balancing.

Load balancing provides another necessary feature: that of resilience. Intuitively speaking, a resilient system is one in which the failure of individual components does not cause the system itself to fail. Software, or the hardware on which it runs, will fail. By distributing traffic only to instances which are capable of serving it, load balancing allows us to join multiple fallible components into a single resilient system.

We can extend this model of resilience one step further, to address another unwelcome visitor in distributed systems: latency. Just as the components of a system may fail, so too may they become slow. A good load balancer must protect against latency, just as it protects against failure. Even in the presence of slow replicas, the system as a whole must remain fast.

This third criterion is more subtle than the first two. Algorithmically speaking, addressing scalability and resilience is straightforward: given a set of replicas, distribute traffic across all live replicas, and don't distribute traffic to replicas that have failed. (We will ignore, for the moment, the not insignificant challenge of assessing the health of a replica.) For latency, the story is less clear: given a set of replicas performing at a variety of speeds, what is the best strategy for distributing load among them?

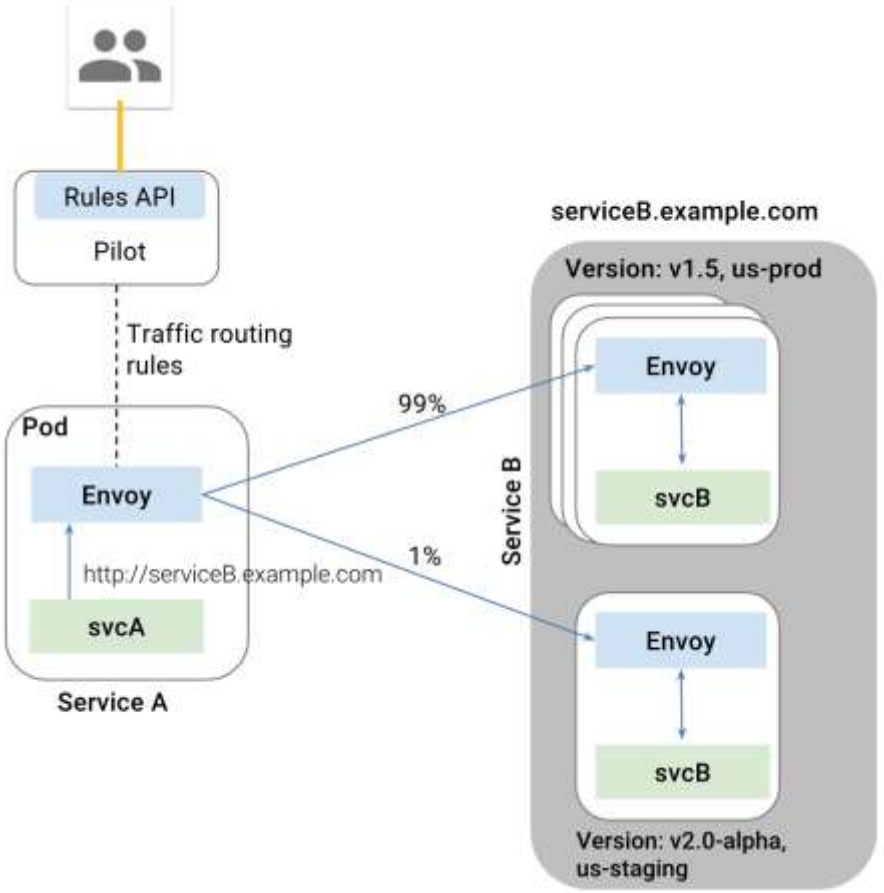
In this article, we run a simple experiment with three algorithms: round robin, least loaded, and peak exponentially-weighted moving average ("peak EWMA"). The three algorithms serve as a test bed for demonstrating the effect that the right—or wrong—choice of load balancing algorithm can have. In particular, we test the effectiveness of these algorithms at handling component latency.

Loosely speaking, the three algorithms behave as follows:

- **Round robin**: distribute requests to each replica in turn.
- **Least loaded**: maintain a count of outstanding requests to each replica, and distribute traffic to replicas with the smallest number of outstanding requests.
- **Peak EWMA**: maintain a moving average of each replica's round-trip time, weighted by the number of outstanding requests, and distribute traffic to replicas where that cost function is smallest.

https://buoyant.io/2016/03/16/beyond-round-robin-load-balancing-for-latency/

Traffic control




```
destination: serviceB.example.cluster.local match:
source: serviceA.example.cluster.local route:
- tags:
    version: v1.0
    env: uk-prod
  weight: 90
- tags:
    version: v2.0-RC
    env: uk-staging
weight: 10
```

```
destination: serviceB.example.cluster.local match:
  httpHeaders:
    user-a
regex: ^(.
route: - t
    versio
```

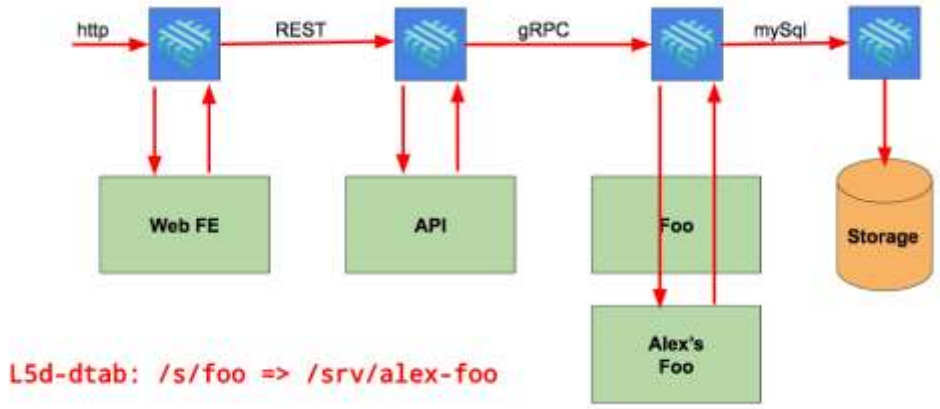
```
ping $ cat istio/route-rules/bar.yaml
type: route-rule
name: bar-default
spec:
  destination: bar.default.svc.cluster.local
  precedence: 1
  route:
    - tags:
        version: v1
      weight: 100
  httpReqRetries:
    simpleRetry:
      attempts: 3
      perTryTimeout: 5s
  httpReqTimeout:
    simpleTimeout:
      timeout: 10s
ping $
```

