Service Mesh

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.

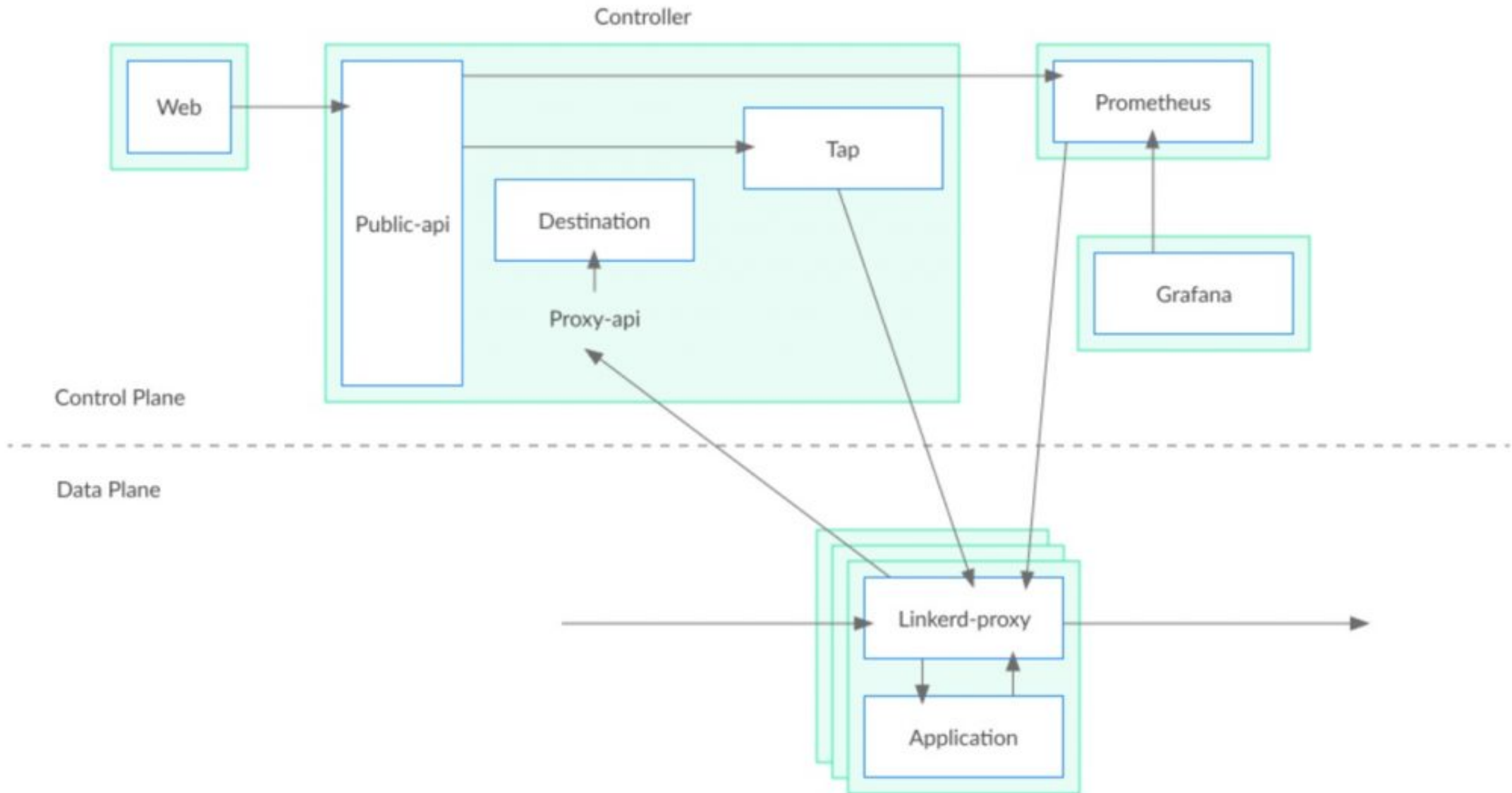
Popular versions

- Linkerd
- Istio
- Consul

What is Linkerd?

