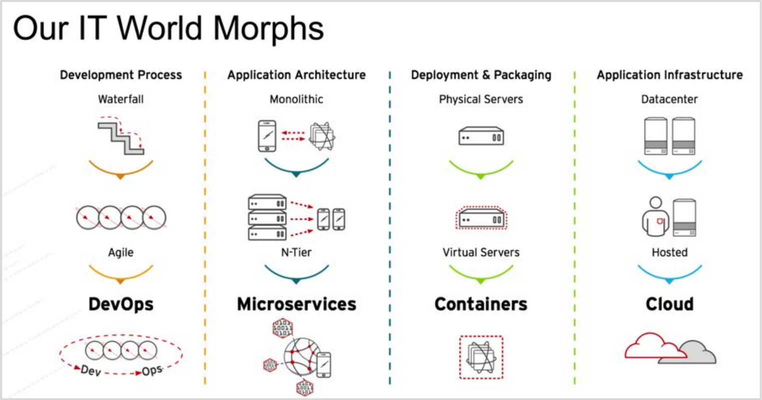
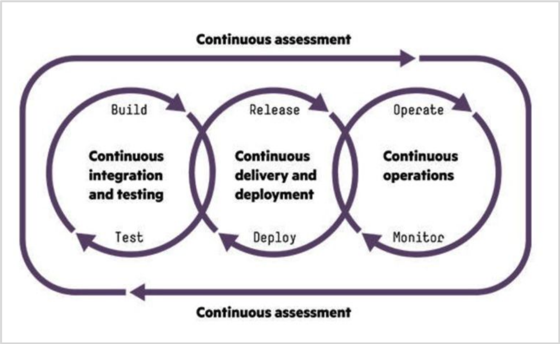
Service Mesh & Security

On Container-based Schedulers

Version 1.2

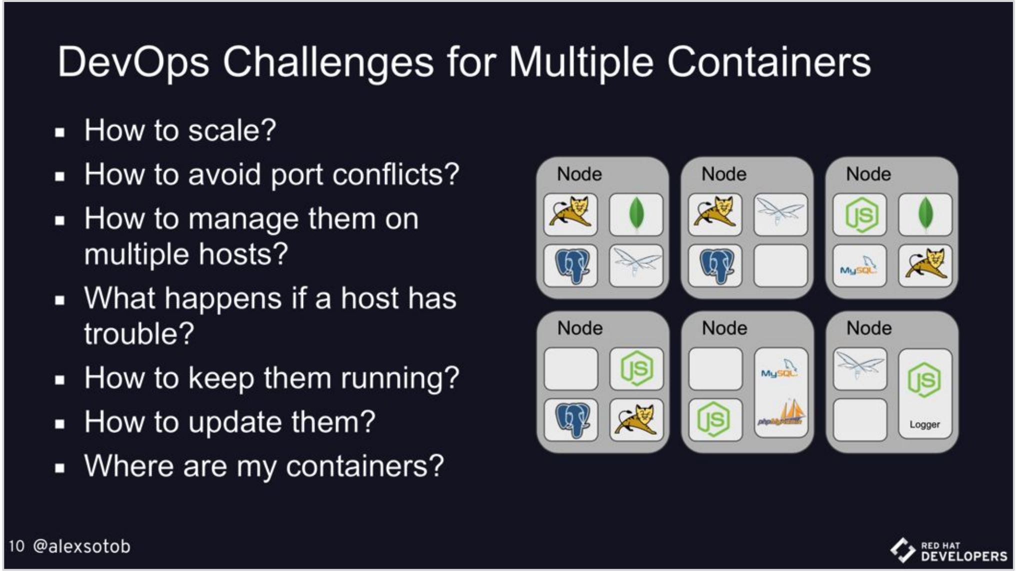
1. Digital Transformation: IT World is changing





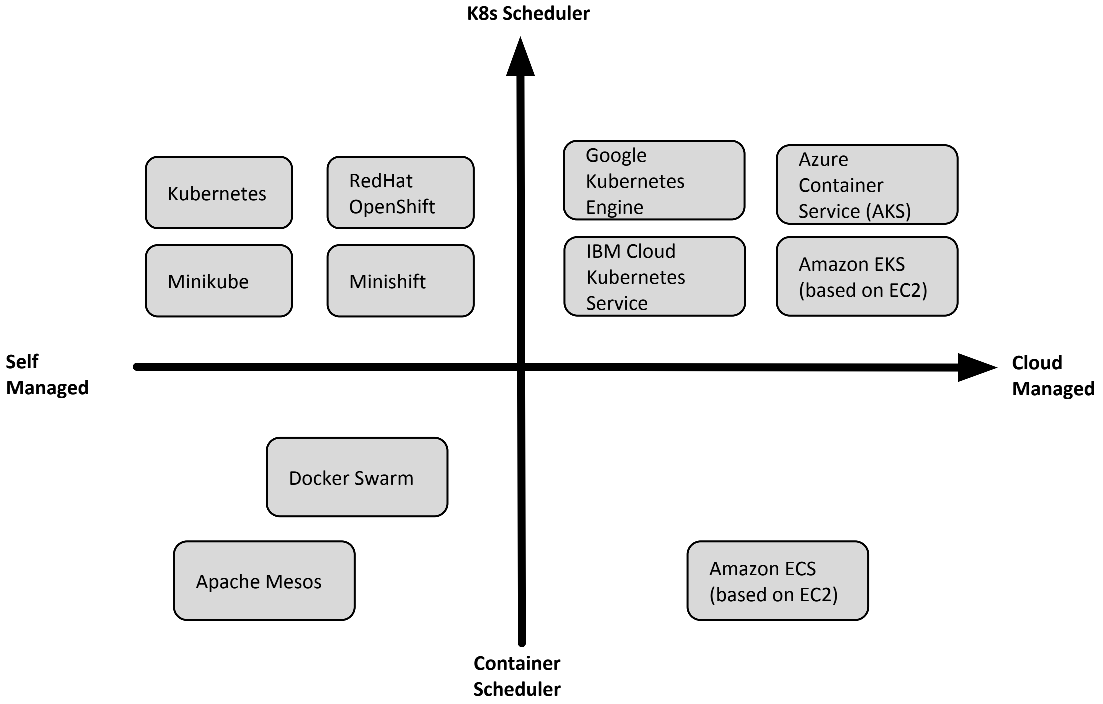
https://www.slideshare.net/asotobu/sail-in-the-cloud