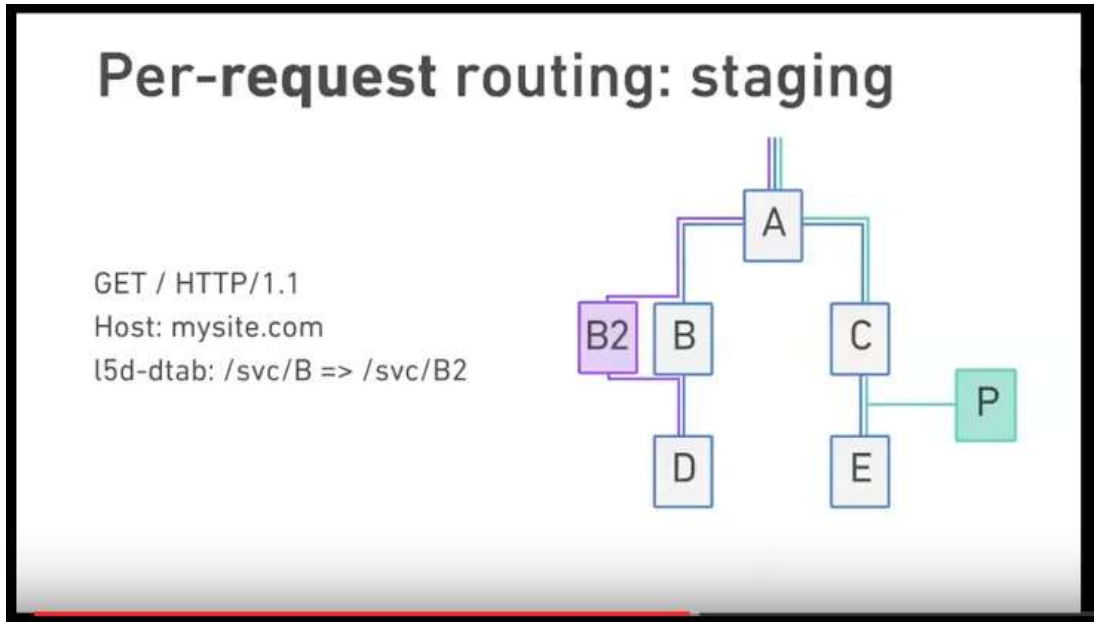
27:02 / 29:57

https://istio.io/docs/concepts/traffic-management/request-routing.html

https://www.youtube.com/watch?v=s4qasWn_mFc

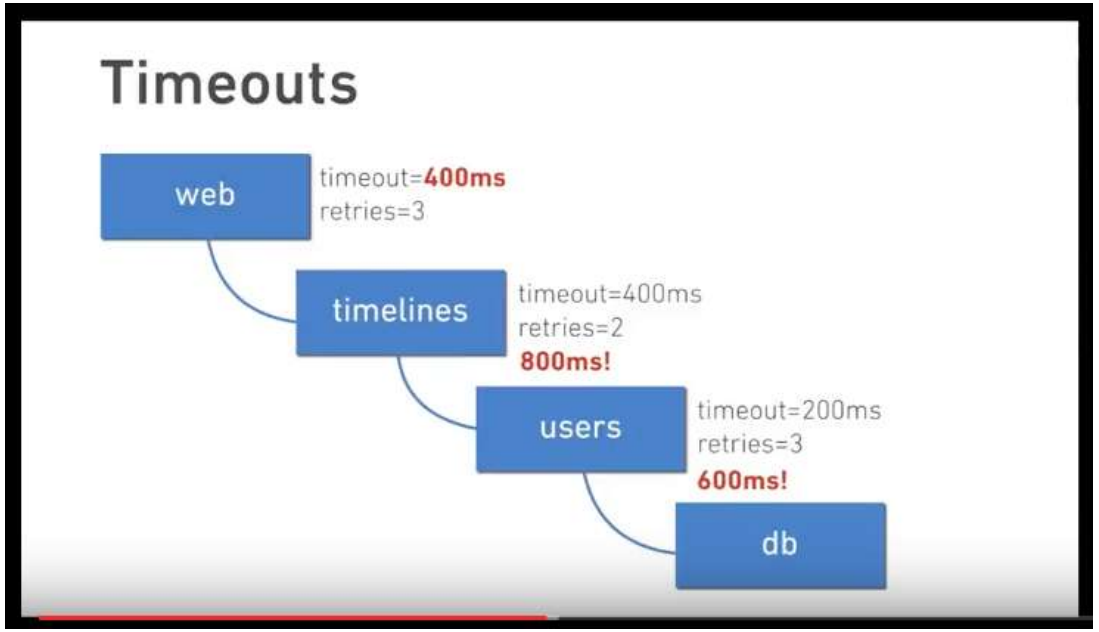
Per-request routing: shadow, fault inject, debug

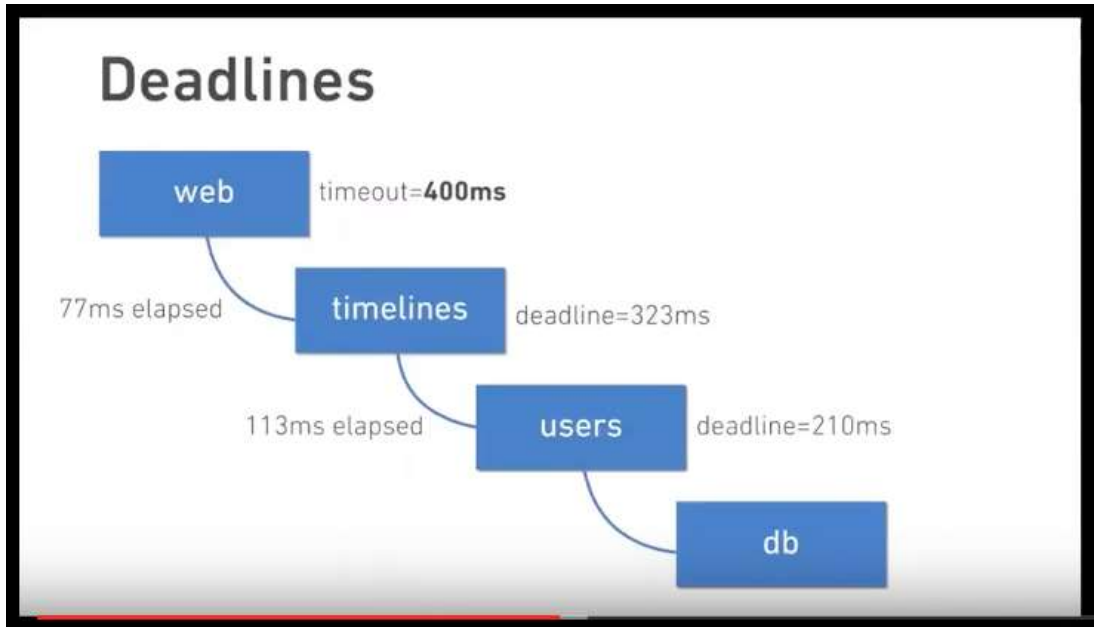




https://buoyant.io/2017/01/06/a-service-mesh-for-kubernetes-part-vi-staging-microservices-without-the-tears/

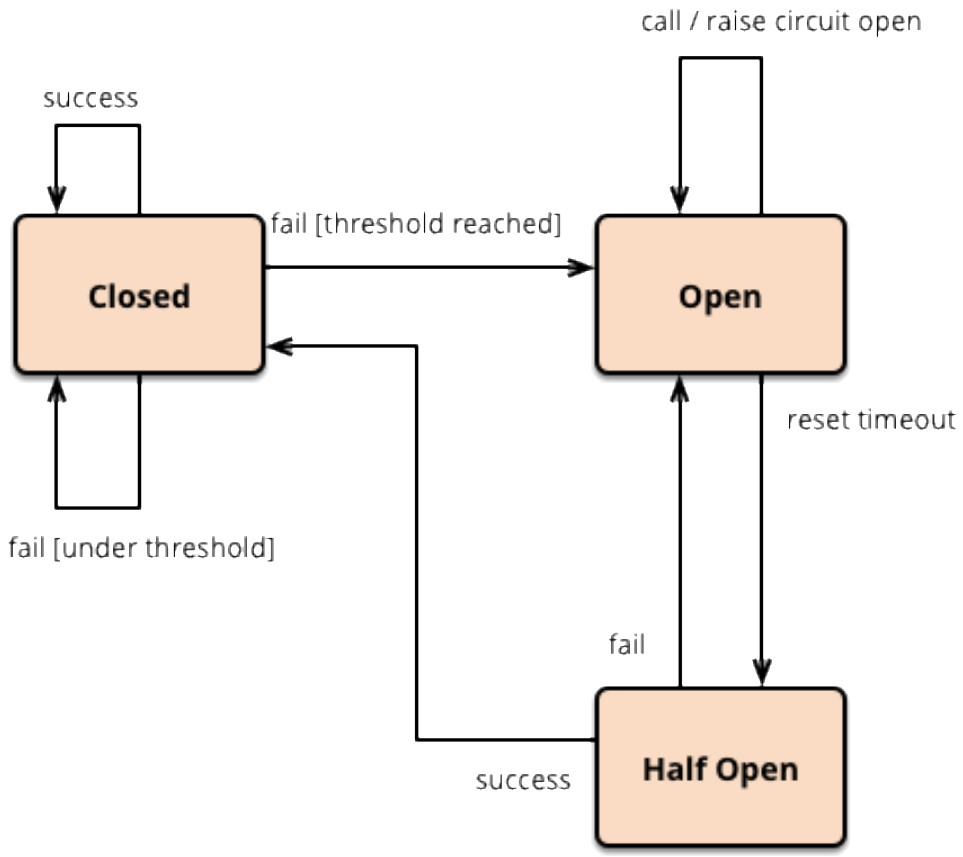
Timeouts / deadlines





[William Morgan Introduction to Linkerd: https://www.youtube.com/watch?v=0xYSy6OmjUM]

