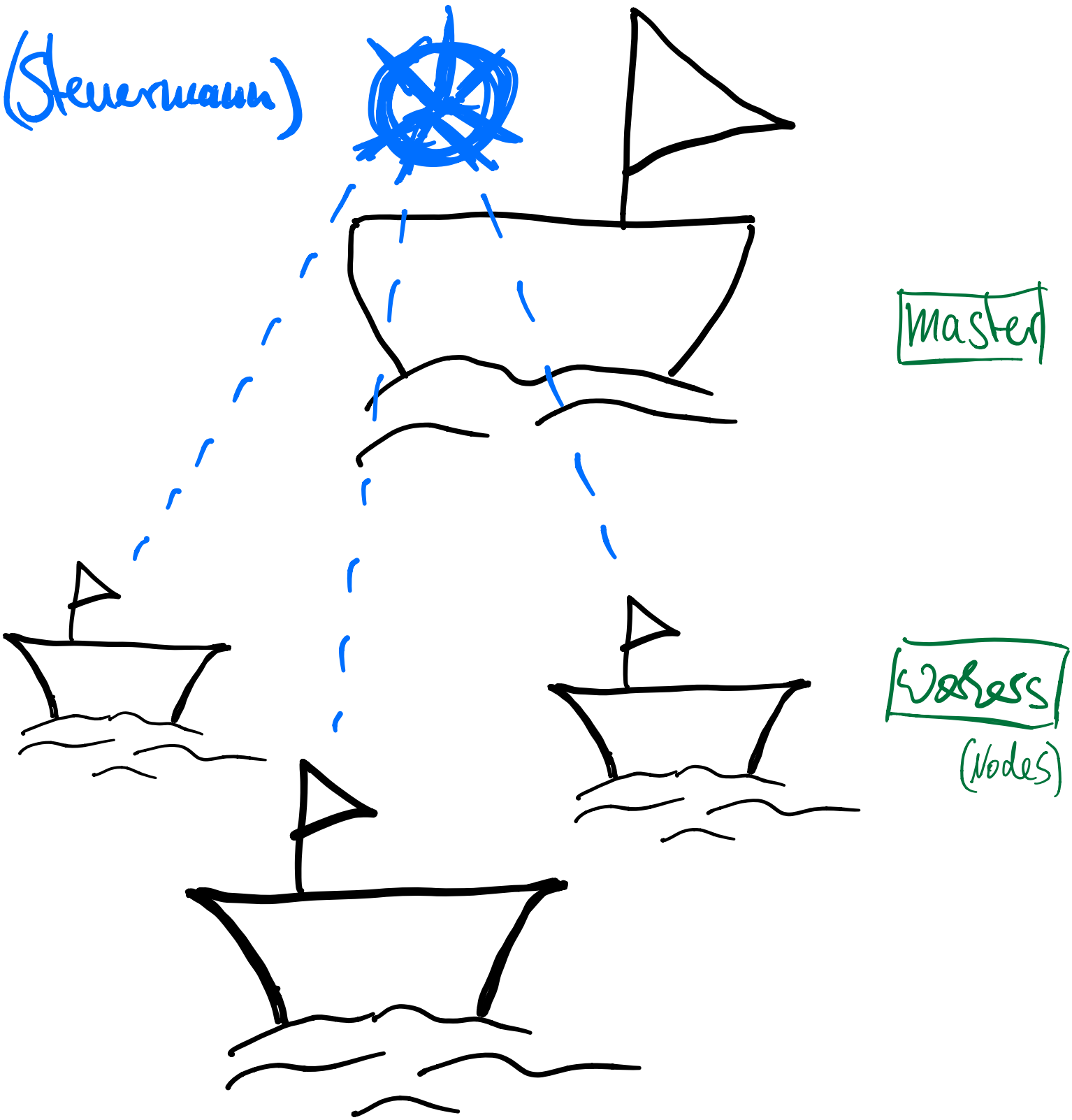
Kubernetes

Good Practices