1. Digital Transformation: New challenges



https://www.slideshare.net/asotobu/sail-in-the-cloud

2. Choosing the Container-based Scheduler

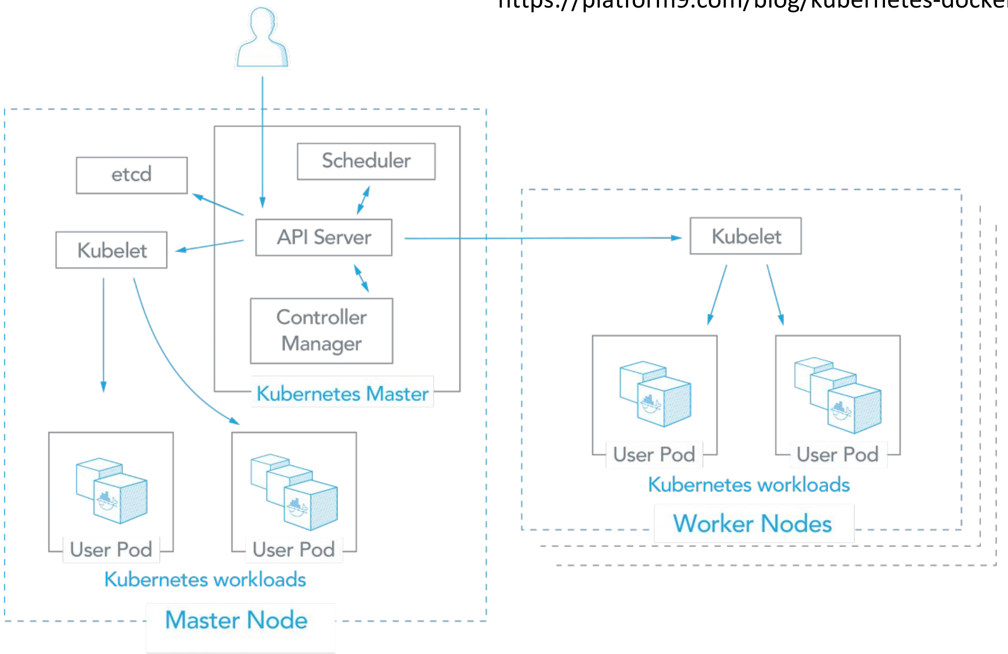


Lab 00: Building K8s

3. Kubernetes

3.1. Kubernetes Architecture

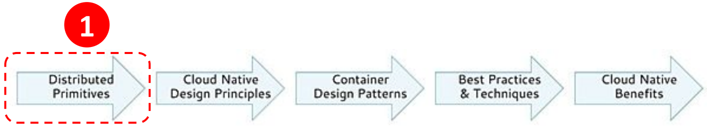
https://platform9.com/blog/kubernetes-docker-swarm-compared



API Server: management hub for Kubernetes
Scheduler: places a workload on the appropriate Node
Controller Manager: scales workloads up/down
etcd: stores configuration data which can be accessed by API Server

Kubelet: Receives pod specifications from API Server, updates Nodes
Master Node: places workloads on Nodes
Worker Nodes: receives requests from Master Nodes and dispatches them
User Pod: a group of containers with shared resources

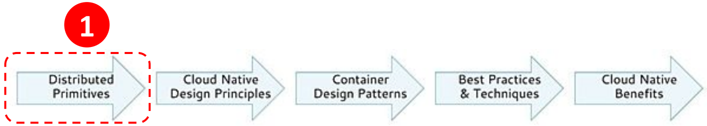
3.2. Kubernetes effect: Primitives (1/2)

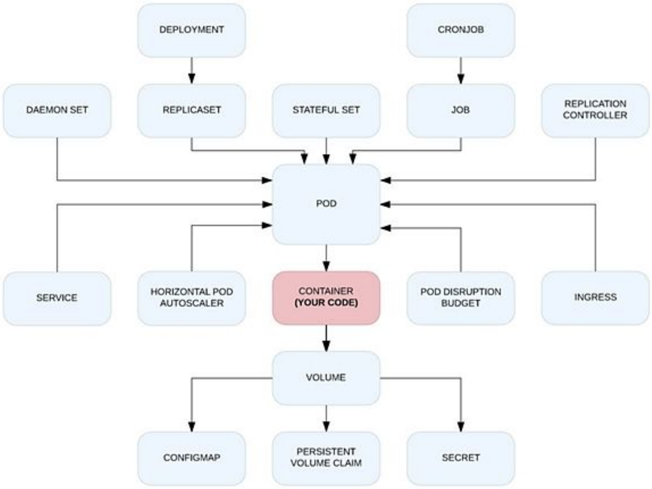


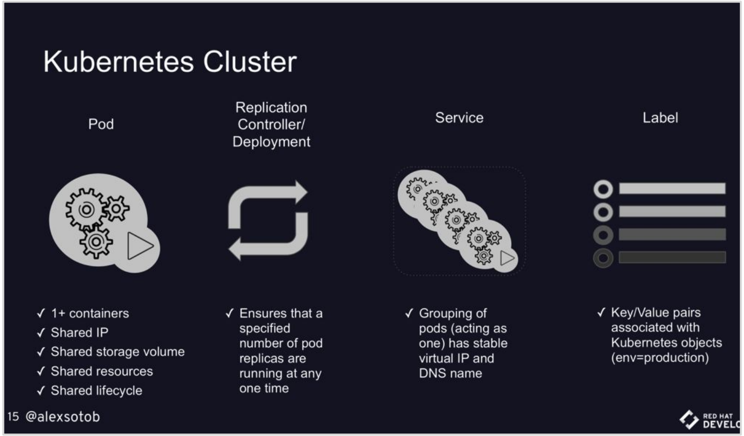
Concern	Java & JVM	Kubernetes
Behaviour encapsulation	Class	Container Image
Behaviour instance	Object	Container
Unit of reuse	.jar	Container Image
Deployment unit	.jar/.war/.ear	Pod
Buildtime/Runtime isolation	Module, Package, Class	Container Image, Namespace
Initialization preconditions	Constructor	Init-container
Post initialization	init-method	PostStart
Pre destroy	destroy-method	PreStop
Cleanup procedure	finalize(), ShutdownHook	Defer-container*
Asynchronous & Parallel execution	ThreadPoolExecutor, ForkJoinPool	Job
Periodic task	Timer, ScheduledExecutorService	CronJob
Background task	Daemon Thread	DaemonSet
Configuration management	System.getenv(), Properties	ConfigMap, Secret