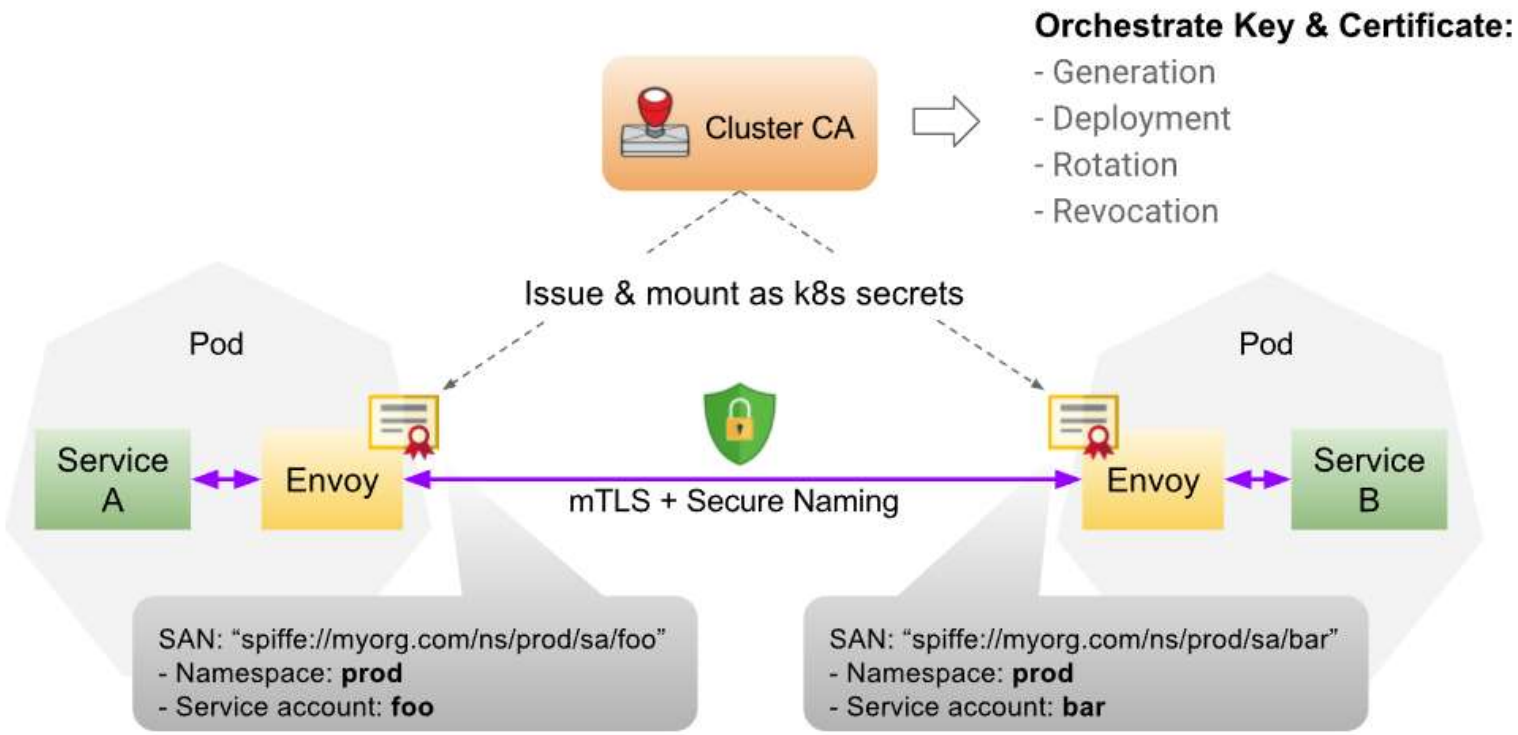
Circuit breaking (out of process, not Hystrix)




```
destination: serviceB.example.cluster.local policy:
- tags:
    version: v1
  circuitBreaker:
    simpleCb:
      maxConnections: 100
      httpMaxRequests: 1000
      httpMaxRequestsPerConnection: 10
      httpConsecutiveErrors: 7
      sleepWindow: 15m
      httpDetectionInterval: 5m
```

```
- protocol: http
  client:
    failureAccrual:
      kind: io.l5d.successRate
      successRate: 0.9
      requests: 20
      backoff:
        kind: constant
        ms: 10000
```

Mutual TLS (transparent protocol upgrade)



https://istio.io/blog/istio-auth-for-microservices.html

Communication policies

```
aspects:
- kind: lists              # the aspect's kind
  adapter: myListChecker   # the adapter to use to implement this aspect
  params:
    blacklist: true
    checkExpression: source.ip
```

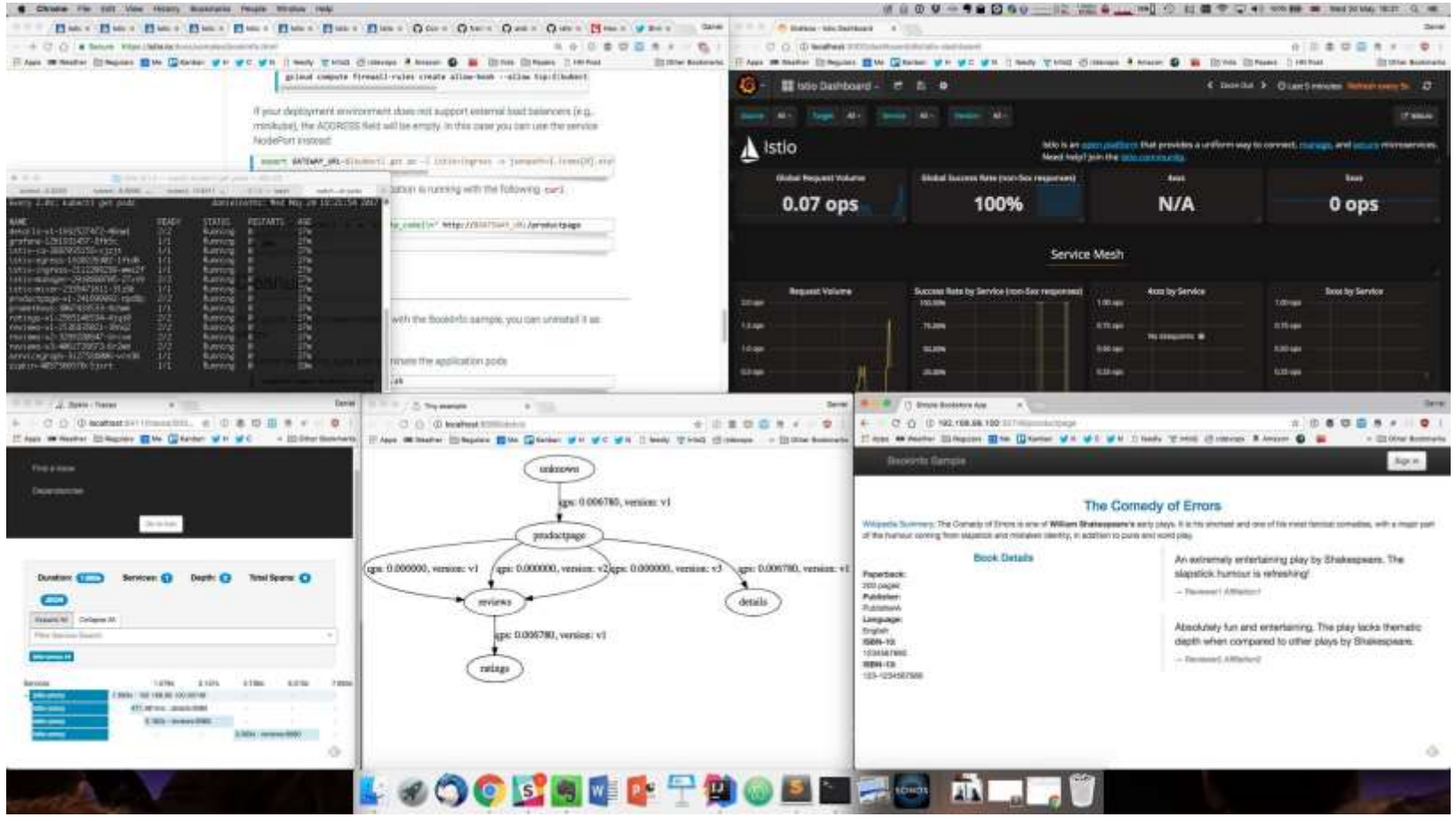
The `kind` field distinguishes the behavior of the aspect being defined. The supported kinds of aspects are shown in the following table.

