

# Service meshes

---

# What is a Service Mesh?

---

A service mesh is a configurable infrastructure layer for a microservices application. It makes communication between service instances flexible, reliable, and fast. The mesh provides service discovery, load balancing, encryption, authentication and authorization, support for the circuit breaker pattern, and other capabilities.

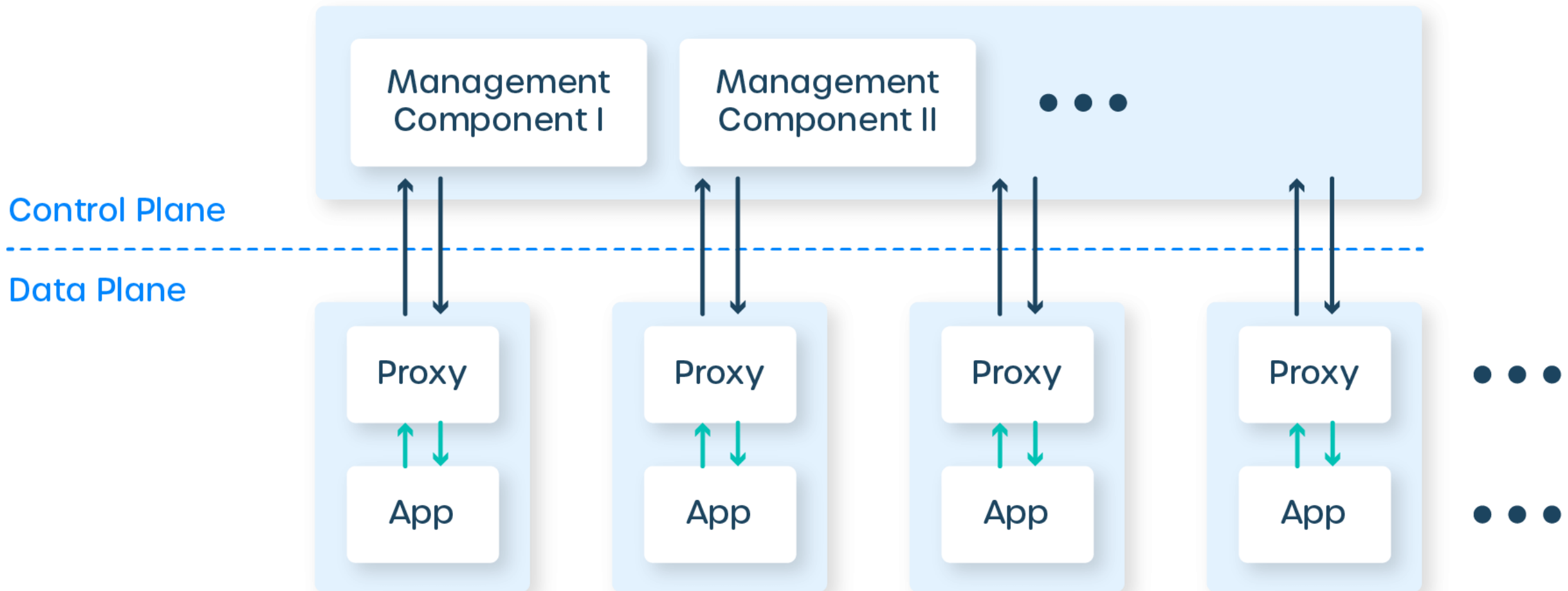
# Sidecars

---

The service mesh is usually implemented by providing a proxy instance, called a sidecar, for each service instance.

Sidecars handle inter-service communications, monitoring, security-related concerns – anything that can be abstracted away from the individual services.

This way, developers can handle development, support, and maintenance for the application code in the services; operations can maintain the service mesh and run the app.



# Popular versions

---

## Envoy

a modular and extensible proxy favored due to modular construction, open architecture and dedication to remaining unmonitized. Often used as a data plane under other tools of a service mesh.