https://www.infoq.com/articles/kubernetes-effect

3.3. Kubernetes effect: Primitives (2/2)

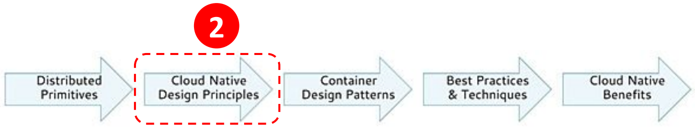


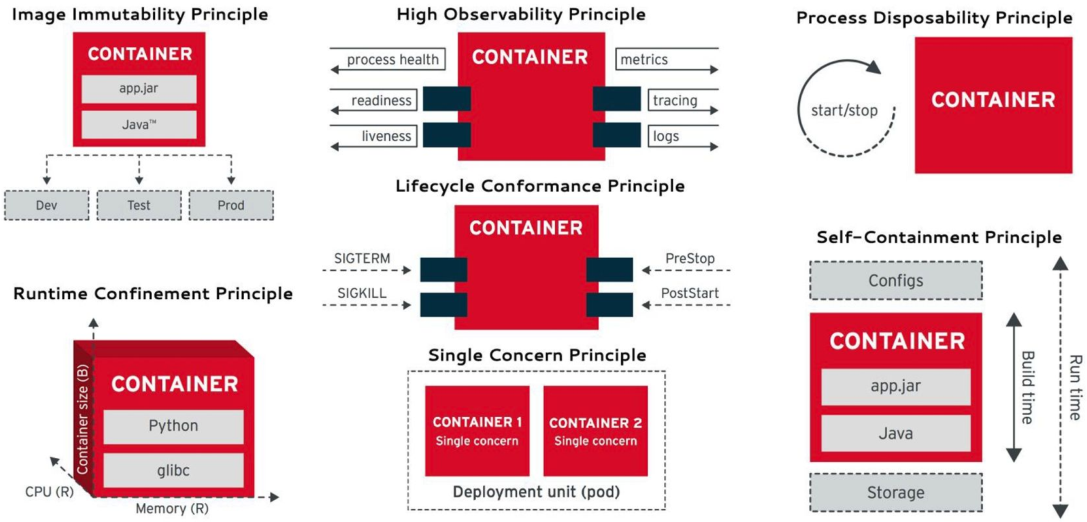




https://www.infoq.com/articles/kubernetes-effect https://www.slideshare.net/asotobu/sail-in-the-cloud

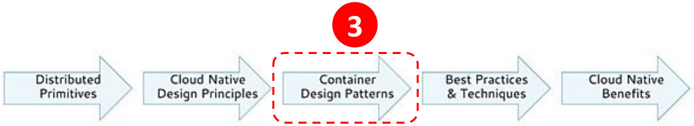
3.4. Kubernetes effect: Principles

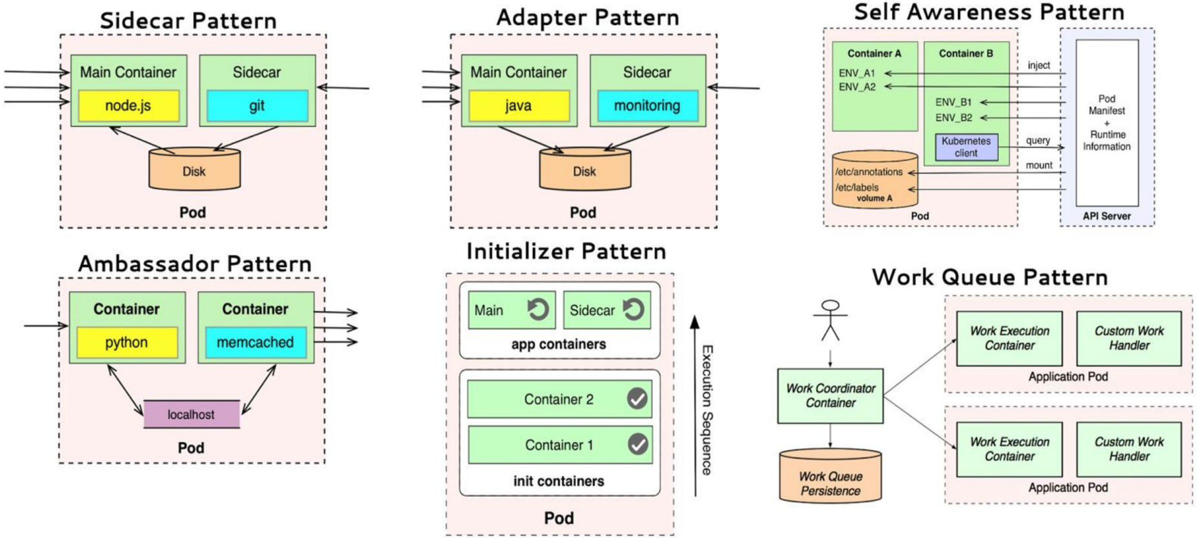




https://www.infoq.com/articles/kubernetes-effect

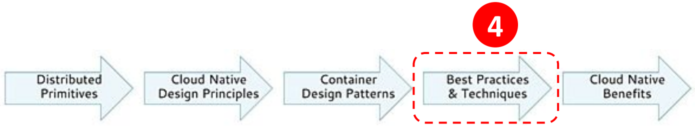
3.5. Kubernetes effect: Patterns





https://www.infoq.com/articles/kubernetes-effect

3.6. Kubernetes effect: Best Practices



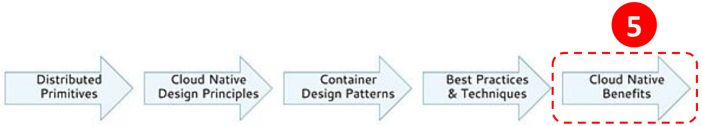
Practices & Techniques

In addition to the principles and patterns, creating good containerized applications requires familiarity with other container-related best practices and techniques. Principles and patterns are abstract, fundamental ideas that change less often. Best practices and the related techniques are more concrete and may change more frequently. Here are some of the common container-related best practices:

- **Aim for small images** - this reduces container size, build time, and networking time when copying container images.
- **Support arbitrary user IDs** - avoid using the sudo command or requiring a specific userid to run your container.
- **Mark important ports** - specifying ports using the EXPOSE command makes it easier for both humans and software to use your image.
- **Use volumes for persistent data** - the data that needs to be preserved after a container is destroyed must be written to a volume.
- **Set image metadata** - Image metadata in the form of tags, labels, and annotations makes your container images more discoverable.
- **Synchronize host and image** - some containerized applications require the container to be synchronized with the host on certain attributes such as time and machine ID.
- **Log to STDOUT and STDERR** - logging to these system streams rather than to a file will ensure container logs are picked up and aggregated properly.

https://www.infoq.com/articles/kubernetes-effect

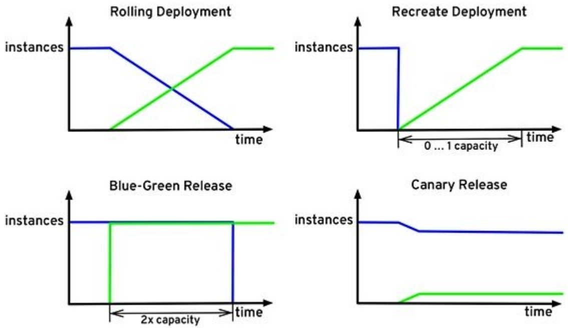
3.7. Kubernetes effect: Benefits

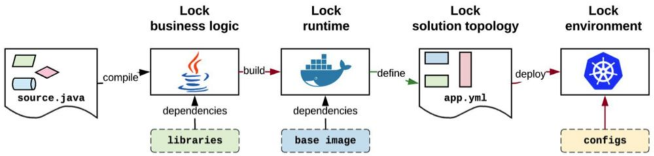


- **Self Service Environments** - enables teams and team members to instantly carve isolated environments from the cluster for CI/CD and experimentation purposes.
- **Dynamically Placed Applications** - allows applications to be placed on the cluster in a predictable manner based on application demands, available resources, and guiding policies.
- **Declarative Service Deployments** - this abstraction encapsulates the upgrade and rollback process of a group of containers and makes executing it a repeatable and automatable activity.

- **Application Resilience** - containers and the management platforms improve the application resiliency in a variety of ways such as:
  - Infinite loops: CPU shares and quotas
  - Memory leaks: OOM yourself
  - Disk hogs: quotas
  - Fork bombs: process limits
  - Circuit breaker, timeout, retry as sidecar
  - Failover and service discovery as sidecar
  - Process bulkheading with containers
  - Hardware bulkheading through the scheduler
  - Auto-scaling & self-healing

- **Service Discovery & Load Balancing & Circuit Breaker** - the platform allows services to discovery and consume other services without in application agents. Further, the usage of sidecar containers and tools such as Istio framework allow to completely move the networking related responsibilities outside of the application to the platform level.
- **Declarative Application Topology** - using Kubernetes API objects allow us to describe how our services should be deployed, their dependency on other services and resources prerequisites. And having all this information in an executable format allows us to test the deployment aspects of the application in the early stage of development and treat it as programmable application infrastructure.





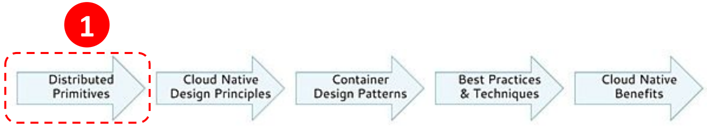
https://www.infoq.com/articles/kubernetes-effect

4. Amazon EKS and ECS

4.1. Comparison (1/2)

#	Criteria	EKS		ECS	
1	Learning Curve and Community	Since it's based on K8s, the same K8s' community and knowledge is applicable to EKS.	8	The unique way to learn about AWS is reading the public documentation and Customer Forums.	4
2	Interoperability	K8s is the Scheduler and Orchestration tool de-facto and all kind of artefacts were implemented following the 'Infrastructure as a Code'. It means that we can move any artefact to other K8s Cloud Provider and/or On-premise to On-cloud.	8	Amazon ECS is tightly integrated with other Amazon services. It isn't interoperable because it will work only with other AWS services.	3
3	Pluggable Architecture Overlay Network (Calico, Flannel, WeaveNet, ...) Storage (Ceph, GlusterFS, NFS) Etc.	K8s is designed to be as much modular as possible. It supports different LB, network models (OpenVSwitch, Flannel, Calico, etc.), volumes. K8s is also designed to support different container engines (runtimes). Docker is the default, however there is an implementation that enables support of rkt containers. ECS supports only Docker containers at the moment.	8	It's pluggable by default to its own Services (IAM, ELB, S3, Route 53, VPC, Security Groups, ...). Integrate to 3r Party Services will require extra effort to do that integration.	6
4	Kubernetes Primitives ConfigMap, Secrets, Controller, Pod, Ingress, ...	See 'The Kubernetes Effect'	9	You have to learn the Primitives behind AWS. ECS relies on other Amazon services, such as Identity and Access Management (IAM), Domain Name System (Route 53), Elastic Load Balancing (ELB), and EC2. This allows using the familiar concepts such as Security Groups, IAM policies to manage your containers. ECS allows to run a custom Docker registry based on S3. ECS does not support secrets directly, however it is possible to encrypt secrets using Amazon Key Management Service (KMS) and decrypt them in containers. In ECS, there is no direct alternative for Kubernetes Config Maps. It does not have a way to pass configuration to a container other than with environment variables, and the only way to specify the same values for several containers is to copy and paste them.	6
5	Security L4 Firewalling L7 Firewalling Network Policy Pod Policy Service Identity (SPIFFE) Throttling Certificate Validation End to end Monitoring in real time (based on Sidecar)				
6	Service Mesh Traffic Management Service Identity (SPIFFE) Certificate Management TLS Termination Ingress and Egress Observability (Monitoring, Tracing, Logging, Metrics, ...)				
			33		**19**