Kind	Description
quotas	Enforce quotas and rate limits.
metrics	Produce metrics.
lists	Enforce simple whitelist- or blacklist-based access control.
access-logs	Produces fixed-format access logs for every request.
application-logs	Produces flexible application logs for every request.
attributes	Produces supplementary attributes for every request.
denials	Systematically produces a predictable error code.

https://istio.io/docs/concepts/policy-and-control/mixer-config.html#aspects

https://www.projectcalico.org/network-policy-and-istio-deep-dive/

Visibility

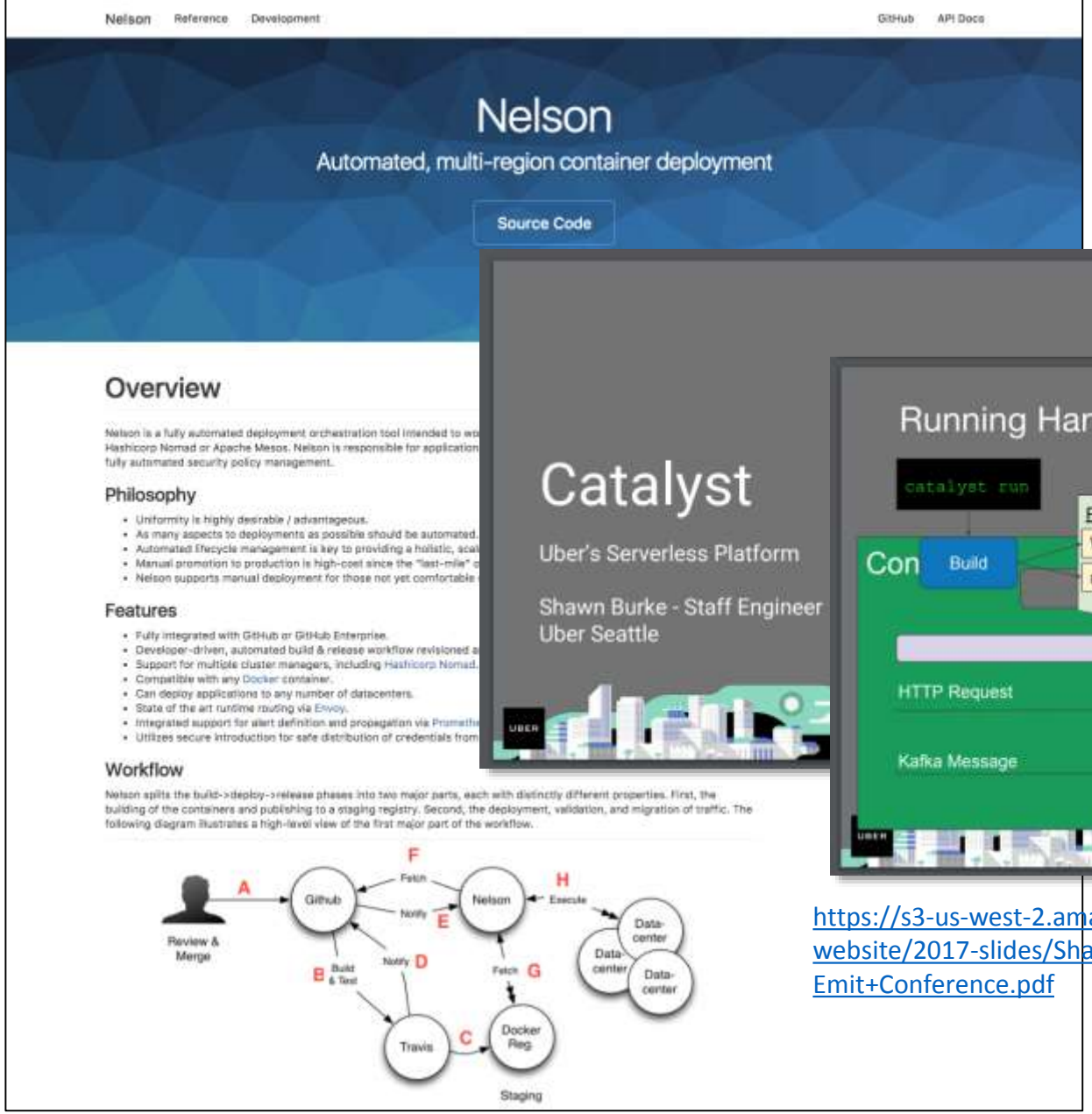

Istio Dashboard
Istio
0.07 ops
100%
N/A
0 ops
Service Mesh
unknown
productpage
reviews
details
ratings
The Comedy of Errors
Book Details