<https://www.envoyproxy.io>

## Istio

a powerful tool set which leverages Envoy proxies via a multi-component control plane. Built to be platform independent it can be used to make the service mesh flexible and feature filled. <https://istio.io>

## Consul

a powerful platform agnostic service discovery and service mesh platform. Allows to run control plane in Kubernetes or one of the many other runtimes and allows to integrate non-Kubernetes services as well. Integrates with secure Vault for better secret storage <https://www.hashicorp.com/products/consul>

## Linkerd

Another service mesh purpose built to be easy to deploy, fast, and ultralight. <https://linkerd.io/>

## Dapr.io

Etc..

# What is Linkerd?

---

Linkerd is a *service mesh* for Kubernetes.

It makes running services easier and safer by giving you runtime debugging, observability, reliability, and security—all without requiring any changes to your code.

It adds:

- monitoring
- circuit-breaking,
- latency-aware load balancing,
- eventually consistent (“advisory”) service discovery,
- retries,
- and deadlines

---

No need to build the networking complexity into your web app / api anymore

Support for HTTP, HTTP/2, and gRPC connections

mutual Transport Layer Security (mTLS)

Load Balancing

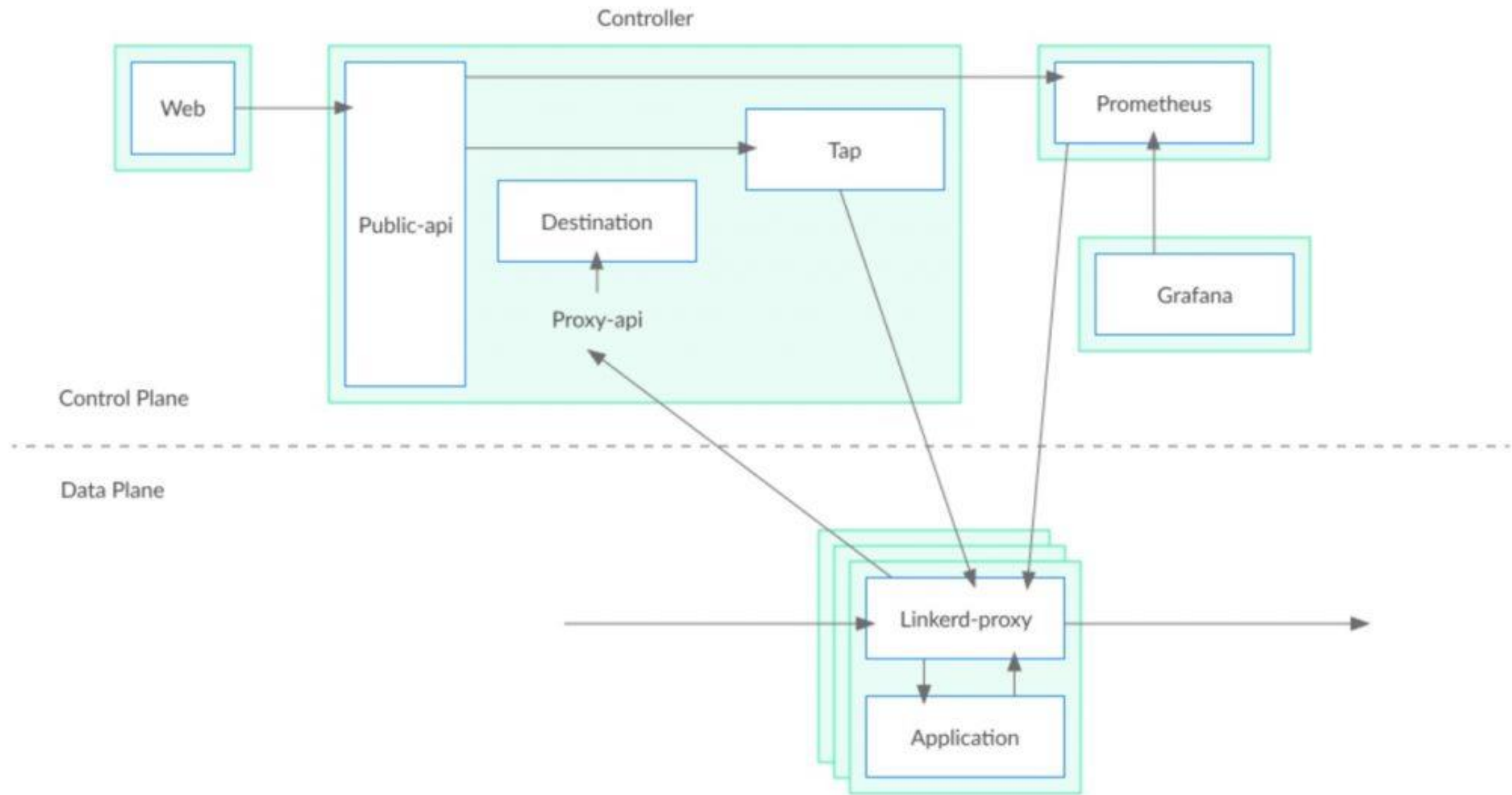
Retries and timeouts

Telemetry and monitoring

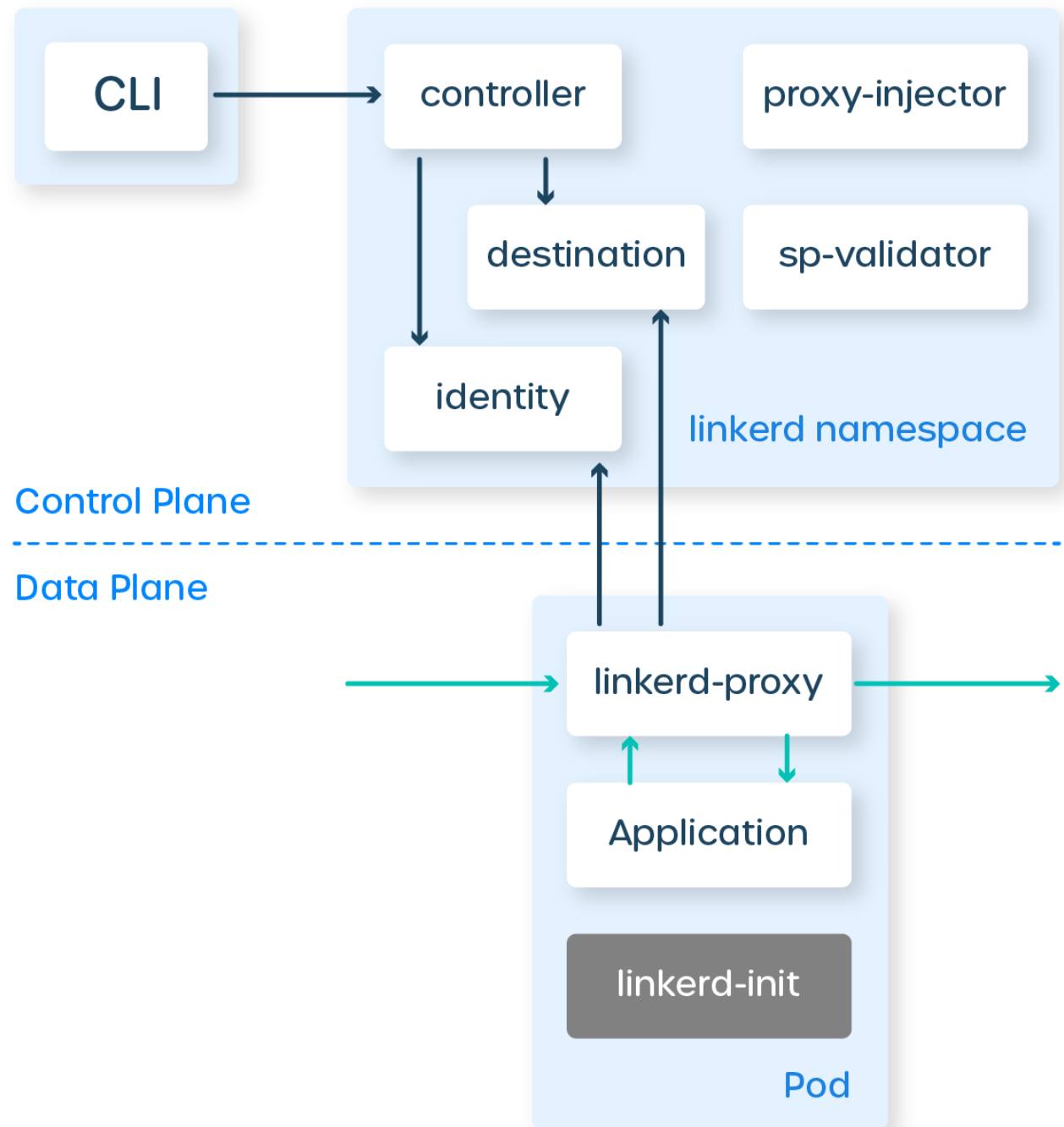
Traffic split

- Green – blue deployments
- Canaries

Fault injection (chaos engineering 😊 )