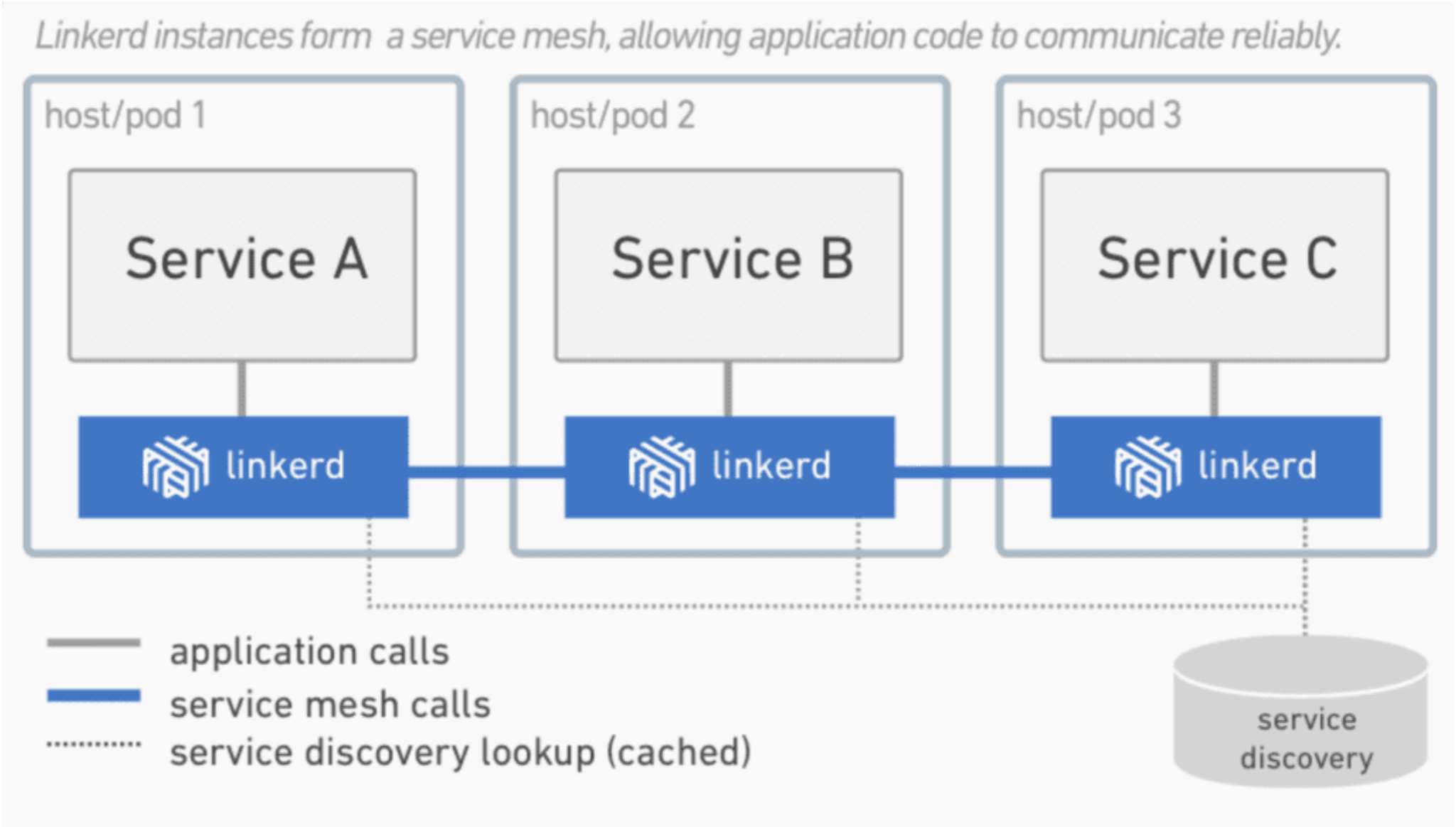
- Linkerd is a *service mesh* for Kubernetes.
  It makes running services easier and safer by giving you runtime debugging, observability, reliability, and security—all without requiring any changes to your code.
- It adds:
  - monitoring
  - circuit-breaking,
  - latency-aware load balancing,
  - eventually 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 ☺)


Controller
Web
Public-api
Tap
Destination
Proxy-api
Prometheus
Grafana
Control Plane
Data Plane
Linkerd-proxy
Application

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.
host/pod 1
host/pod 2
host/pod 3
Service A
Service B
Service C
linkerd
linkerd
linkerd
application calls
service mesh calls
service discovery lookup (cached)
service discovery