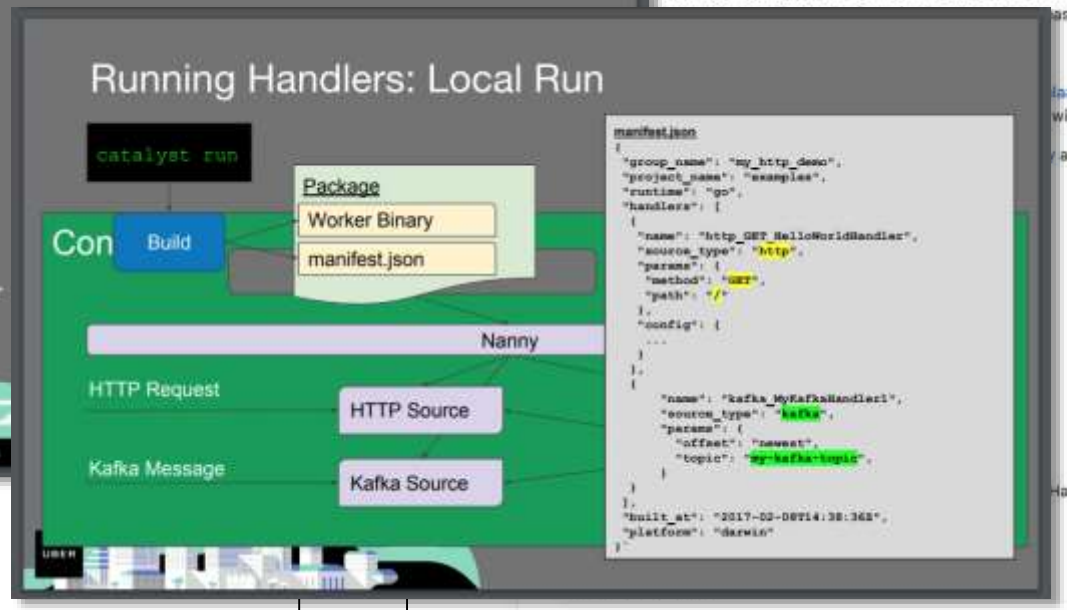
Service Mesh Implementations


envoy
by lyft


NGINX


linkerd





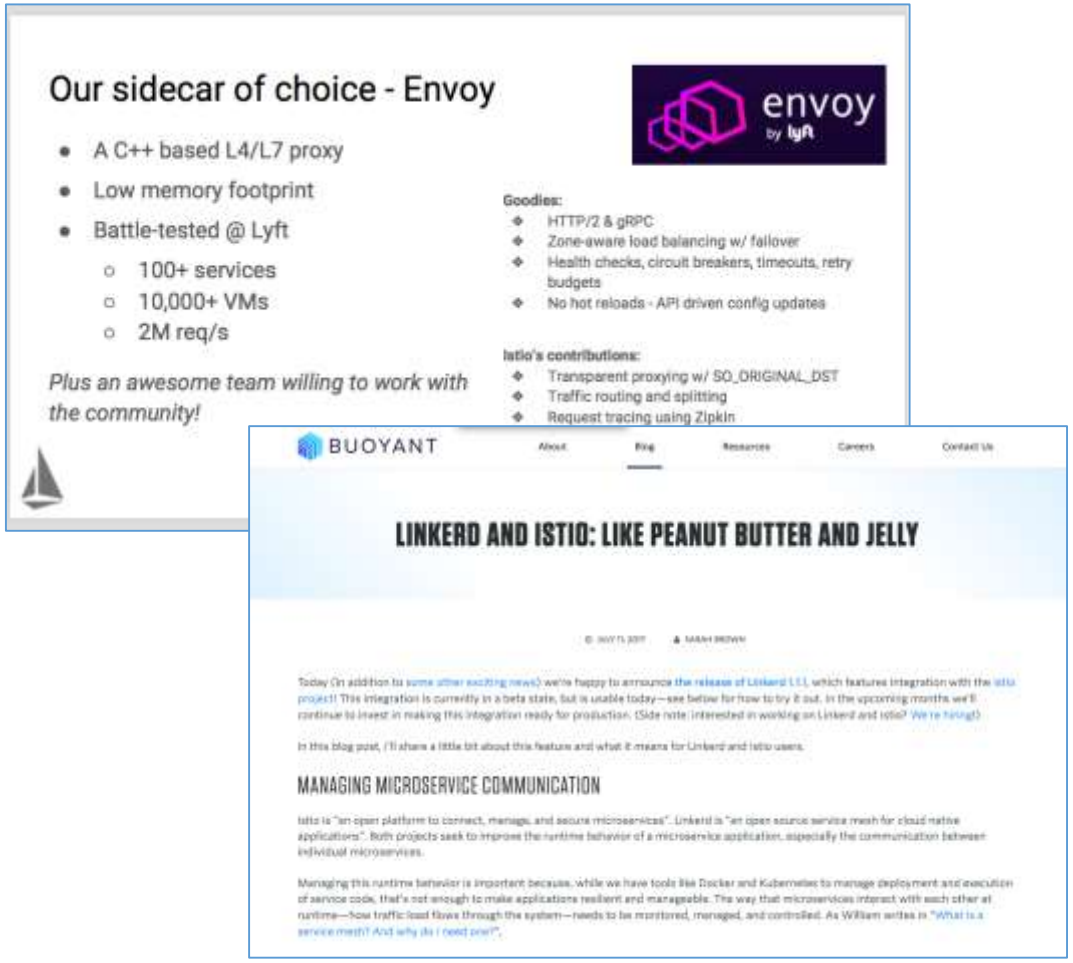
https://s3-us-west-2.amazonaws.com/emit-website/2017-slides/Shawn-Catalyst-Emit+Conference.pdf

https://verzion.github.io/nelson/

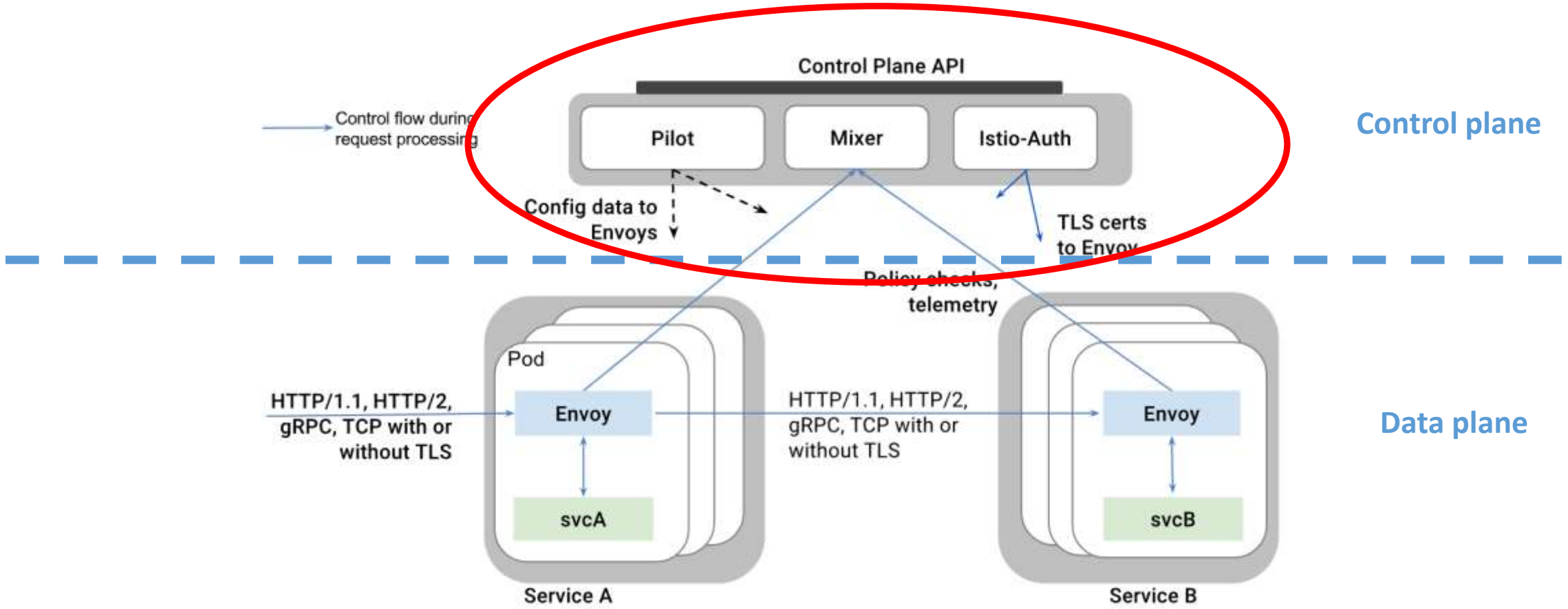
https://github.com/fabiolb/fabio

Putting it all together: Istio

- "Istio" is an open platform
  - Connect, manage, secure services
- Proxies are the data plane / mesh
- Proxies are (in theory) swappable
  - But in reality there are different feature sets, security, performance

