When monoliths applications are broken down into microservices or a new microservices architecture is being started, first problem we see that now there are more components to manage, deploy and all of them should be able interact with each other. Moreover, there are multiple instances of services run to make services highly available.

**Common problems with microservices**

1. Deployment
2. Resource Isolation & Utilization
3. Resiliency
4. Networking

There are different container orchestration tools which makes the deployment easy. Like

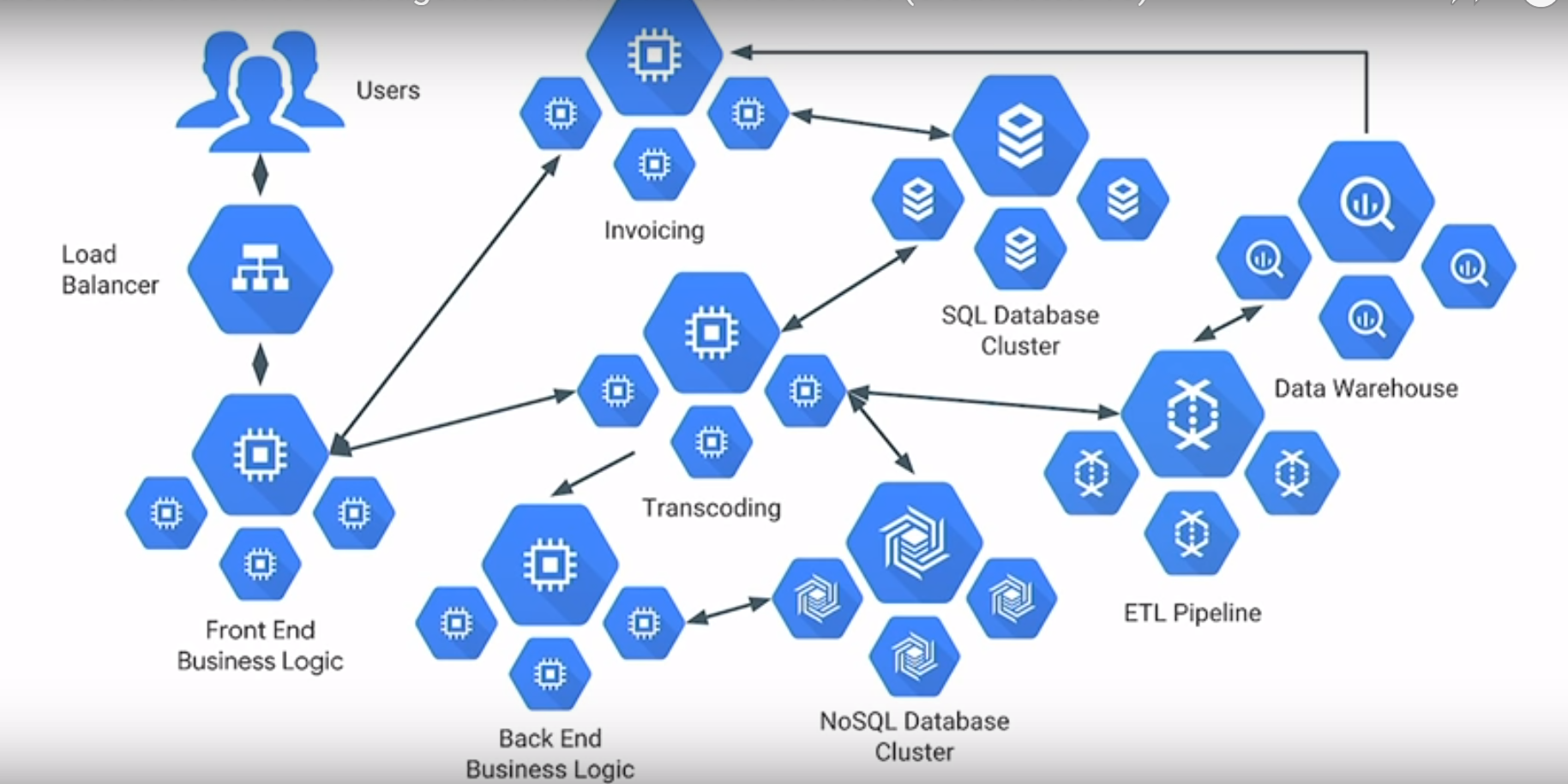
* Kubernetes
* AWS Elastic Container Service (ECS)
* Docker Swarm