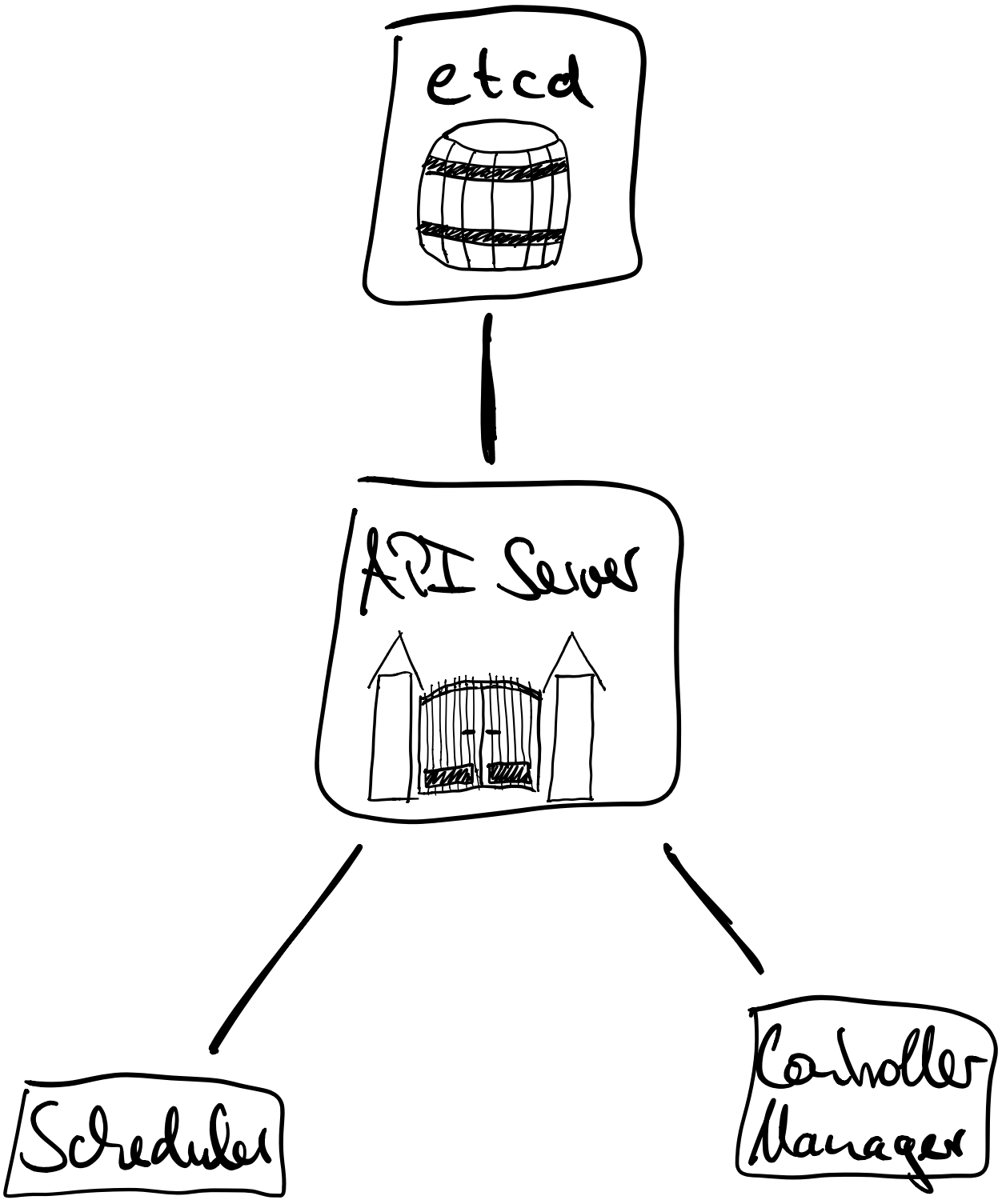
Brainstorming

What is Kubernetes?