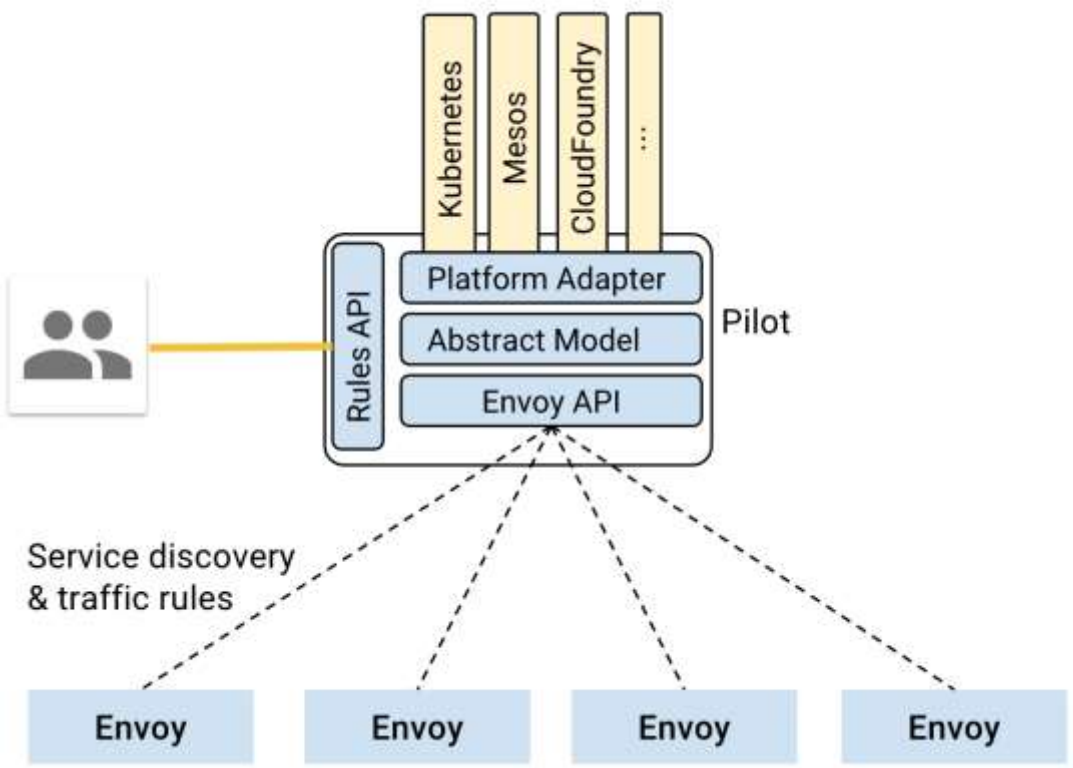


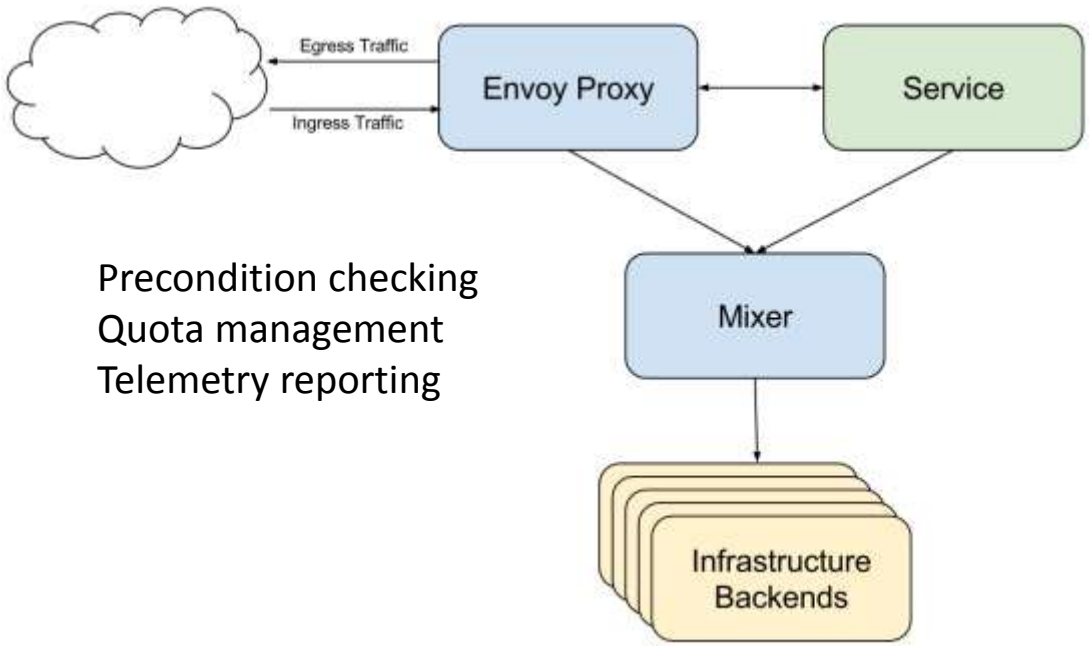
Control Plane / Data Plane (Istio example)