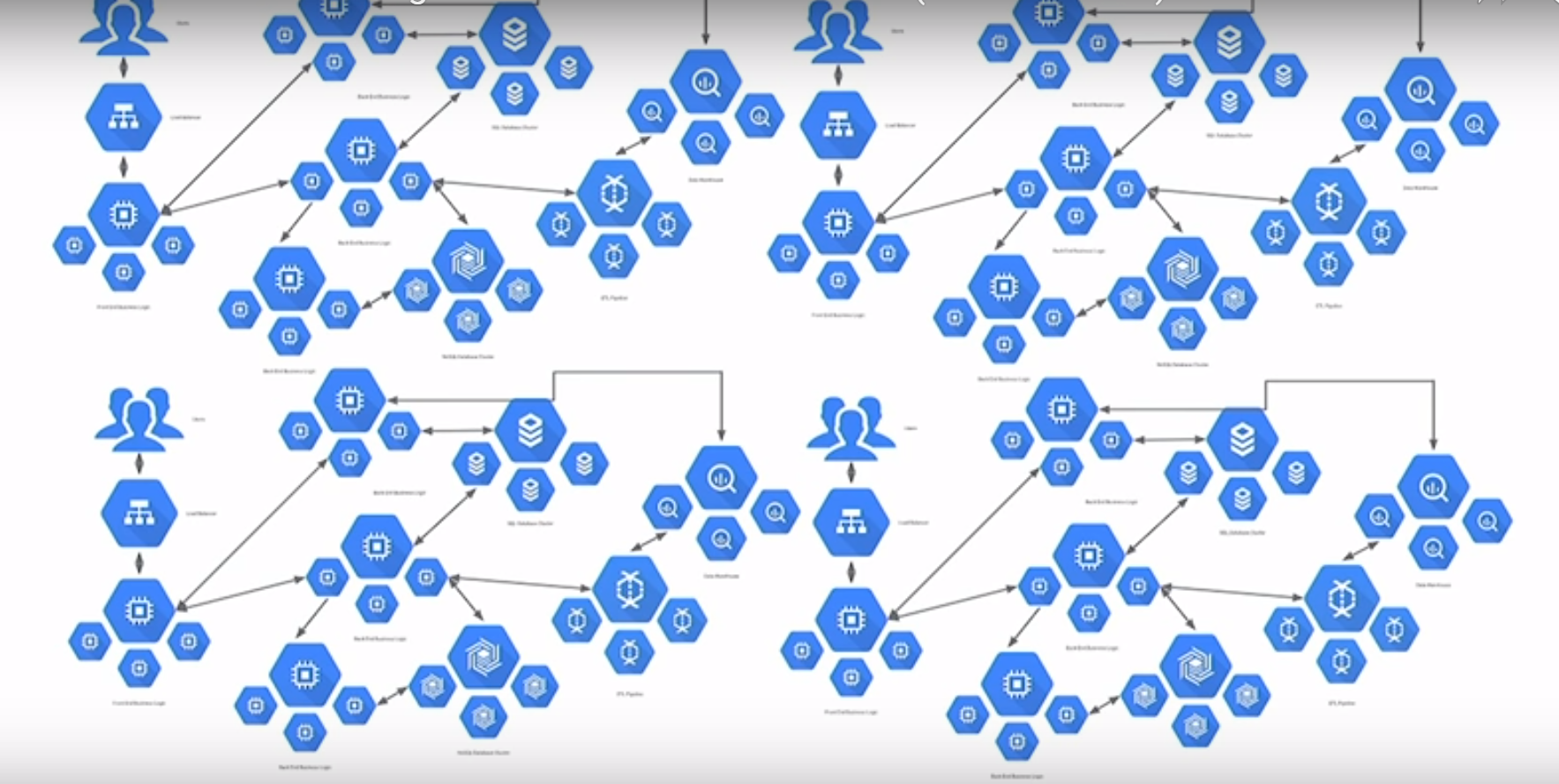
So, the key of container orchestration tool is

* Control Plane
* Cluster of machines as one Well-defined API
* Abstraction of Infrastructure

**Microservices are beyond Development or Deployment**

As the number of services increase, or you scale the microservices to multiple region, you have to deal with the interactions between them, monitor the overall system health, be fault tolerant, have logging and telemetry in place, handle multiple points of failure and more. Microservices add network and infra complexity to your system. Microservices Architecture/Application is beyond just development or deployment.





Most of the Microservices application faces these issues.

* Request Routing
* Load Balancing
* Fault Tolerance
* Observability & Insight
* Monitoring & Tracing
* Circuit Breaking
* Authentication
* Service Discovery (Interaction between microservices)

Performing these tasks has gotten easier in Java since the launch of NetflixOSS in 2011. Netflix OSS stack (Spring Cloud) provides

* Eureka - Service Registry/Discovery
* Ribbon - Client Side LB
* Zuul – API Gateway
* Hystrix - Circuit Breaker
* Zipkin - Distributed Tracing
* Prometheus - Monitoring
* Grafana - Data Visualization



**Issues with Architecture**

* **Cross cutting concerns** - But what happens with these tools is that rather than building the microservices we have potentially more problem to address. We need to make sure that each of them is HA highly available and reliable.
* **Tightly coupled** – These components need to be configured inside your application code and based on the language you are using; the implementation will vary a bit.
* **Complex Implementation –** This is easy when working with single stack such as **Framework w/ Spring Boot**. This becomes more difficult or harder with
  + **Multiple Stack**
  + **Multiple Frameworks**
  + **Polyglot** (For example, one would need to find/build libraries for Node.js to participate in service discovery and support observability)
  + **Legacy**

But what is the main idea of micro services, let microservice A talk to microservice B. All of these we shouldn’t be concerned. Ideally it should be provided by the platform.

Service Mesh comes to the rescue here. It decouples this complexity from your application and puts it in a service proxy & let it handle it for you

**Istio** (Open Source Service Mesh for Microservices)

A complete framework for connecting, securing, managing and monitoring microservices. Istio provides an easy way to create a network of deployed services with load balancing, service-to-service authentication, monitoring, and more, without requiring any changes in service code.

It is a completely open source service mesh that layers transparently onto existing distributed applications and lets you efficiently, run a distributed microservice architecture, and provides a uniform way to secure, connect, and monitor microservices

Istio was first announced in 2017 with key contributions from Google, IBM, Lyft and others.

Multi-environment and multiplatform but Kubernetes first.

One of the key benefits of Istio is that it can be launched ‘on top’ of an existing application — it deploys an Envoy proxy-server for each service as a sidecar-container inside the same Pod. It means you don’t have to make any change to the code of your applications.