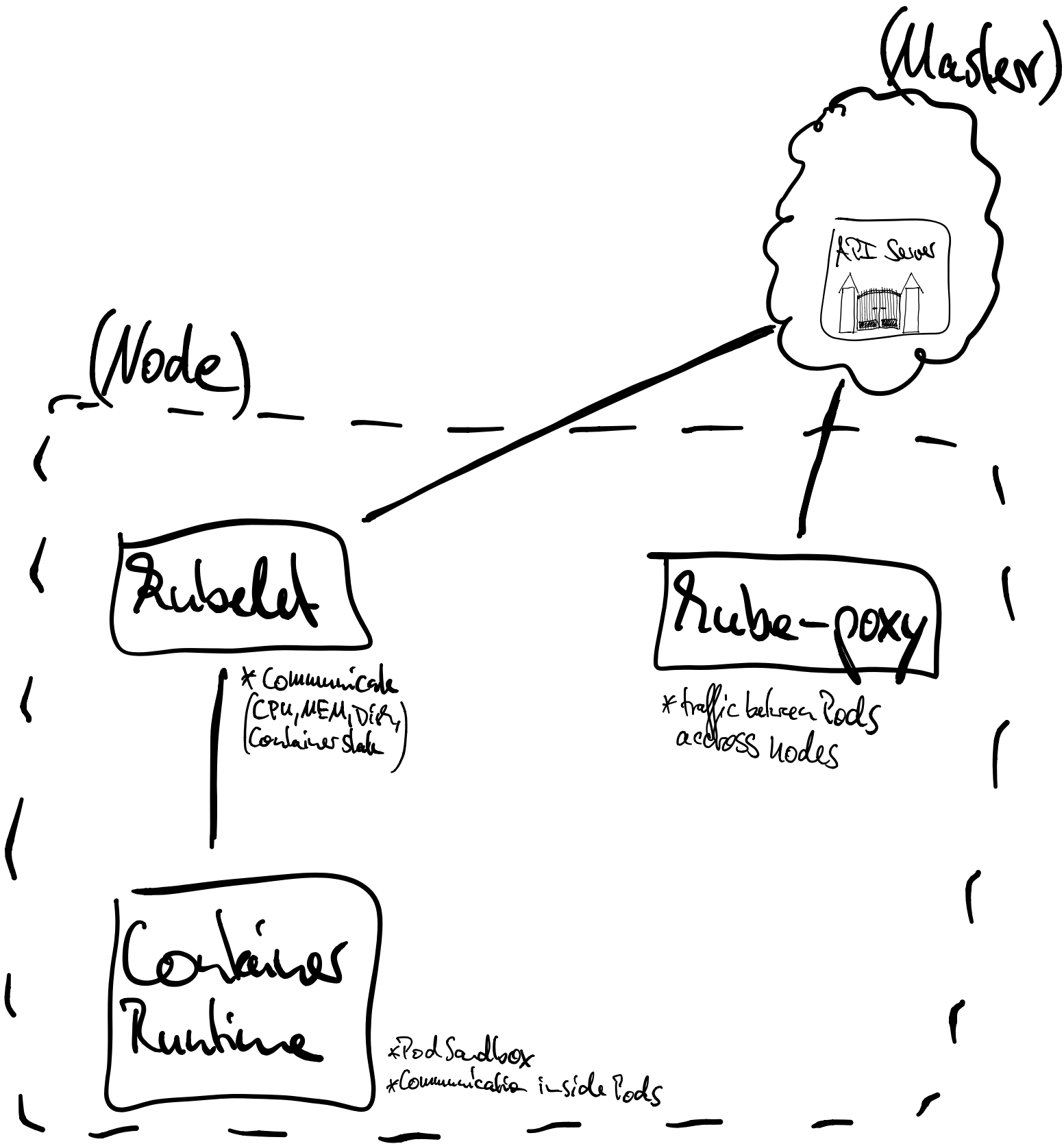
World Map



Master



Node


(Master)
API Server
(Node)
Kubelet
* Communicate (CPU, MEM, Disk, Container Stats)
Kube-proxy
* traffic between Pods across Nodes
Container Runtime
*Pod Sandbox
*Communication inside Pods

What has changed?