https://istio.io/docs/concepts/what-is-istio/overview.html

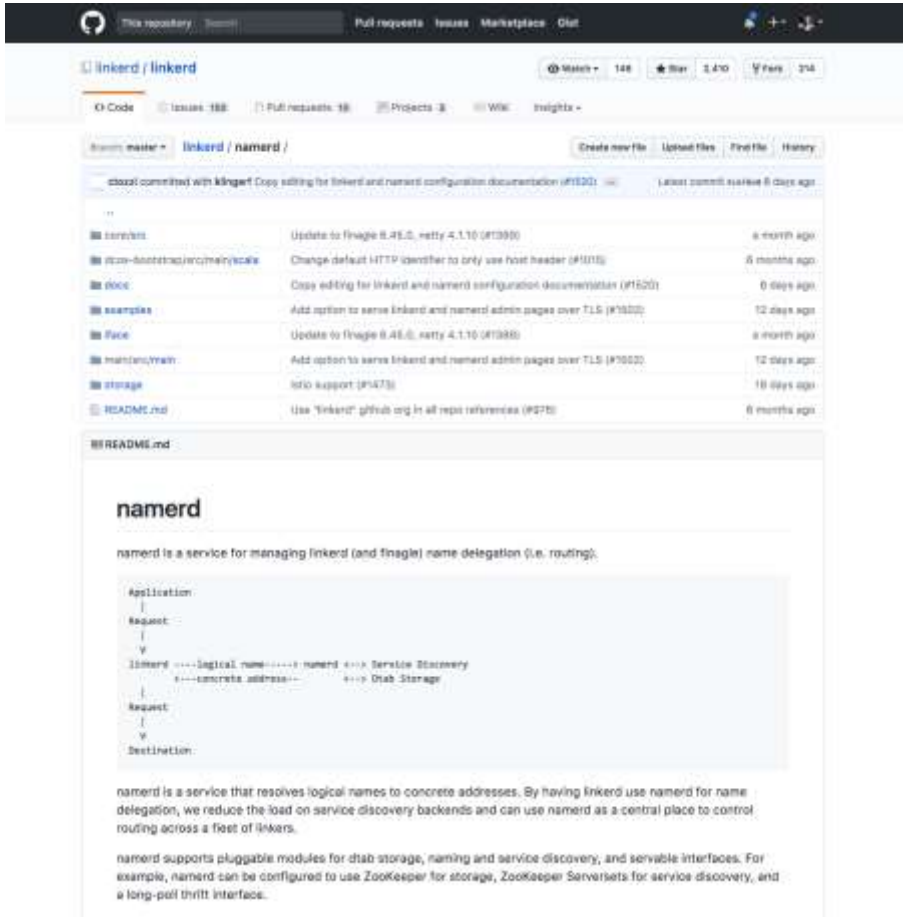
Istio control plane: Pilot and Mixer

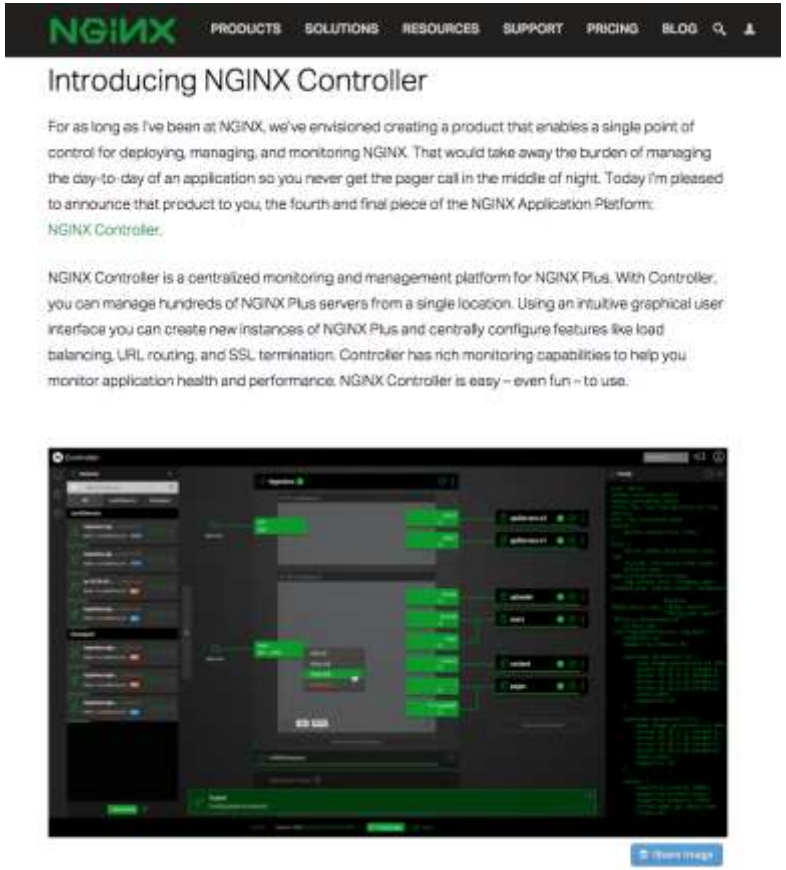




Precondition checking
Quota management
Telemetry reporting

Linkerd and NGINX control plane

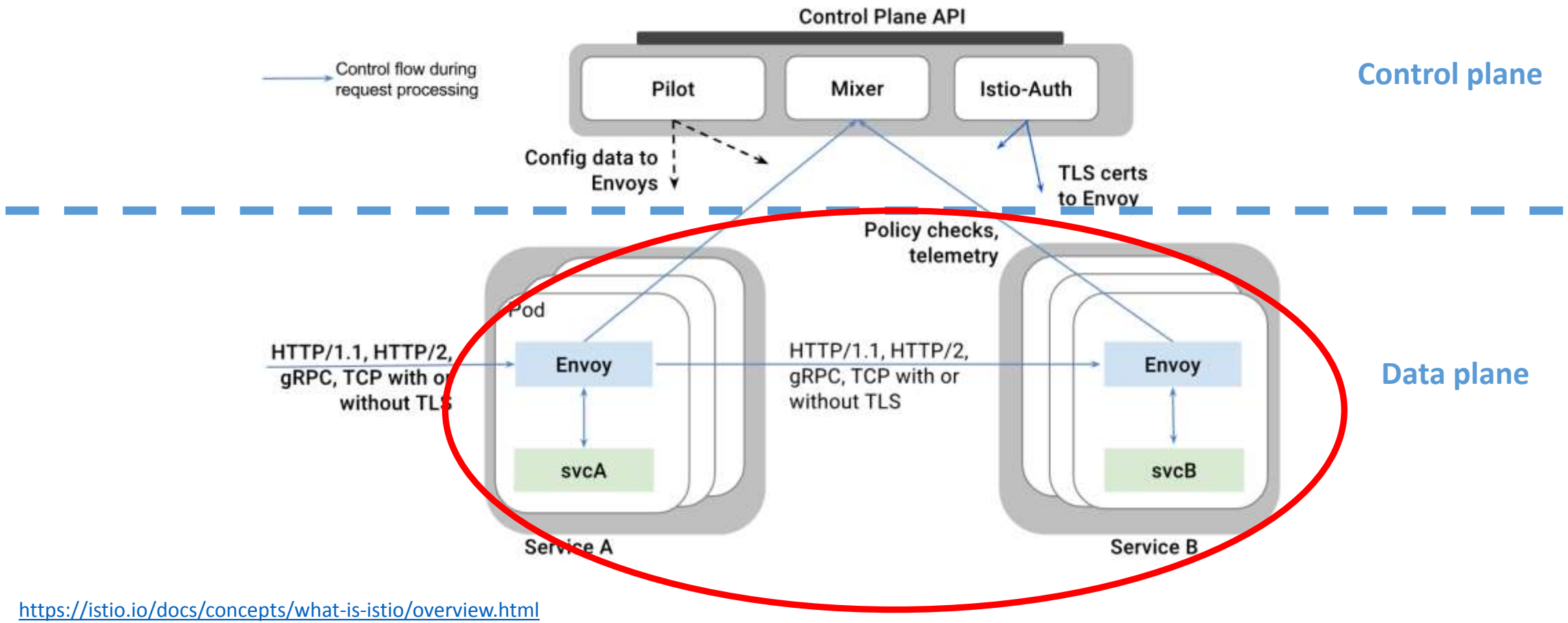




NGINX Controller makes it easier to monitor and manage large NGINX Plus deployments.

www.infoq.com/news/2017/09/nginx-platform-service-mesh

Control Plane / Data Plane (Istio example)

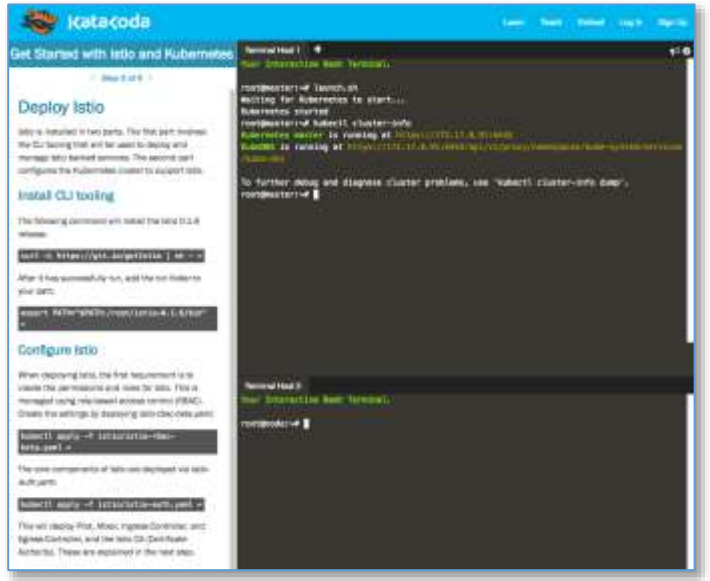


https://istio.io/docs/concepts/what-is-istio/overview.html

Service Mesh data plane (proxy) comparison

Name	Linkerd	Envoy	Nginx Plus
Website	https://linkerd.io/	https://lyft.github.io/envoy/	https://www.nginx.com/products/
Licence	AL2.0	AL2.0	Commercial
Custodians	Buoyant (Twitter pedigree)	Lyft (Matt Klein)	Nginx Inc
Build language	Scala/Rust	C++11	C
Deployment platforms	Any	Any (Istio is K8s only)	Any
Control plane	APIs, namerd, Istio [exp]	APIs, Istio (Google/IBM), Nelson (Verizon)	API, config files, service discovery
Protocols supported	HTTP/1.1, HTTP/2 [exp], gRPC, TCP	HTTP/2, gRPC, TCP	HTTP/1.1, TCP, UDP, *gRPC (Q4 2017)*
Service discovery integration	File, Consul, ZK, etcd, K8s, Marathon	Static, Strict/logical DNS, SDS (REST API)	File, Consul, ZK, etcd, K8s, Marathon (nixy)
Container Deployment	Per Host/Sidecar	PerHost/Sidecar (Istio sidecar only)	Sidecar
Monitoring	Exports stat (linkerd-viz)	Exports stats (e.g. Prometheus/Grafana)	Exports stats, and bespoke dashboard
Request Tracing	Zipkin (service must pass http headers)	Zipkin (service must pass http headers)	Possible, but bespoke
Commercial support	Yes (Buoyant)	No (Not yet)	Yes (Nginx Inc)
Production Deploys	PayPal, Expedia, AOL, Monzo	Lyft, Verizon, JDI	Buzzfeed, WIX, ARM