4.2. Comparison (2/2)



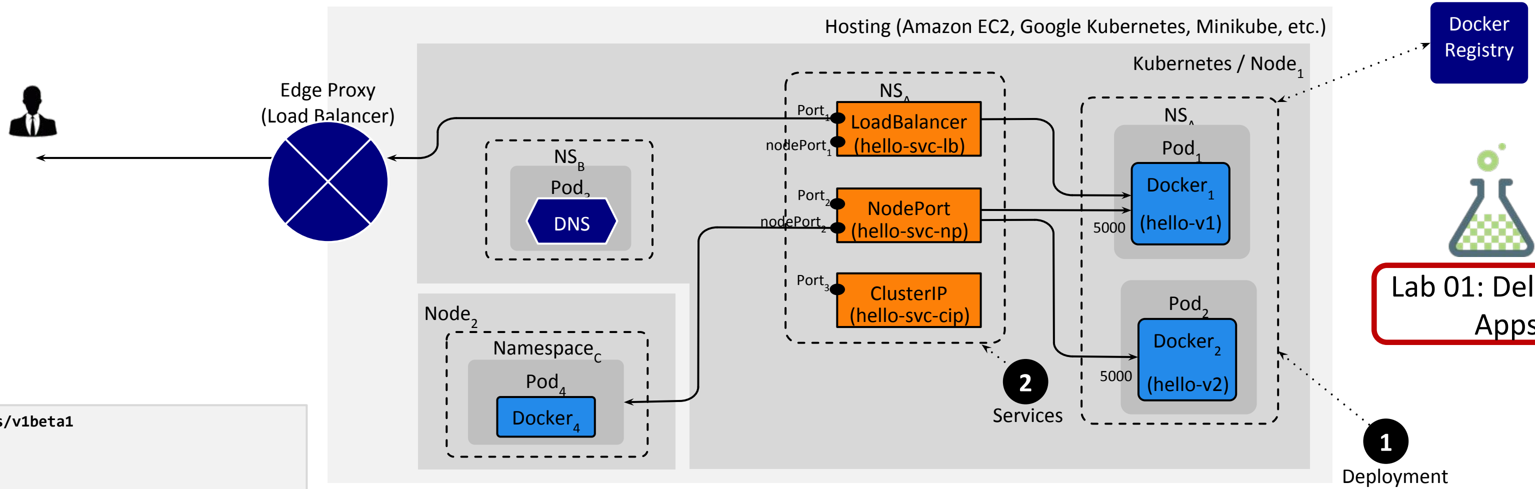
Concern	Java & JVM	Kubernetes
Behaviour encapsulation	Class	Container Image
Behaviour instance	Object	Container
Unit of reuse	.jar	Container Image
Deployment unit	.jar/.war/.ear	Pod
Buildtime/Runtime isolation	Module, Package, Class	Container Image, Namespace
Initialization preconditions	Constructor	Init-container
Post initialization	init-method	PostStart
Pre destroy	destroy-method	PreStop
Cleanup procedure	finalize(), ShutdownHook	Defer-container*
Asynchronous & Parallel execution	ThreadPoolExecutor, ForkJoinPool	Job
Periodic task	Timer, ScheduledExecutorService	CronJob
Background task	Daemon Thread	DaemonSet
Configuration management	System.getenv(), Properties	ConfigMap, Secret

ECS
Where are the Primitives, Patterns ?

https://www.infoq.com/articles/kubernetes-effect

5. Distributed System delivery on Kubernetes

5.1. Delivery of Pods & Services



Lab 01: Delivering Apps

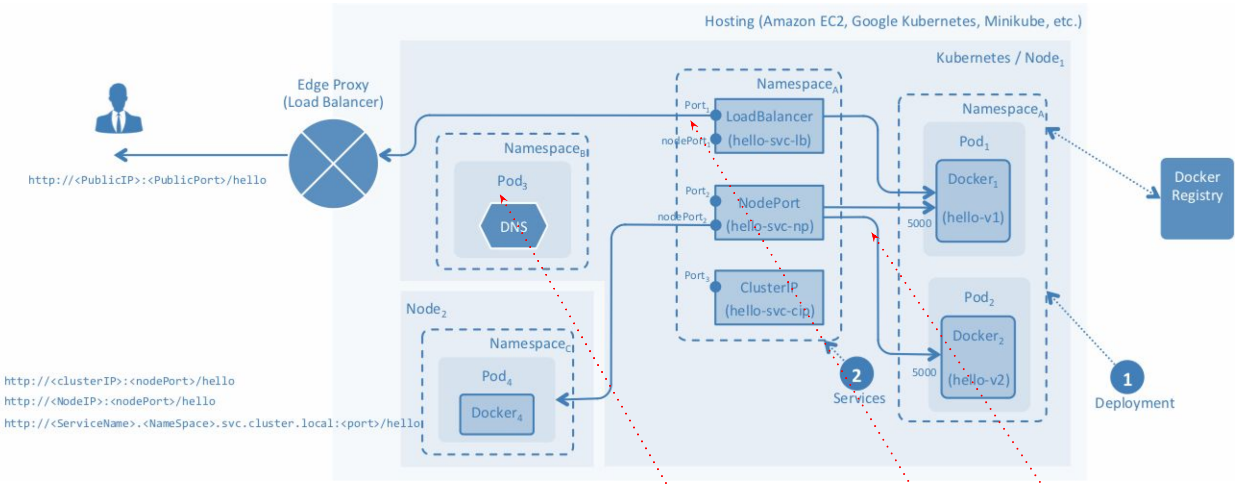
```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-v1
  namespace: hello
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: hello
        version: v1
    spec:
      serviceAccountName: helloworld-sa
      containers:
      - name: helloworld
        image: istio/examples-helloworld-v1
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5000
```

1 Deployment

2 Kubernetes Services:

- `ClusterIP` : Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster. This is the default `ServiceType` .
- `NodePort` : Exposes the service on each Node's IP at a static port (the `NodePort`). A `ClusterIP` service, to which the `NodePort` service will route, is automatically created. You'll be able to contact the `NodePort` service, from outside the cluster, by requesting `<NodeIP>:<NodePort>` .
- `LoadBalancer` : Exposes the service externally using a cloud provider's load balancer. `NodePort` and `ClusterIP` services, to which the external load balancer will route, are automatically created.
- `ExternalName` : Maps the service to the contents of the `externalName` field (e.g. `foo.bar.example.com`), by returning a `CNAME` record with its value. No proxying of any kind is set up. This requires version 1.7 or higher of `kube-dns` .