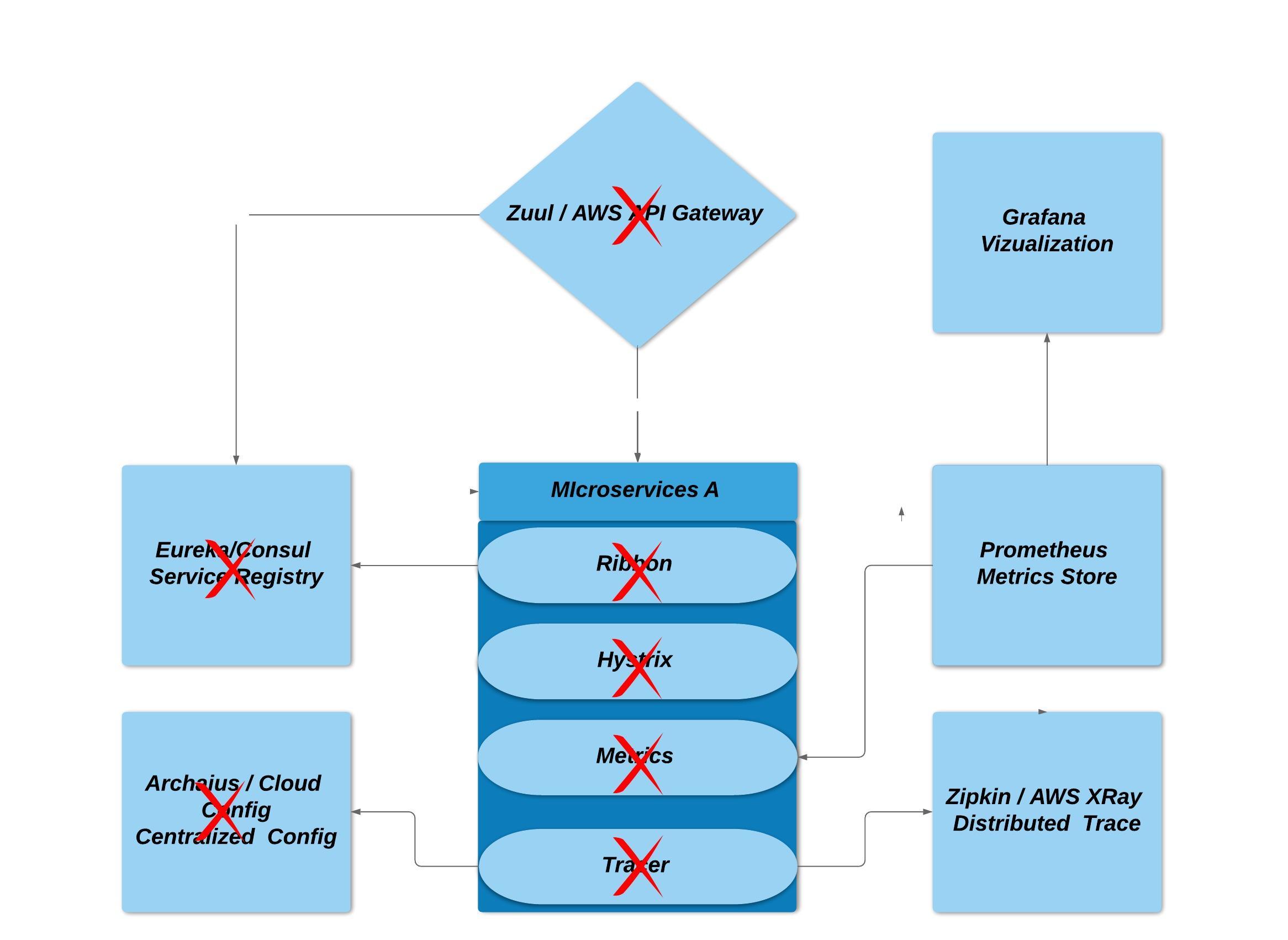
**Service Mesh**

The network of microservices that make up such applications and the interactions between them. As a service mesh grows in size and complexity, it can become harder to understand and manage.

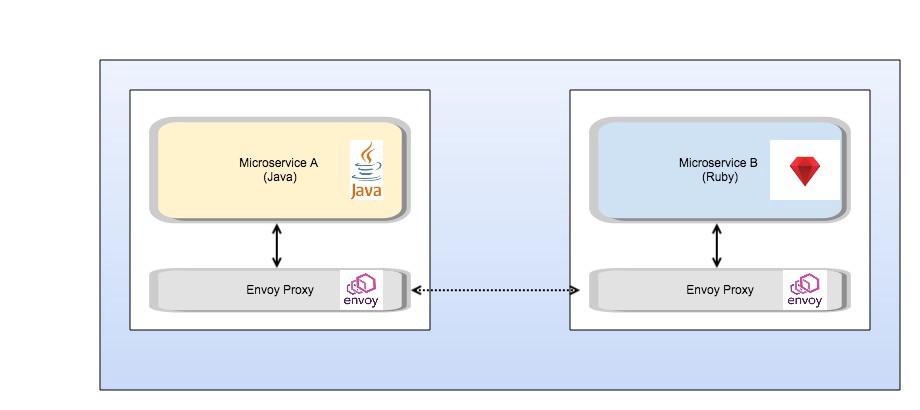
**What does it do**

* Service to service communication
* Routing rules
* Retries
* Circuit Breaker
* Performance Monitoring
* Tracing
* Authentication

We can get rid of



Take all of these cross-cutting concerns and move it to them in proxy/sidecar

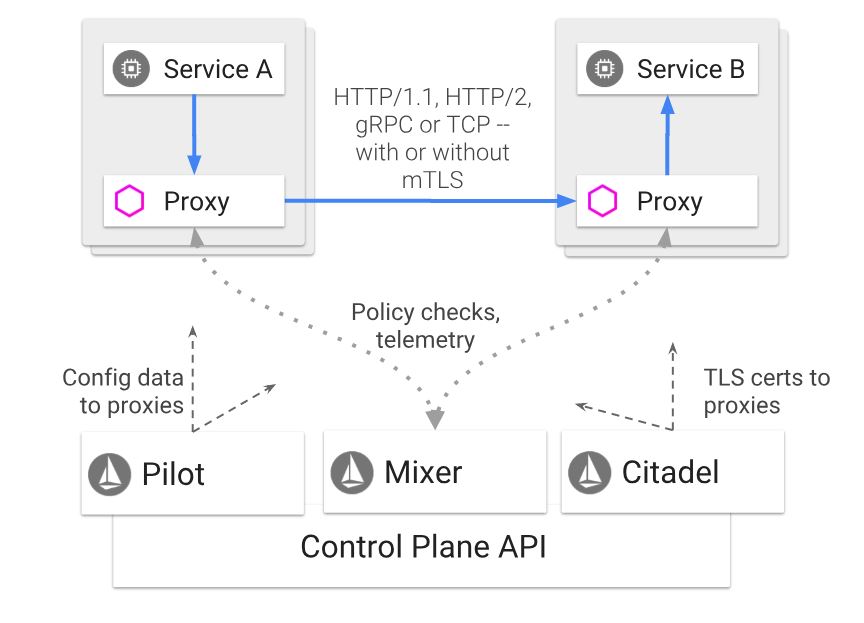


**Architecture**

Istio can be split into two parts

**Data Plane** – It consist of proxies (Envoy) deployed as sidecars. These proxies intercept all the communication between the microservices.