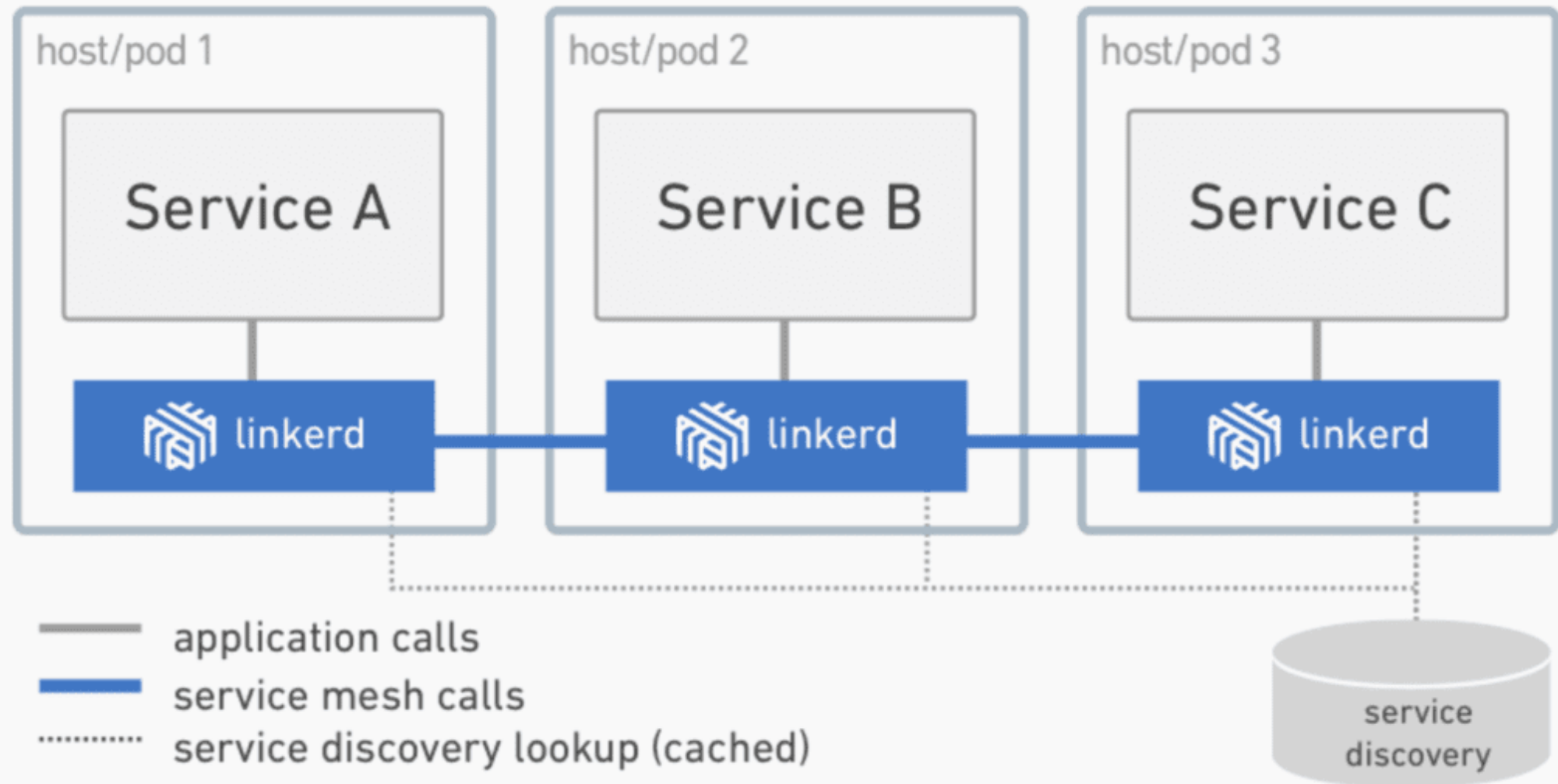


# 3 Simple steps

---

1. Installing the CLI on your local system;
2. Installing the control plane into your cluster;
3. Adding your services to Linkerd's data plane.