5.2. Delivering Distributed Apps: Issues



Any container can hit to other container running in different namespaces.
Too difficult to add security (MTLS, ACL, ..)

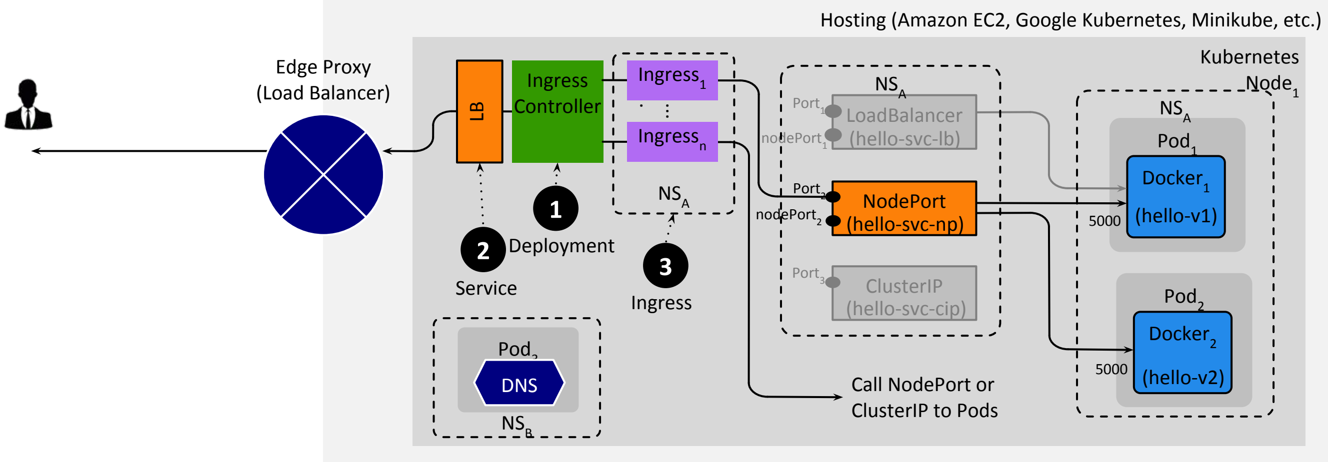
Too difficult to add more routes.

Sometimes the Edge Proxy should be managed / reconfigured everytime a new Route is added/changed.

?

I've deployed my App (Pods & Services) in Kubernetes. Now, Will the CI/CD scale?, Is it secure?

5.3. Traffic Mgmt. - Ingress



Lab 02: Ingress

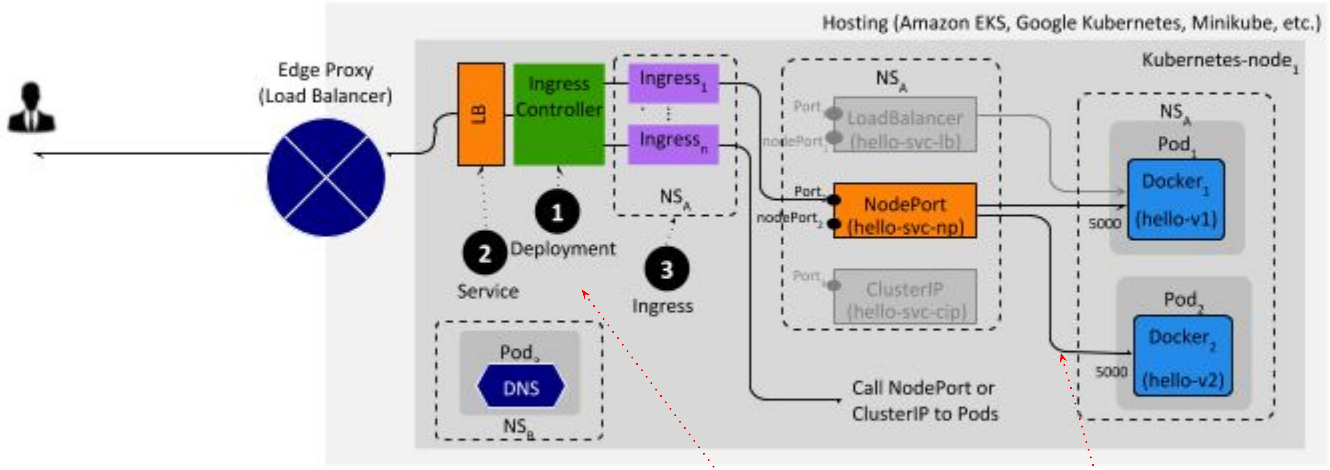
Ingress Controller

1. Deploy the choosen Ingress Controller in Pods and in a specified Namespace. Default HTTP Backend with custom error pages can be implemented.
2. Configure and integrate the deployed Ingress Controller with the EdgeProxy (Firewall, Root Proxy, Load Balancer, etc.). Generally, Ingress Controller is configured as Kubernetes Service LoadBalancer.
3. Finally, during CD time, every API/Microservice will be deployed with Ingress Resource definition (routing). TLS, MTLS or HTTP can be defined.

```
---
apiVersion: v1
kind: Service
metadata:
  name: hello-svc-np
  labels:
    app: hello
  namespace: hello
spec:
  type: NodePort
  ports:
  - name: http
    port: 5030
    targetPort: 5000
  selector:
    app: hello
---
```

```
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-ing
  annotations:
    kubernetes.io/ingress.class: istio
spec:
  rules:
  - http:
      paths:
      - path: /hello
        backend:
          serviceName: hello-svc-np
          servicePort: 5030
---
```

5.4. Traffic Mgmt. - Ingress: Issues



?