**Control Plane** – It manages and configures the proxies to route traffic. Also, using the Mixer to enforce policies and collect telemetry.



**Envoy** - It is a high-performance proxy developed in C++ by Lyft to mediate all inbound and outbound traffic for all services in the service mesh. It is deployed as a sidecar to the relevant service in the same Kubernetes pod.

* Dynamic service discovery
* Load balancing
* TLS termination
* HTTP/2 and gRPC proxies
* Circuit breakers
* Health checks
* Staged rollouts with %-based traffic split
* Fault injection
* Rich metrics

**Mixer** – It enforces access control and usage policies across the service mesh, and collects telemetry data from the Envoy proxy and other services. The plugin model enables Istio to interface with a variety of host environments and infrastructure backends.

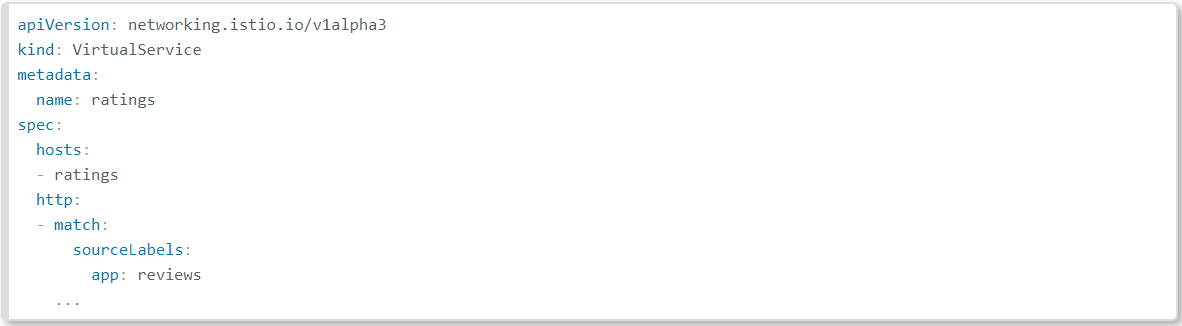
**Pilot** – It provides service discovery for the Envoy sidecars, traffic management capabilities for intelligent routing, and resiliency. It converts high level routing rules and configurations, and send it to proxies at runtime.

**Citadel** – It provides strong service-to-service and end-user authentication using Mutual TLS and JWT. It can be used to encrypt the traffic between the services.

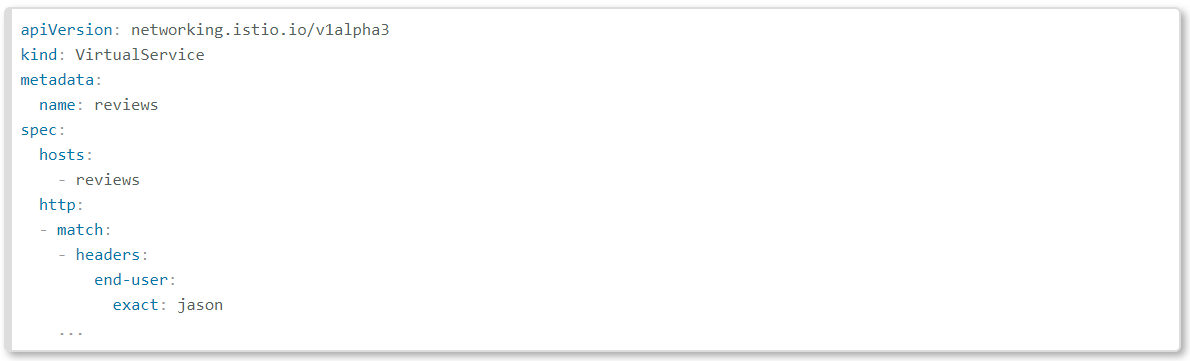
**Splitting traffic between versions based on weight**