Getting started



https://www.katacoda.com/courses/istio/deploy-istio-on-kubernetes

- Articles:
  - Linkerd + Kubernetes:
    - https://buoyant.io/2016/10/04/a-service-mesh-for-kubernetes-part-i-top-line-service-metrics/
  - Installing Istio:
    - https://istio.io/docs/tasks/installing-istio.html
  - Tim Perrett: Envoy with Nomad and Consul
    - http://timperrett.com/2017/05/13/nomad-with-envoy-and-consul
  - NGINX Fabric Model:
    - https://www.nginx.com/blog/microservices-reference-architecture-nginx-fabric-model/

- Videos:
  - William Morgan - Linkerd:
    - https://www.youtube.com/watch?v=0xYSy6OmjUM
  - Christian Posta – Envoy/Istio:
    - http://blog.christianposta.com/microservices/00-microservices-patterns-with-envoy-proxy-series/
  - Matt Klein – Envoy:
    - https://www.youtube.com/watch?v=RVZX4CwKhGE
  - Kelsey Hightower - Istio:
    - https://www.youtube.com/watch?v=s4qasWn_mFc

Common Questions

“But what about…"

How do service meshes relate to (Edge/API) gateways?

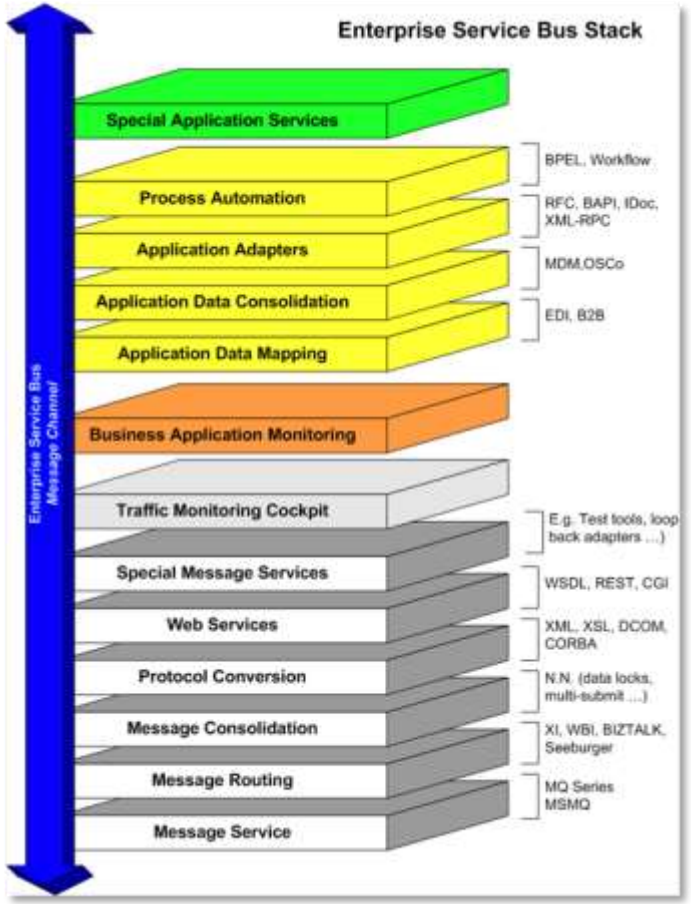
- Gateways primarily sit on the edge of your network
  - Perform ingress cross-cutting concerns (authn/z, rate limiting, logging etc)

- My experience
  - NGINX
  - Cloud implementations
  - Traefik and Datawire's Ambassador (based on Envoy)

- Some are vying to act as the communication backbone too
  - Kong API
  - Mulesoft
  - NGINX

Isn’t this just ESB 2.0 or “web scale” ESB

- No
  - At least not yet…
- ESB development was vendor-driven
- Overly centralised/coupled/conflated
  - Process choreography
  - Document transformation
  - Tight integration with vendor products



https://en.wikipedia.org/wiki/Enterprise_service_bus#/media/File:ESB_Component_Hive.png

Isn’t this just adding more network hops?

- Maybe… It depends on your network config

- …but good (infrastructure) architecture is all about
  - Choosing the right abstraction
  - Making trade-offs
  - Separation of concerns

- Make an educated choice with your platform, and make it explicitly

Shouldn’t this be part of the “platform”?

- Yep…

- And it probably will be in the near future
  - But expect much innovation (and change) over the next 6-12 months

- Assess if it will be beneficial for your organisation to leverage this now

Who owns the Service Mesh? Dev, SREs, Ops?

- Yes…

- As mentioned earlier
  - We work with a sociotechnical system when delivering value/software
  - Everything is context dependent (on your organisation)
  - But deployment descriptor and service mesh config can provide good dev/ops collaboration zones as part of the “platform”

- Make a decision, communicate it, and regularly retrospect

So, Service Mesh all-the-things… right?

- No…
  - It’s all about context and trade-offs
- Service meshes are great for point-to-point RPC
- Messaging is useful to decouple services in space and time
  - Async work queues, pub/sub, topics e.g. RabbitMQ
  - Distributed txn logs and stream processing e.g. Kafka

Look for Problems, Not Solutions

Use cases for Service Meshes

- Real-time (operator) configuration and observability
- The evolution from complicated to complex systems
- Monolith-to-service migration
  - All components can use the same communication fabric
  - Multi-platform / hybrid cloud etc
  - Routing (shadow traffic, A/B, canarying etc)

Wrapping Up

In conclusion…

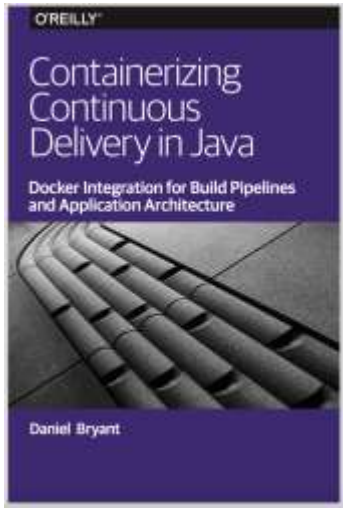
- Deploying cloud native services/functions to a “platform” is essential
  - Service meshes are responsible for platform comms e.g. routing, traffic shifting
- Need clear collaboration zones for dev/ops/platform
  - Service meshes can provide collaboration zone for run-time config of comms
- Managing lots of out-of-process communication going “over the wire”
  - Service meshes can provide observability, reliability and fault tolerance

Massive thanks to everyone who has helped!

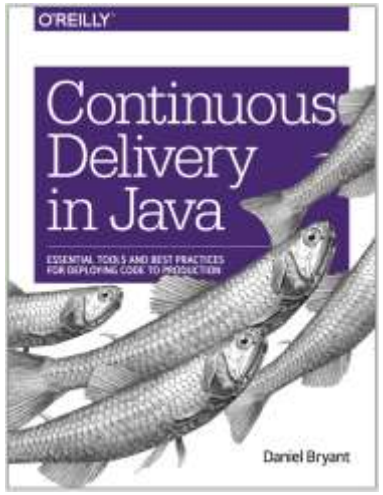
- William Morgan @ Buoyant
- Owen Garrett @ NGINX
- Christian Posta @ Red Hat
- Matt Klein @ Lyft
- Shriram Rajagopalan (Istio-users)
- Louis Ryan (Istio-users)
- Varun Talwar @ Google
- Many more from the community

Thanks for listening…

bit.ly/2jWDSF7



Available Q2 2018!



Twitter: @danielbryantuk

Email: daniel.bryant@specto.io

Writing: www.infoq.com/profile/Daniel-Bryant

Talks: www.youtube.com/playlist?list=PLoVYf_0qOYNeBmrpjuBOOAqJnQb3QAEtM