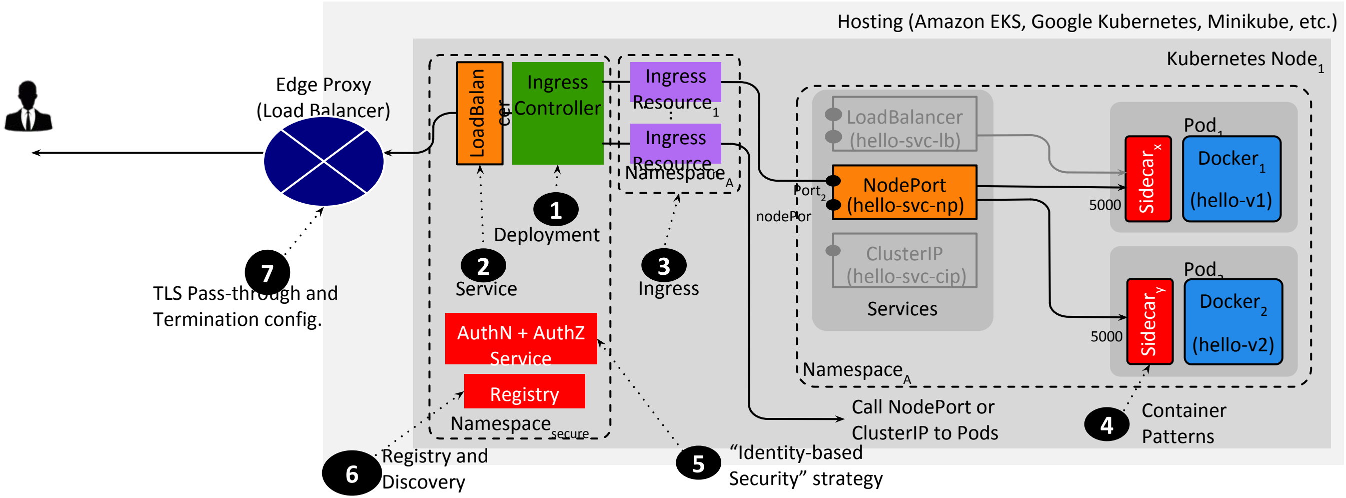
I've implemented the Ingress Controller in Service Mesh. Now, Is that Secure?

It isn't secure because the Pods still are accesibles.
Any other Pod (container) can call this.

It isn't secure because that doesn't have:

- Separation of Duties
- Principle of Last Privilege
- Identity-based Security
- End-to-end security: the App Container still is available and nobody can stop to make call to that Container.
- Container Patterns (sidecar container as last-mile protection)
- Registry, Auditability and Traceability. It means that all App Containers (APIs and Microservices) living in the Service Mesh should be Identified and Traceable.

5.5. Traffic Mgmt. - Ingress, Sidecar



Lab 03: Ingress and Sidecar

Ingress Controller

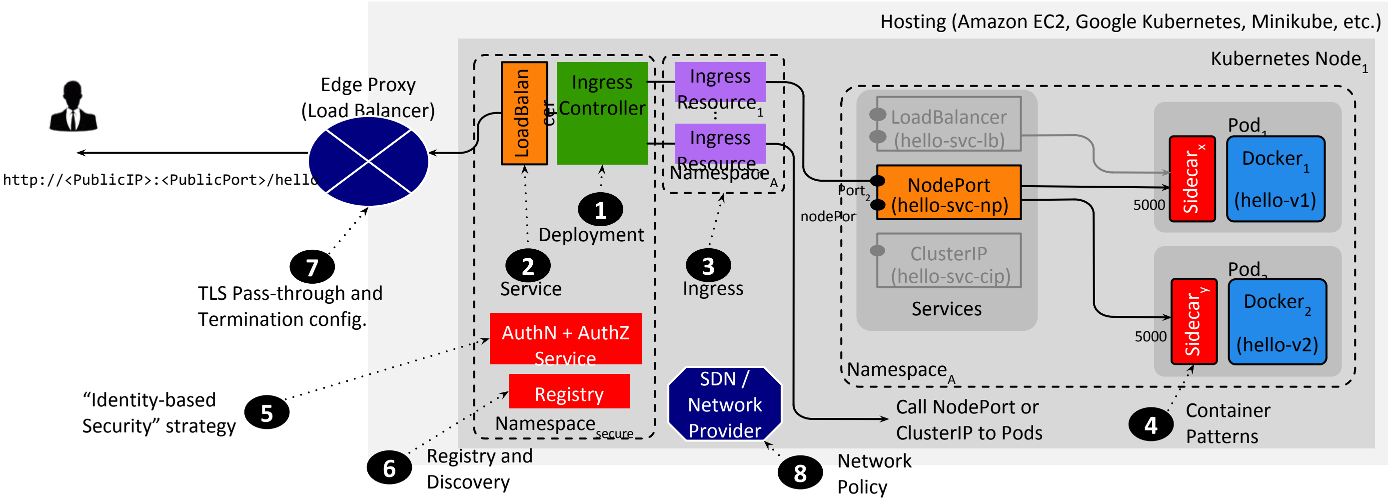
1. Deploy the chosen Ingress Controller in Pods and in a specified Namespace. Default HTTP Backend with custom error pages can be implemented.
2. Configure and integrate the deployed Ingress Controller with the EdgeProxy (Firewall, Root Proxy, Load Balancer, etc.). Generally, Ingress Controller is configured as Kubernetes Service LoadBalancer.
3. Finally, during CD time, every API/Microservice will be deployed with Ingress Resource definition (routing). TLS, MTLS or HTTP can be defined.

\+

Traffic Management (L7 Firewall)

4. Inject Sidecar in the same Pod where the App Container is living. Sidecar bootstraps security config. to reject traffic from untrusted source.
5. The "Single Source of Truth" does validate and authorize the incomming traffic.
6. It provides a fine-grained control over all App Containers running in the Service Mesh.
7. The TLS termination should be configured properly in order to be managed dynamically for the Ingress Controller. It means to manage the lifecycle of the TLS Certs.

5.6. Traffic Mgmt. - Ingress, Sidecar, Network Policy



Lab 04: Ingress, Sidecar and Network Policy

Ingress Controller

1 Deploy the choosen Ingress Controller in Pods and in a specified Namespace. Default HTTP Backend with custom error pages can be implemented.