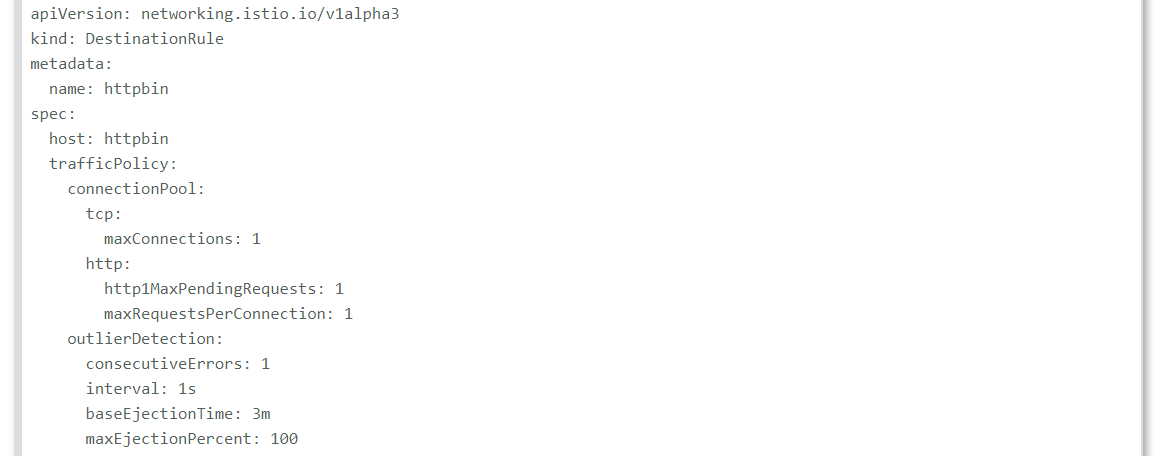
**Route traffic using workload labels**



**Route Traffic based on HTTP headers**



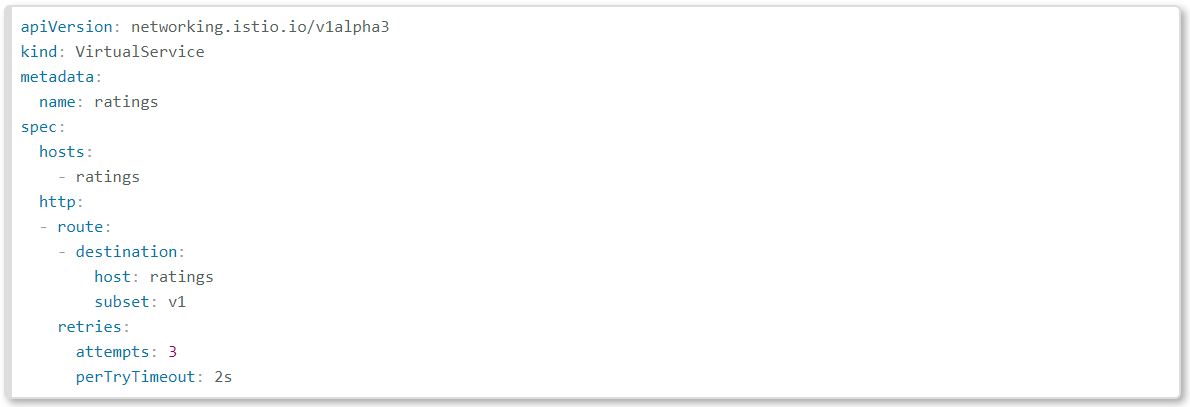
**Circuit Breaking**



**Timeout**

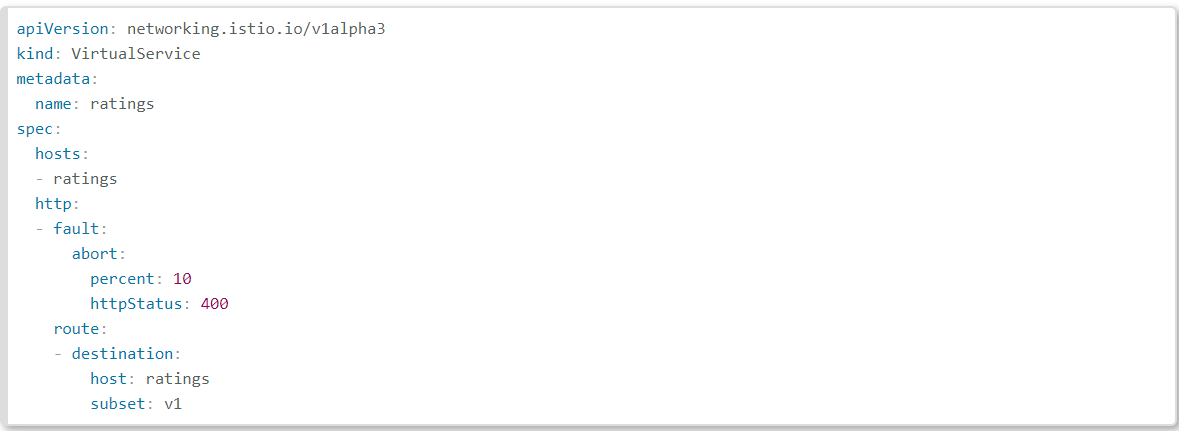


**Retries**



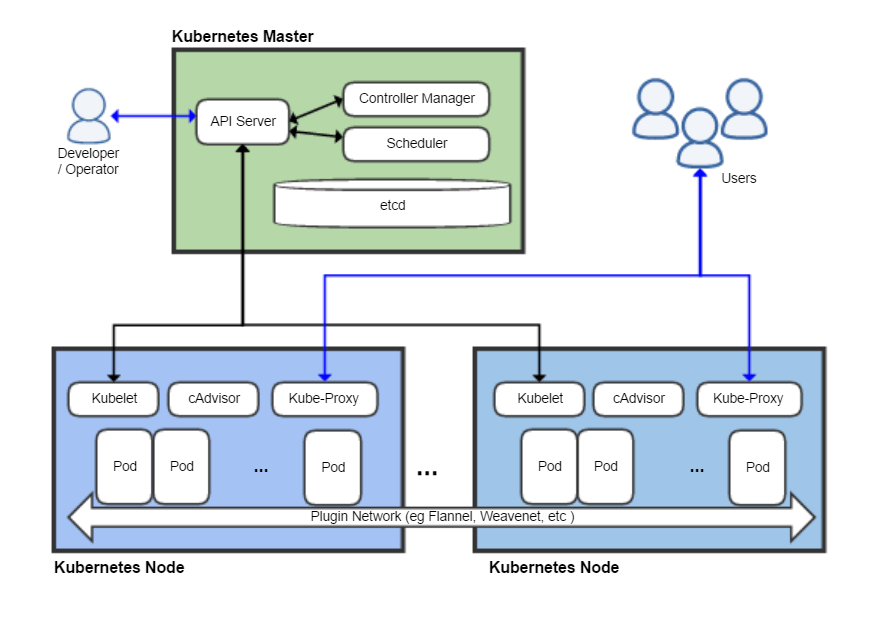
**Fault Injection**





**Kubernetes**

Kubernetes (commonly known as K8s) is an open-source container-orchestration system for automating deployment, scaling and management of containerized applications. It was originally designed by Google and was released in 2015, and is now maintained by the Cloud Native Computing Foundation.



**Architecture**

Kubernetes follows the master-slave architecture. The components of Kubernetes can be divided into those that manage the **worker node** and those that are part of the **control plane**.

**Master (Kubernetes control plane)**

Master components provide the cluster’s control plane. It manages workload, directing communication across the system, make global decisions about the cluster (for example, scheduling), and detecting and responding to cluster events (starting up a new pod when a replication controller’s ‘replicas’ field is unsatisfied).

**kube-apiserver**

It is the front-end for the Kubernetes control plane which exposes the Kubernetes API using JSON over HTTP, which provides both the internal and external interface to Kubernetes.

**Etcd**

It is a persistent, lightweight, distributed, key-value data store developed by CoreOS that reliably stores the configuration data of the cluster, representing the overall state of the cluster at any given point of time.

**kube-scheduler**

It watches newly created pods that have no node assigned and selects a node for them to run on.

Factors considered for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

**kube-controller-manager**

Component on the master that runs controllers. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process. For example : Replication Controller, Node Controller etc.

**cloud-controller-manager**

It runs controllers that interact with the underlying cloud providers.

**Kubernetes node (slave)**

The Node, also known as **Worker** or Minion, is a machine where containers are deployed. Every node in the cluster must run a container runtime such as **Docker**, as well as the below-mentioned components, for communication with master for network configuration of these containers.