- Open proxy to dashboard:

```
linkerd dashboard
```

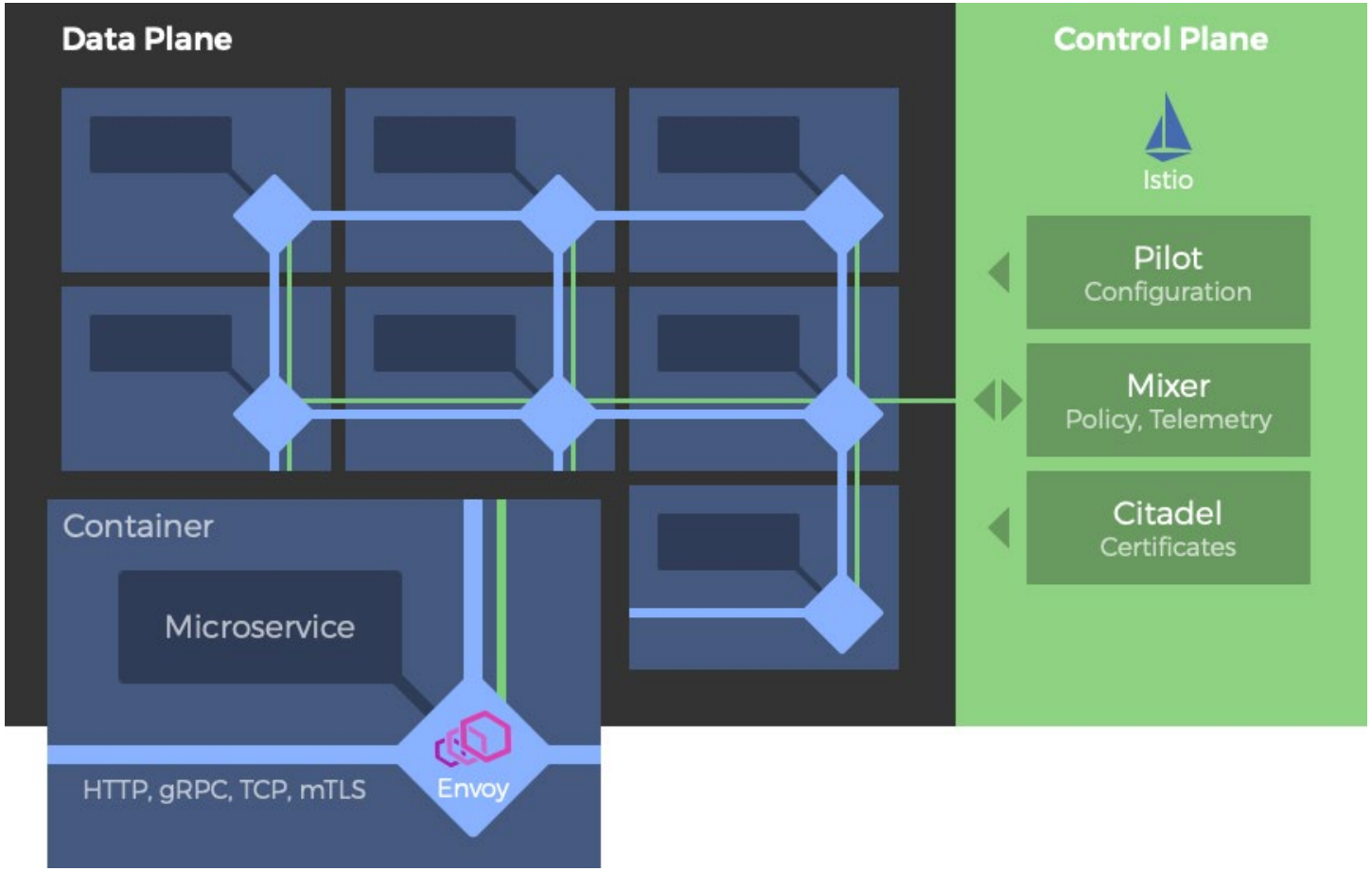
- Show top processes

```
linkerd -n linkerd top deploy/linkerd-web
```

- Inject Linkerd into existing deployment

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

Istio


Data Plane
Control Plane
Istio
Pilot
Configuration
Mixer
Policy, Telemetry
Citadel
Certificates
Container
Microservice
HTTP, gRPC, TCP, mTLS
Envoy

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 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.