*Linkerd instances form a service mesh, allowing application code to communicate reliably.*



---

Open proxy to dashboard:

```
linkerd dashboard  
expose linkerd viz plugin
```

Show top processes

```
linkerd viz top -n emoji voto deployment/web
```

Inject Linkerd into existing deployment

```
kubectl get -n emoji voto deploy -o yaml \  
| linkerd inject - \  
| kubectl apply -f -
```

# Splitting traffic in Linkerd

---

apiVersion: split.smi-spec.io/v1alpha1

kind: TrafficSplit

metadata:

name: error-split

namespace: booksapp

spec:

service: books

backends:

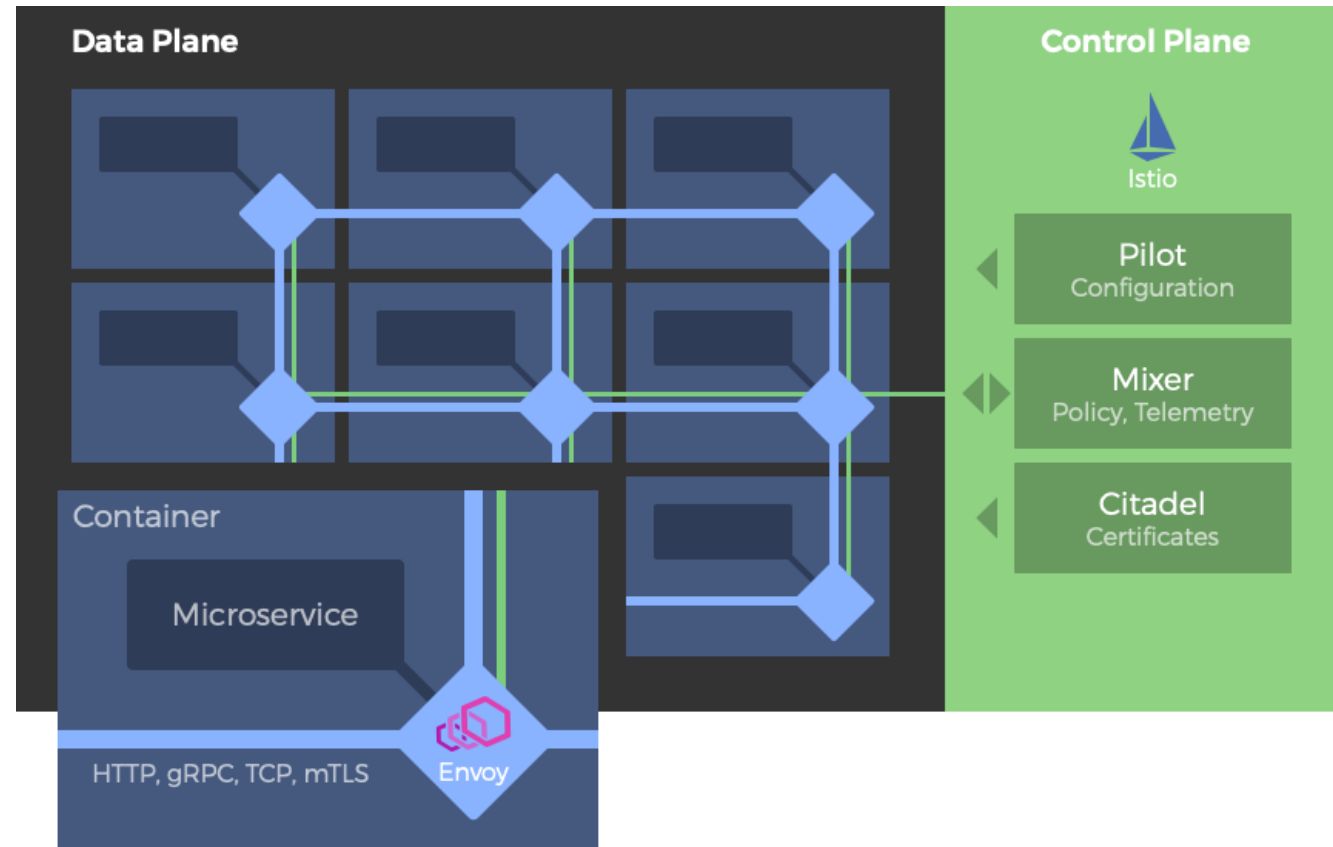
- service: books

weight: 900m

- service: error-injector

weight: 100m

# Istio



# Istio

---

## **Traffic management**

Controls the flow of traffic and application program interface (API) calls between services. Makes API calls more reliable. This includes circuit breaker, error injection, traffic splitting, timeouts and request mirroring for disaster recovery.

## **Observability**

Provides insights on performance. A dashboard offers visibility to quickly identify issues. This includes application mapping, app logging, and tracing.

## **Policy enforcement**

Ensuring policies are enforced and allowing for policy changes without changing application code.

## **Security**

Secure service communications allow for consistent enforcement of policies consistently across all protocols. These include authentication, authorization, rate limiting and a [distributed web application firewall](#) for both ingress and egress.

# K-native

---

Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications

Knative components build on top of Kubernetes, abstracting away the complex details and enabling developers to focus on what matters.

Built by codifying the best practices shared by successful real-world implementations, Knative solves the "boring but difficult" parts of deploying and managing cloud native services so you don't have to.



# K-native features

---