**Kubelet**

An agent that runs on each node in the cluster. It makes sure that containers are running in a pod. It ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn’t manage containers which were not created by Kubernetes.

**kube-proxy**

It is an implementation of a network proxy and a load balancer, and it supports the service abstraction along with other networking operation. It is responsible for routing traffic to the appropriate container based on IP and port number of the incoming request.

**Container Runtime**

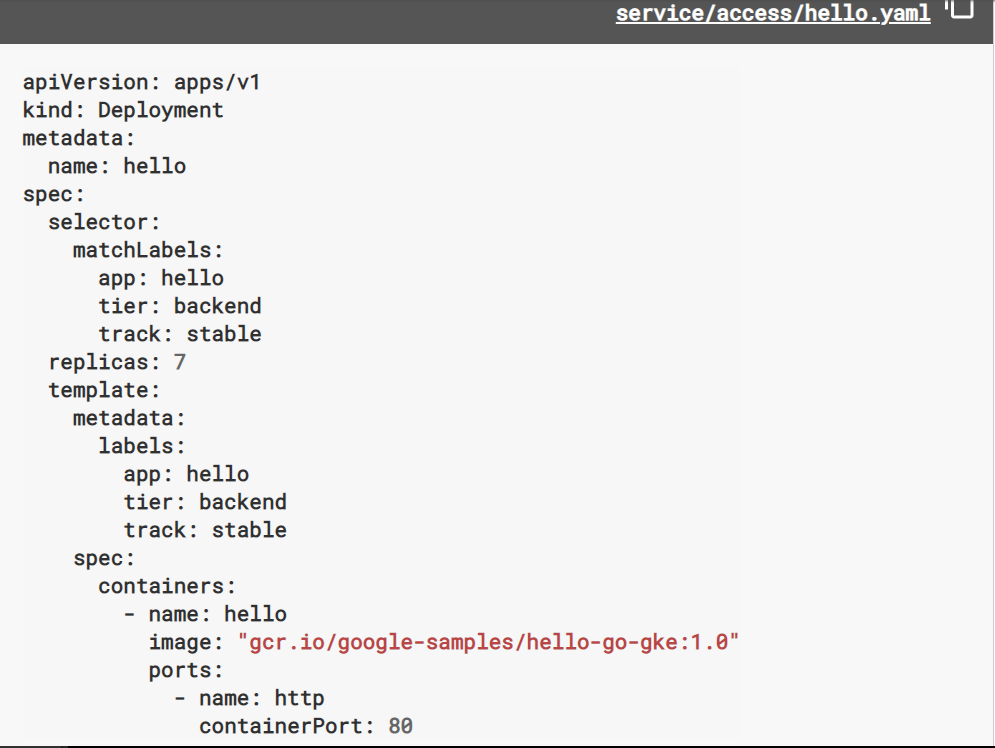
The container runtime is the software that is responsible for running containers. Kubernetes supports several runtimes: Docker, rkt, runc and any OCI runtime-spec implementation.

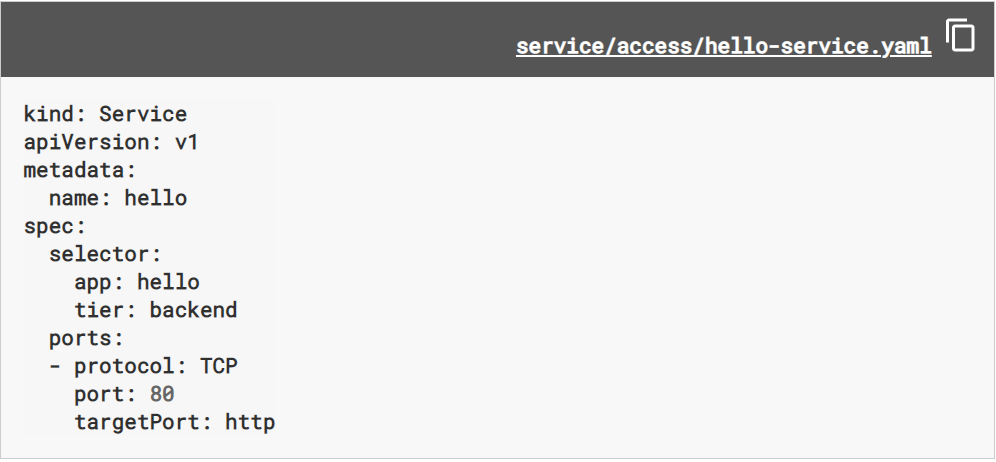
**cAdvisor**

It is an agent that monitors and gathers resource usage and performance metrics such as CPU, memory, file and network usage of containers on each node.

**Kubernetes Features/Advantages**

* **Automated Scheduling**: Kubernetes provides advanced scheduler to launch container on cluster nodes.
* **Self-Healing Capabilities**: Rescheduling, replacing and restarting the containers which are died. Containers are ephemeral and they can die.
* **Automated Rollouts and Rollbacks**: For the big ecosystem, it’s necessary to roll out features with maximum availability. And if something goes wrong it’s always good to go back to the previous state. Kubernetes supports rollouts and rollbacks for the desired state of the containerized application.
* **Horizontal Scaling**: Kubernetes can scale up and scale down the application as per the requirements. This scaling can be automated with the CPU usage.
* **Service Discovery and load balancing**: Kubernetes uses unique IP addresses and DNS name to containers. It helps to identify them uniquely across the cluster.