Users and Systems (IoT) use applications developed and deployed by developers and hosted by operators
Istio
Knative
Kubernetes
Developers build and deploy apps
API
GitHub
Contributors develop and contribute code and docs to the OSS project
Operators deploy and manage Knative instances using Kubernetes API and tools
Platform Providers (such as Google Cloud Platform) provide underlying infrastructure

Components

- Knative consists of the Serving and Eventing components:
  - Eventing - Management and delivery of events
  - Serving - Request-driven compute that can scale to zero

```
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 def.

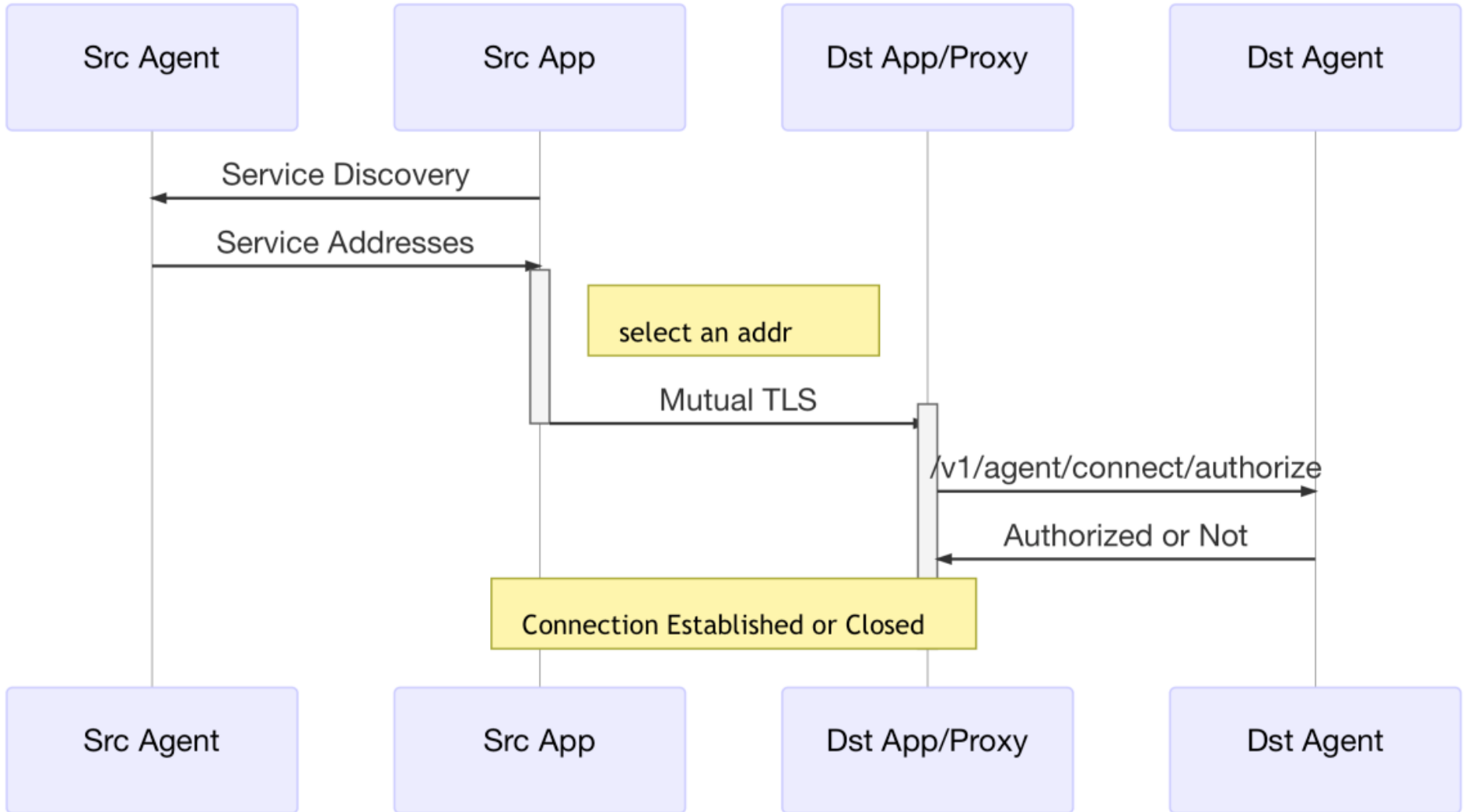
```
apiVersion: serving.knative.dev/v1alpha1
   kind: Service
   metadata:
     name: helloworld-csharp
     namespace: default
   spec:
     template:
       spec:
         containers:
           - image: docker.io/{username}/helloworld-csharp