* IT
* Software

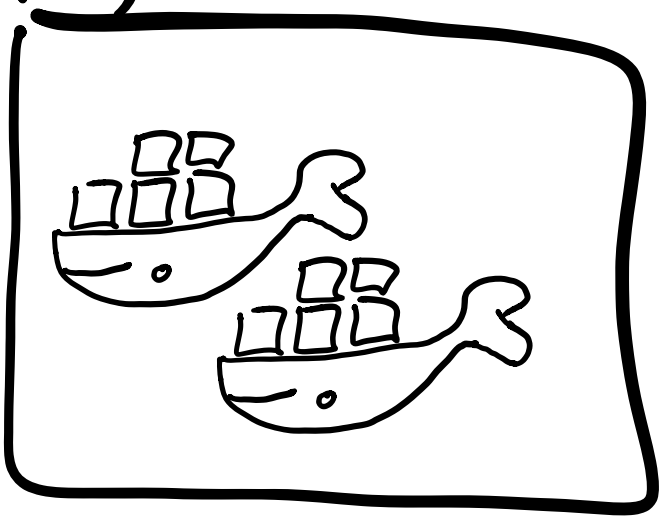
Kubernetes

Basic Objects

* Pod
* Deployment
* Service
* Namespace
* Ingress

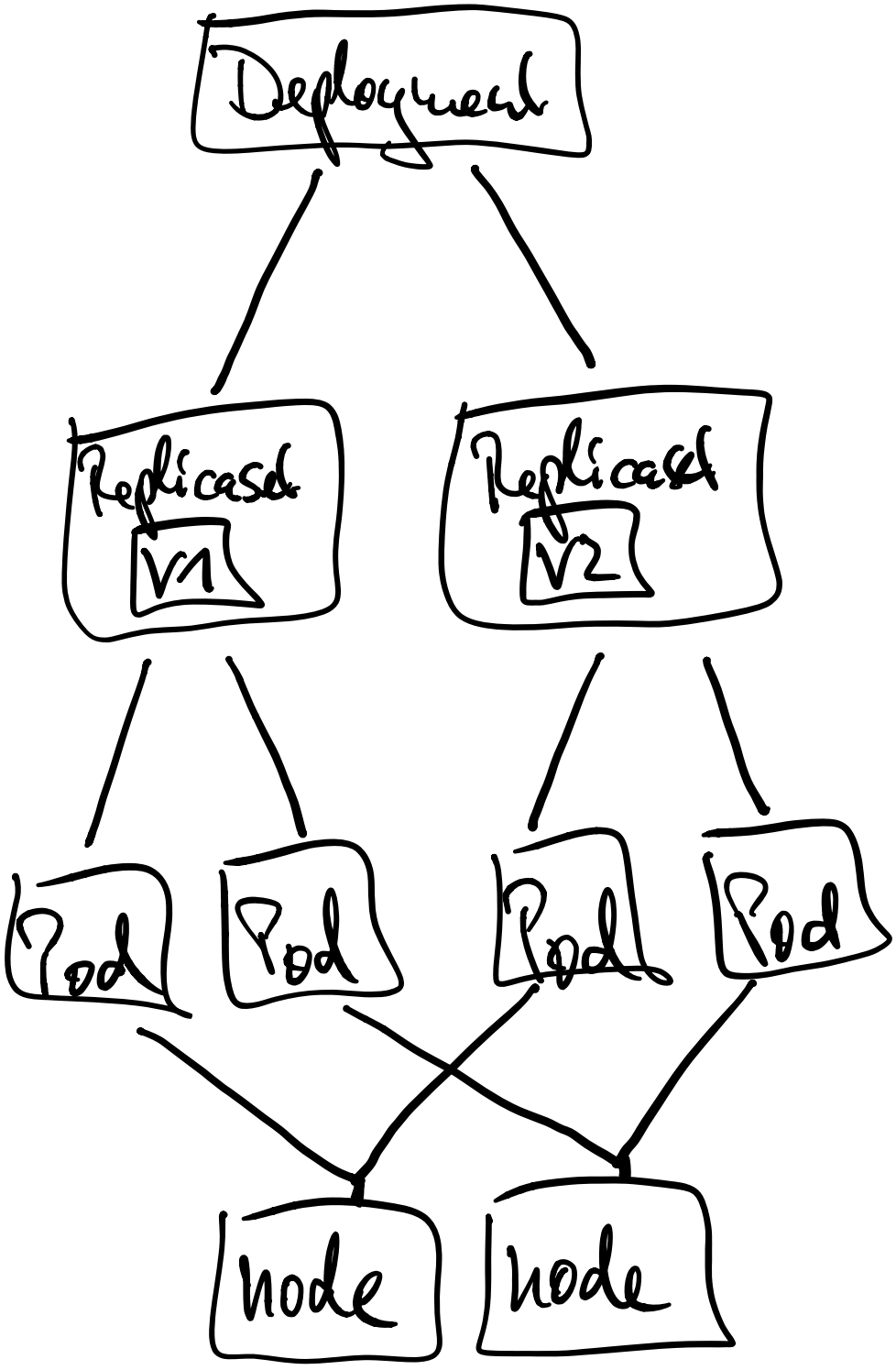
Pod

(Pod)