2 Configure and integrate the deployed Ingress Controller with the EdgeProxy (Firewall, Root Proxy, Load Balancer, etc.). Generally, Ingress Controller is configured as Kubernetes Service LoadBalancer.

3 Finally, during CD-time, every API/Microservice will be deployed with Ingress Resource definition (routing). TLS, MTLS or HTTP can be defined.

+

Secure Traffic Management (L7 Firewall)

4 Inject Sidecar in the same Pod where the App Container is living. Sidecar bootstraps security config. to reject traffic from untrusted source.

5 The “Single Source of Truth“ does validate and authorize the incomming traffic.

6 It provides a fine-grained control over all App Containers running in the Service Mesh.

7 The TLS termination should be configured properly in order to be managed dynamically for the Ingress Controller. It means to manage the lifecycle of the TLS Certs.

+

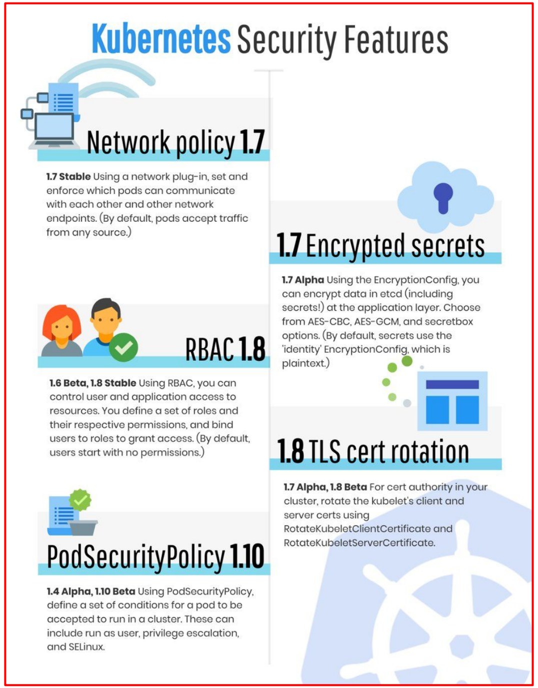
Secure Traffic Management (L3/L4 Firewall)

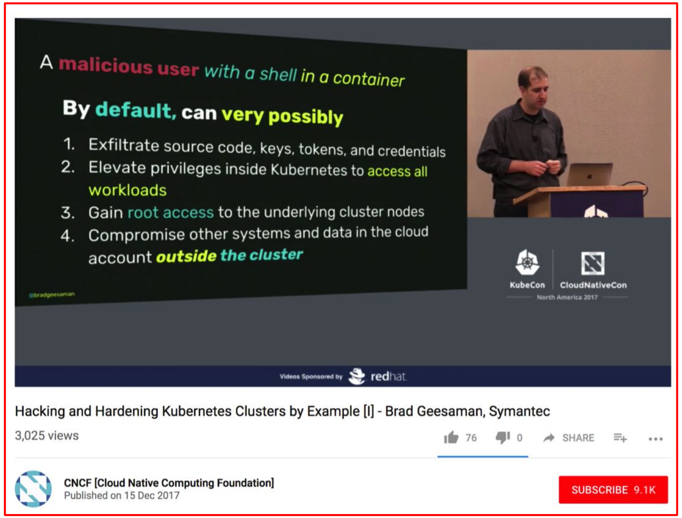
7 The TLS termination should be configured properly in order to be managed dynamically for the Ingress Controller. It means to manage the lifecycle of the TLS Certs.

4 The Sidecar assures End-to-end TLS communication (between API/Microservice and Edge Proxy through Ingress Controller). Additionally, Sidecar manages the TLS Certificate Lifecycle (revocation, renewal, validations, etc.)

6. Continuous Security

6.1. Security Assessment – New threat vectors



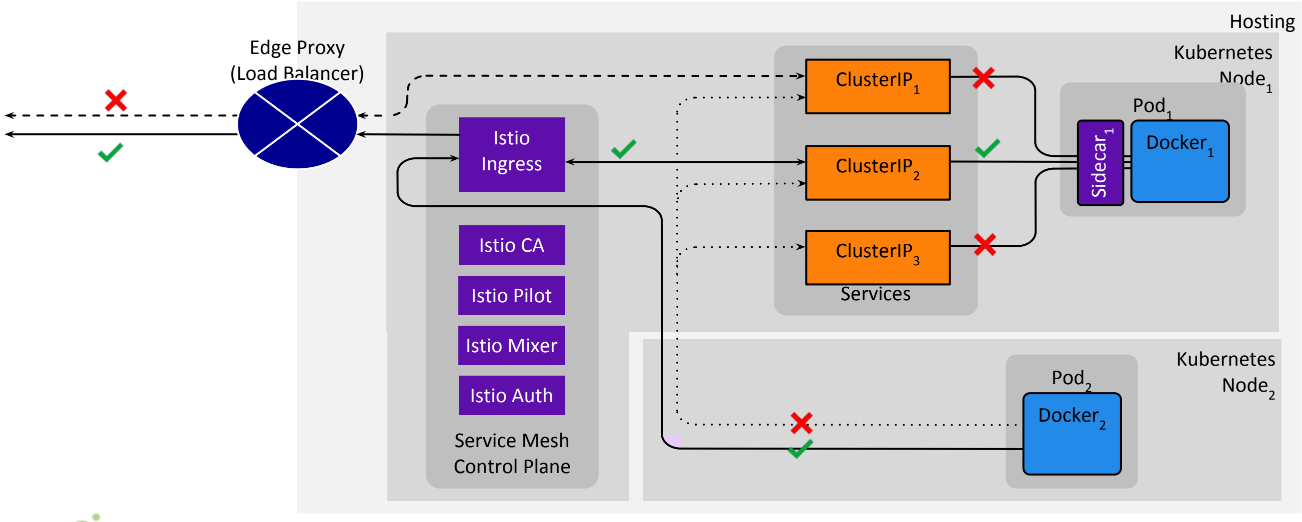


Lab 05: Security Assessment

7. Service Mesh

Reference Architecture

7.1. Service Mesh: Istio Framework



Lab 06: Weaving Service Mesh