             env:
               - name: TARGET
                 value: "C# Sample v1"
```

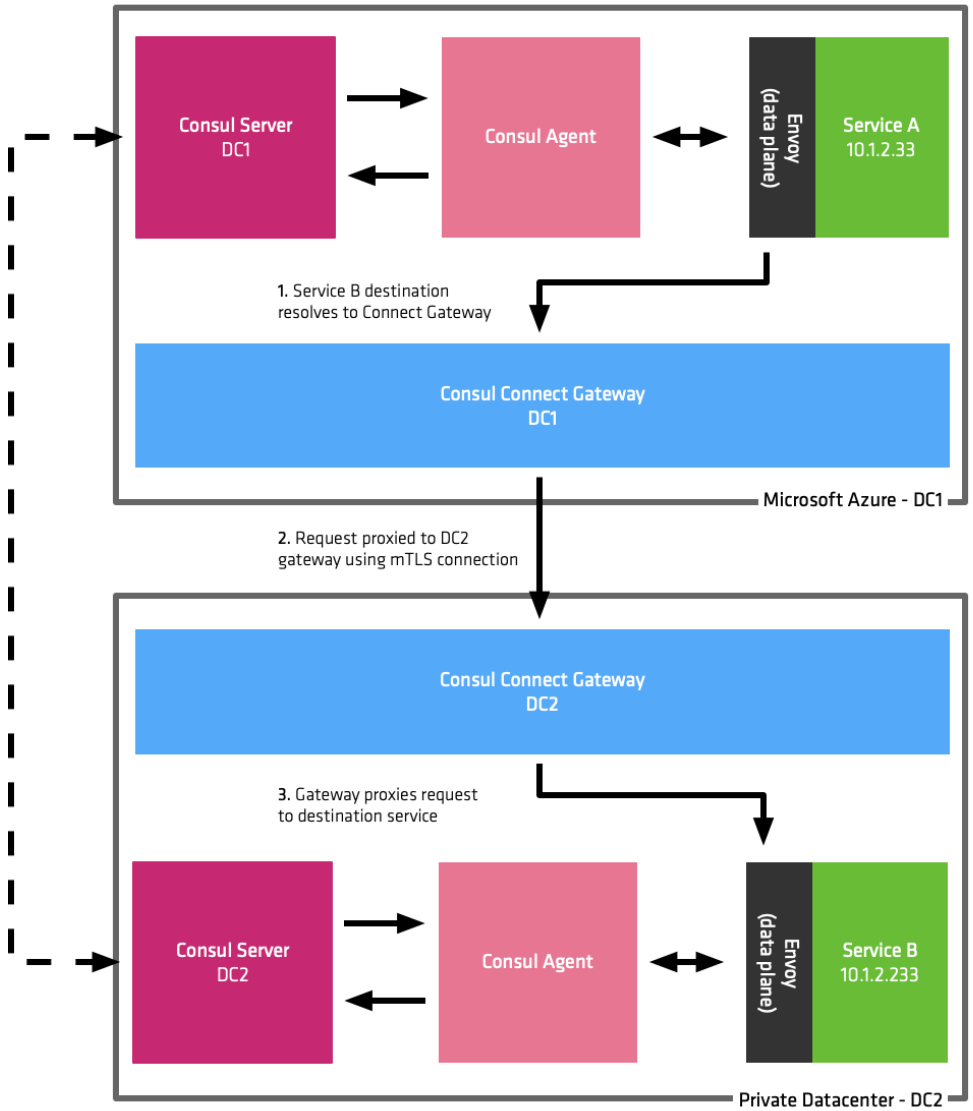
Consul Connect

- 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.


Src Agent
Src App
Dst App/Proxy
Dst Agent
Service Discovery
Service Addresses
select an addr
Mutual TLS
/v1/agent/connect/authorize
Authorized or Not
Connection Established or Closed
Src Agent
Src App
Dst App/Proxy
Dst Agent


Consul Server
DC1
Consul Agent
Envoy
(data plane)
Service A
10.1.2.33
1. Service B destination
resolves to Connect Gateway
Consul Connect Gateway
DC1
Microsoft Azure - DC1
2. Request proxied to DC2
gateway using mTLS connection
Consul Connect Gateway
DC2
3. Gateway proxies request
to destination service
Consul Server
DC2
Consul Agent
Envoy
(data plane)
Service B
10.1.2.233
Private Datacenter - DC2

```
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
```