* Containers inside a pod share many resources
* atomic unit of scheduling

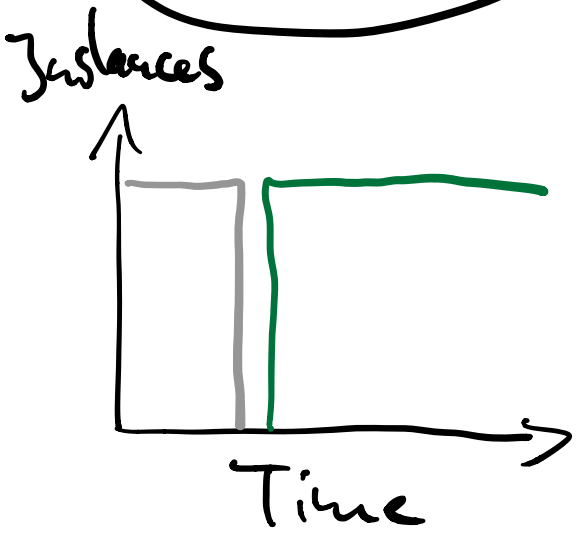
Deployment


Deployment
Replicaset
V1
Replicaset
V2
Pod
Pod
Pod
Pod
node
node

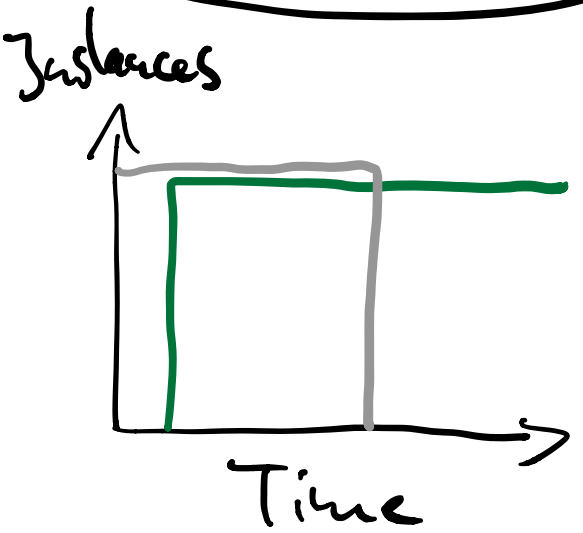