## Serving

- Run serverless containers on Kubernetes with ease, Knative takes care of the details of networking, autoscaling (even to zero), and revision tracking. You just have to focus on your core logic.

## Eventing

- Universal subscription, delivery, and management of events. Build modern apps by attaching compute to a data stream with declarative event connectivity and developer-friendly object model.



```
kubectl apply --selector knative.dev/crd-install=true \  
  --filename https://github.com/knative/serving/releases/download/v0.8.0/serving.yaml \  
  --filename https://github.com/knative/eventing/releases/download/v0.8.0/release.yaml \  
  --filename https://github.com/knative/serving/releases/download/v0.8.0/monitoring.yaml
```

```
kubectl apply --filename  
https://github.com/knative/serving/releases/download/v0.8.0/serving.yaml \  
  --filename https://github.com/knative/eventing/releases/download/v0.8.0/release.yaml \  
  --filename https://github.com/knative/serving/releases/download/v0.8.0/monitoring.yaml
```

# Simple to integrate

---

Only get portnumber from env-variable and listen for incoming traffic

Build Docker image and push

Rest is abstracted

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args)
{
    string port = Environment.GetEnvironmentVariable("PORT") ?? "8080";
    string url = String.Concat("http://0.0.0.0:", port);

    return WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>().UseUrls(url);
}
```

# Create k-native service definition

---

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello
spec:
  template:
    spec:
      containers:
      - image: gcr.io/knative-samples/helloworld-go
        ports:
        - containerPort: 8080
      env:
      - name: TARGET
        value: "World"
```

# Knative

```
kubectl get ksvc
```

NAME	URL	LATESTCREATED	LATESTREADY	READY
hello	http://hello.default.\${LOADBALANCER_IP}.sslip.io	hello-00001	hello-00001	True

```
kubectl get pod -l serving.knative.dev/service=hello -w
```

NAME	READY	STATUS
hello-world	2/2	Running
hello-world	2/2	Terminating
hello-world	1/2	Terminating
hello-world	0/2	Terminating