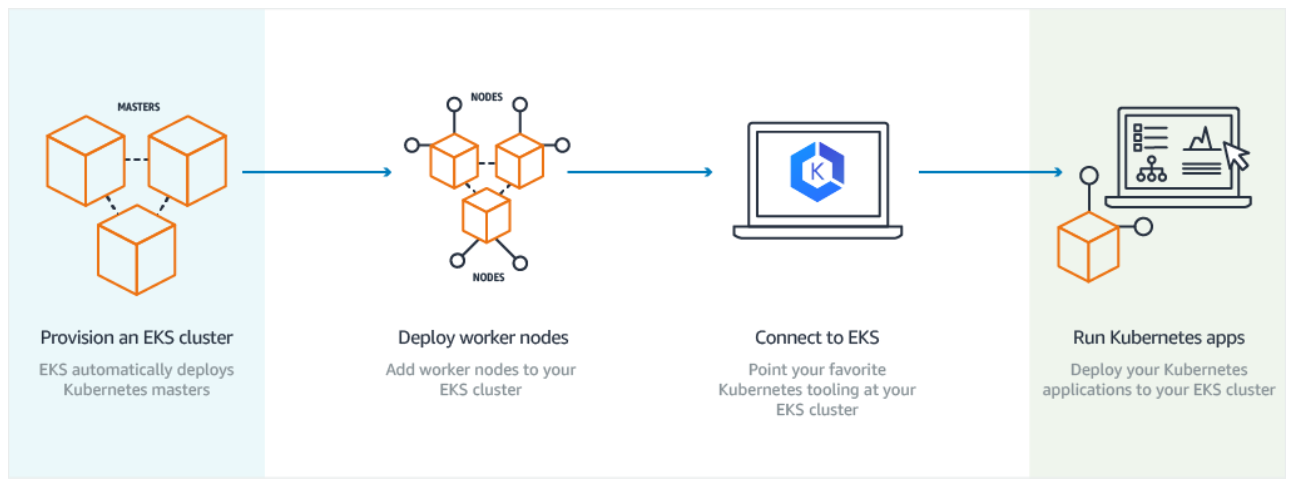
**kubectl create -f https://k8s.io/examples/service/access/hello.yaml**

**kubectl create -f https://k8s.io/examples/service/access/hello-service.yaml**

**Amazon Elastic Container Service for Kubernetes (Amazon EKS)**

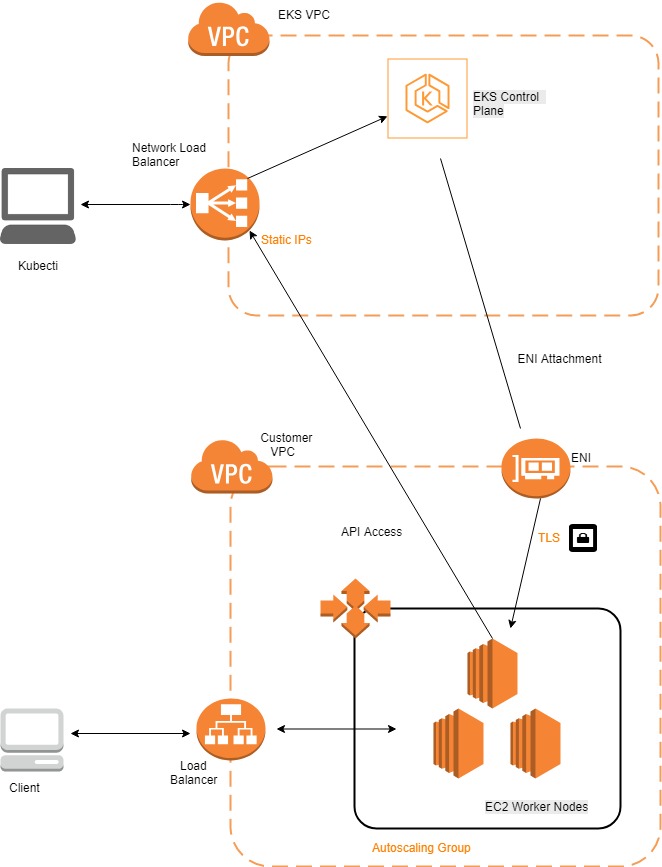
Amazon Elastic Container Service for Kubernetes (EKS) is a managed Kubernetes service that makes it easy for you to run Kubernetes on AWS without needing to install, operate, and maintain your own Kubernetes control plane (Master Node).

* Amazon EKS runs Kubernetes control plane instances across multiple Availability Zones to ensure high availability.
* Amazon EKS automatically detects and replaces unhealthy control plane instances.
* It provides automated version upgrades and patching for them.
* Applications running on any standard Kubernetes environment are fully compatible and can be easily migrated to Amazon EKS.
* Amazon EKS is generally available for all AWS customers.



**Amazon EKS Features**

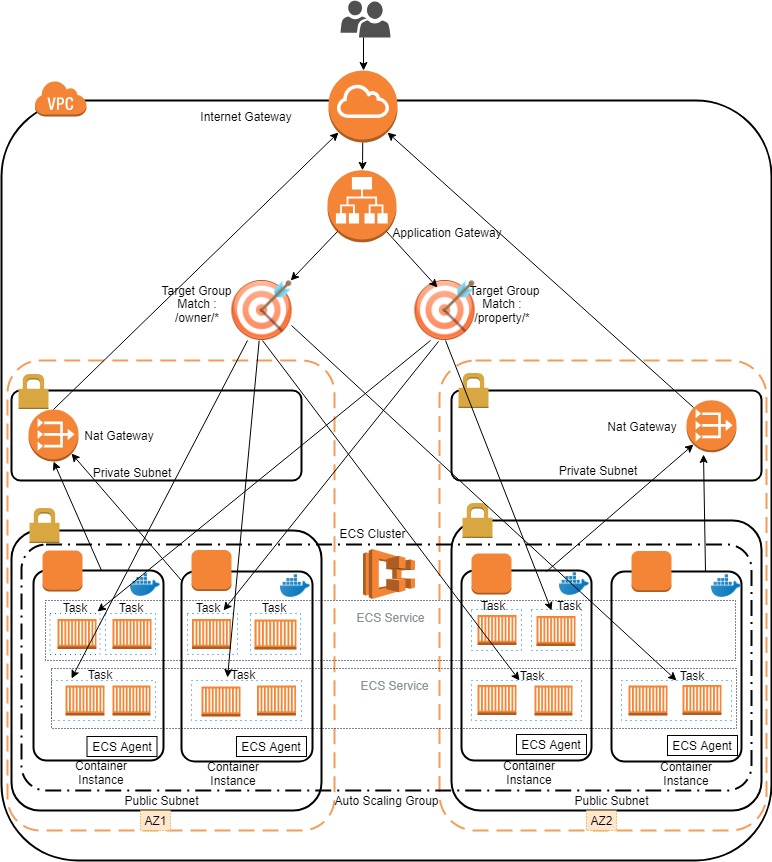
* **Managed Kubernetes Control Plane** - EKS provides a scalable and highly-available control plane that runs across multiple AWS availability zones.
* **Networking and Security** - EKS makes it easy to provide security for your Kubernetes clusters, with advanced features and integrations to AWS services and technology partner solutions. For example, IAM provides fine-grained access control and Amazon VPC isolates your Kubernetes clusters from other customers
* **VPC Support** - EKS clusters run in an Amazon VPC, allowing you to use your own VPC security groups and network ACLs. No compute resources are shared with other customers.
* **IAM Authentication** - EKS integrates Kubernetes RBAC with IAM authentication through a collaboration with **Heptio**. You can assign RBAC roles directly to each IAM entity allowing you to granularly control access permissions to your Kubernetes masters.
* **Load Balancing** - EKS supports using Elastic Load Balancing including Application Load Balancer (ALB), Network Load Balancer (NLB), and Classic Load Balancer.
* **Logging** - EKS is integrated with AWS CloudTrail to provide visibility and audit history of your cluster and user activity.
* **Certified Conformant** - Amazon EKS runs upstream Kubernetes and is certified Kubernetes conformant
* **Support for Advanced Workloads** – It provides an optimized Amazon Machine Image (AMI) that includes configured NVIDIA drivers for GPU-enabled P2 and P3 EC2 instances.
* **Works with Community Tools** - It is fully compatible with Kubernetes community tools and supports popular Kubernetes add-ons



**ECS (Amazon Elastic Container Service)**

Amazon Elastic Container Service (Amazon ECS) is a highly scalable, high-performance container orchestration service that supports Docker containers and allows you to easily run and scale containerized applications on AWS. Amazon ECS eliminates the need for you to install and operate your own container orchestration software, manage and scale a cluster of virtual machines, or schedule containers on those virtual machines.

With simple API calls, you can launch and stop Docker-enabled applications, query the complete state of your application, and access many familiar features such as IAM roles, security groups, load balancers, Amazon CloudWatch Events, AWS CloudFormation templates, and AWS CloudTrail logs.



**Features of ECS**

* Easy management of clusters
* Flexibility in container placement
* Can be used with other AWS services
* Extensible
* Performance delivered even after scaling up
* Secure
* Programmatic control
* Load balance
* Monitoring
* Logging
* Security

Logging

Secrets

Istio Authentication/Security

ECS VS EKS

|  |  |  |
| --- | --- | --- |
|  | ECS | EKS |
| Load Balancer | It provides an integration with the Application Load Balancer (ALB), the Network Load Balancer (NLB) as well as the Classic Load Balancer (CLB) | It supports the Network Load Balancer(NLB) and the Classic Load Balancer(CLB). Application Load Balancer is available from version 1.0.0  https://aws.amazon.com/about-aws/whats-new/2018/11/amazon-eks-adds-alb-support-with-aws-alb-ingress-controller/ |
| Elastic Network Interface (ENI) | In ECS as each task - a group of containers - is assigned to a separate ENI. Maximum of 15 tasks you can place per instance with ECS | In EKS some pods share the network interface with each other. As EKS is sharing ENIs between pods, you can place up to 750 pods per instance. |
| IAM | ECS supports IAM Roles for Tasks which is great to grant containers access to AWS resources. For example, to allow containers to access S3, DynamoDB, SQS, or SES at runtime. | Unfortunately, EKS does not support IAM for pods out-of-the-box at the moment. |
| Pricing | ECS is free. For both, EKS and ECS you have to pay for the underlying EC2 instances and related resources. | Each EKS cluster costs you 0.20 USD per hour which is about 144 USD per month |
| Compatibility | ECS is only available on AWS | EKS is an option at other cloud providers, on-premises, or even on your developer machine |

|  |  |
| --- | --- |
| Advantages of Kubernetes Over Amazon ECS | |
| * Can be deployed on-premises, private clouds, and public clouds. (anywhere you can run x86 servers, or even on your laptops) * Wide variety of storage options, including on-premises SANs and public clouds. * Based on extensive experience running Linux containers at Google. Deployed at scale more often among organizations. Kubernetes is also backed by enterprise offerings from both Google (GKE) and RedHat (OpenShift). * Largest community among container orchestration tools. Over 50,000 commits and 1200 contributors. | * Vendor lock-in. Containers can only be deployed on Amazon, and ECS can only manage containers that it has created. * External storage is limited to Amazon, including Amazon EBS. * Validated within Amazon. ECS is not publicly available for deployment outside Amazon. * Much of ECS code is not publicly available. Parts of ECS, including [Blox](https://github.com/blox/blox/tree/master/cluster-state-service), a framework that helps customers build custom schedulers, are open source. 200 commits and 15 contributors, many from Amazon. |

|  |  |
| --- | --- |
| https://platform9.com/wp-content/uploads/2017/06/kubernetes-4.png  Disadvantages of Kubernetes https://platform9.com/wp-content/uploads/2017/07/amazon-ecs.png | |
| * Lack of single vendor control can complicate a prospective customer’s purchasing decision. Community includes Google, Red Hat, and over 2000 authors. (Source: [CNCF)](https://www.cncf.io/blog/2017/06/05/30-highest-velocity-open-source-projects/) * Do-it-yourself installation can be complex for Kubernetes. Deployment tools for Kubernetes include kubeadm, kops, kargo, and others. | * Single vendor control may allow for accountability with bug fixes and better coordination with feature development. * ECS does not require installation on servers. [ECS CLI installation](http://docs.aws.amazon.com/AmazonECS/latest/developerguide/ECS_CLI_installation.html) is simple. |

Refer : <https://cloudonaut.io/eks-vs-ecs-orchestrating-containers-on-aws/>

<https://medium.com/devopslinks/ecs-vs-eks-vs-fargate-the-good-the-bad-the-ugly-9f68bfc3bb73>

the Kubernetes API is exposed via the Amazon EKS endpoint associated with your cluster.

Amazon EKS worker nodes run in your AWS account and connect to your cluster's control plane via the API server endpoint and a certificate file that is created for your cluster.