# Consul Connect

---

Consul Connect is an extension of Consul

Distributed service discovery

key-value store

HA

# Consul Connect

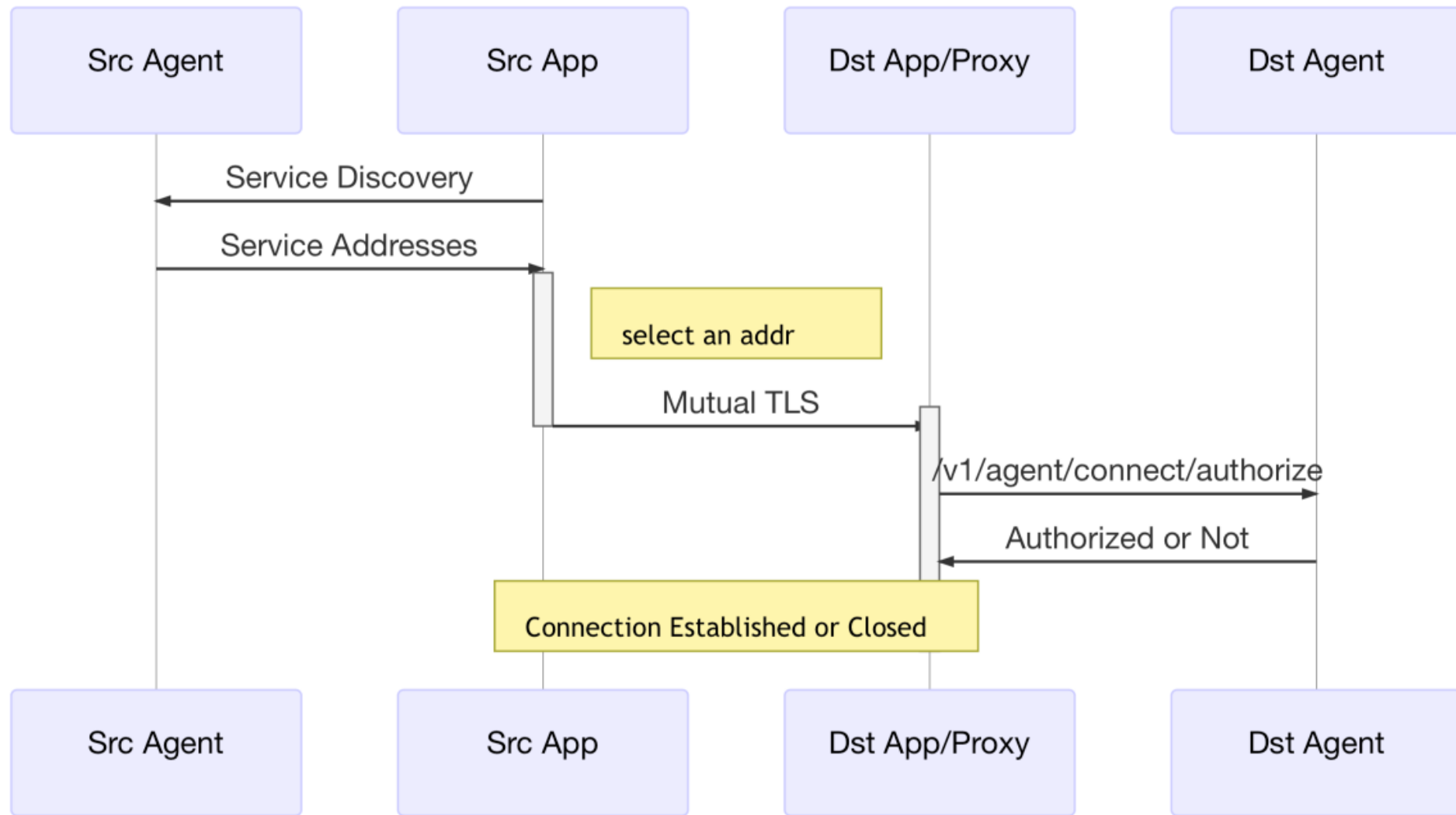
---

Consul Connect provides service-to-service connection authorization and encryption using mutual Transport Layer Security (TLS).

Applications can use sidecar proxies in a service mesh configuration to automatically establish TLS connections for inbound and outbound connections without being aware of Connect at all.

Applications may also natively integrate with Connect for optimal performance and security. Connect can help you secure your services and provide data about service-to-service communications.







```
apiVersion: v1
kind: Pod
metadata:
  name: static-server
  annotations:
    "consul.hashicorp.com/connect-inject": "true"
spec:
  containers:
    - name: static-server
      image: hashicorp/http-echo:latest
      args:
        - -text="hello world"
        - -listen=:8080
      ports:
        - containerPort: 8080
          name: http
```

# Dapr Components