Deployment Strategies

Rolling

Fixed

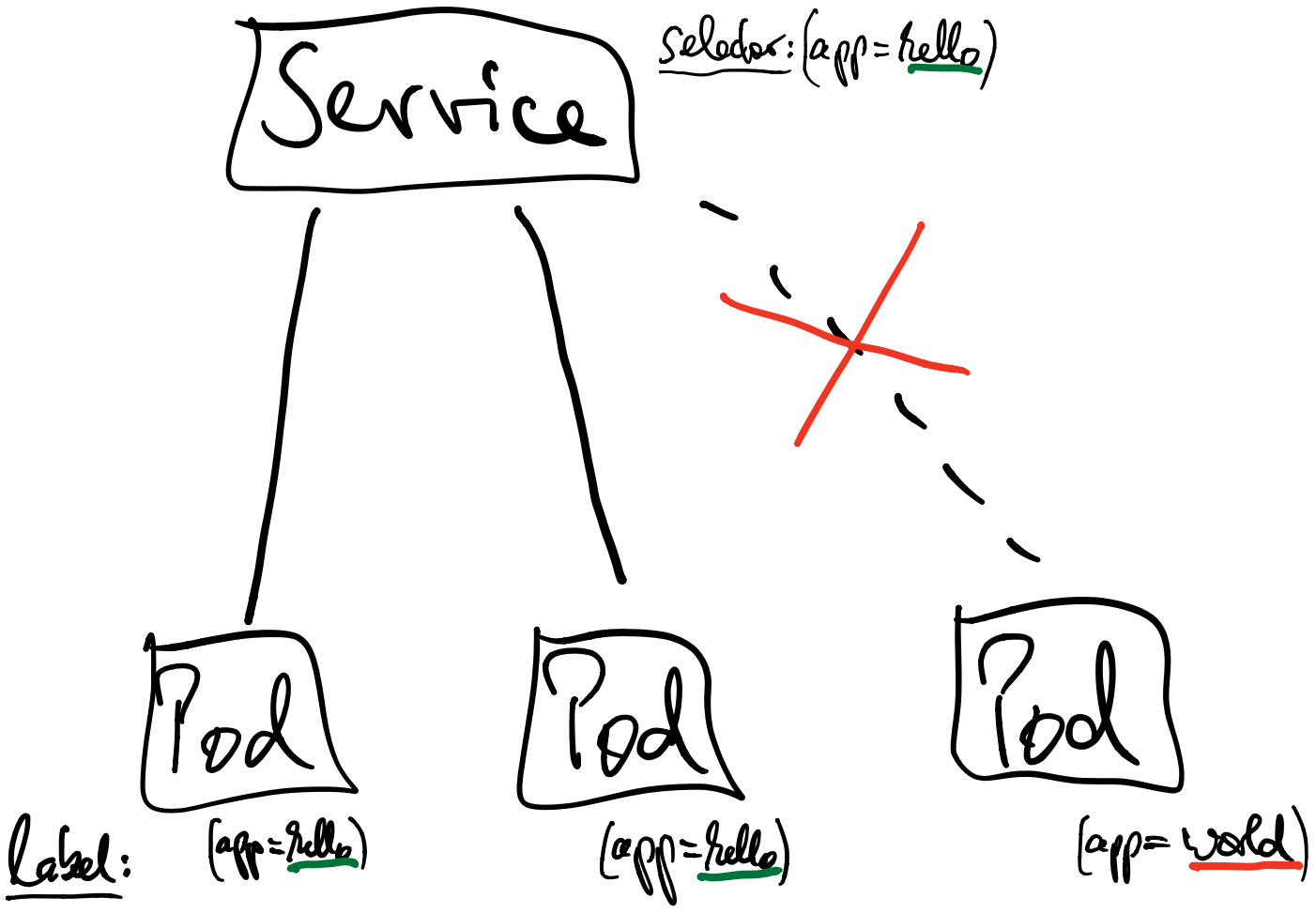


Blue-Green

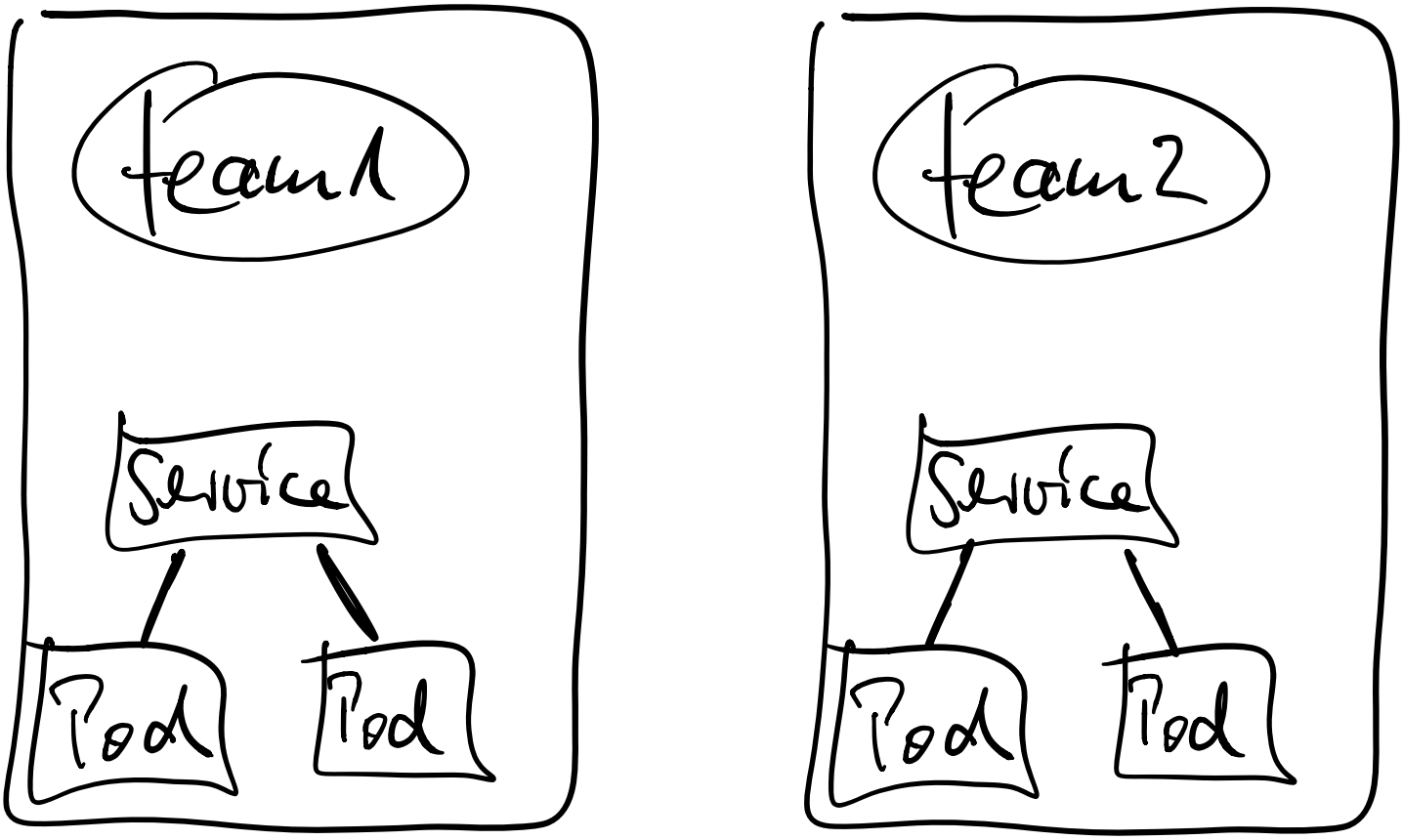


Canary

Service

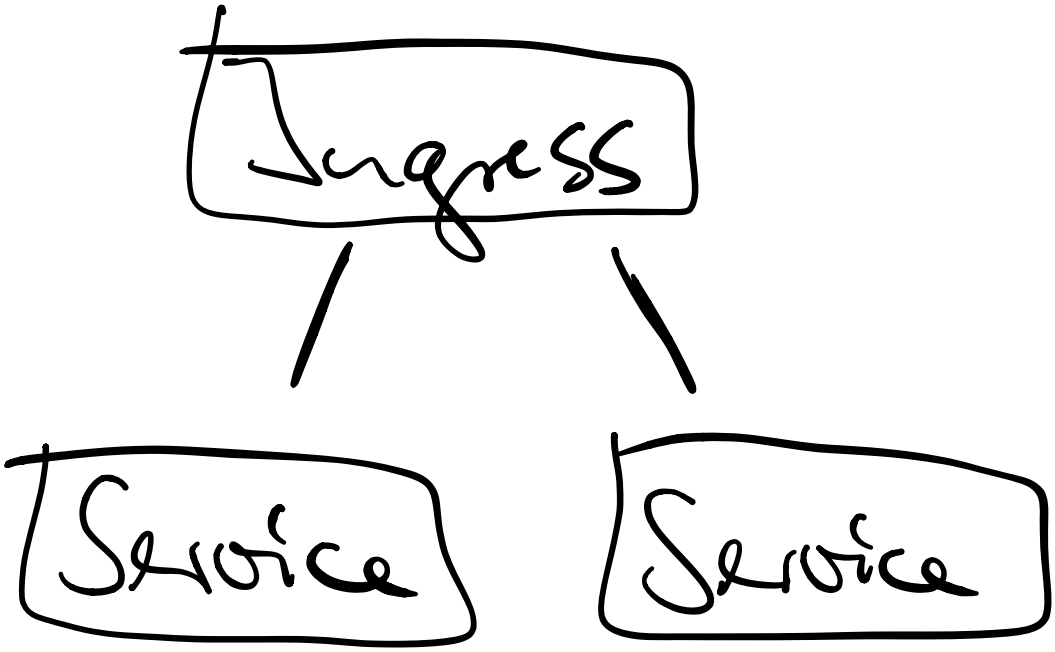

Service
Selector: (app = hello)
Pod
Pod
Pod
Label:
(app=hello)
(app=hello)
(app= world)

Namespace



* isolate names
* Resource Quotas

Ingress



* HTTP-level router
* routing rules (regex, path, domain)

Hands-On

I hear, and I forget.

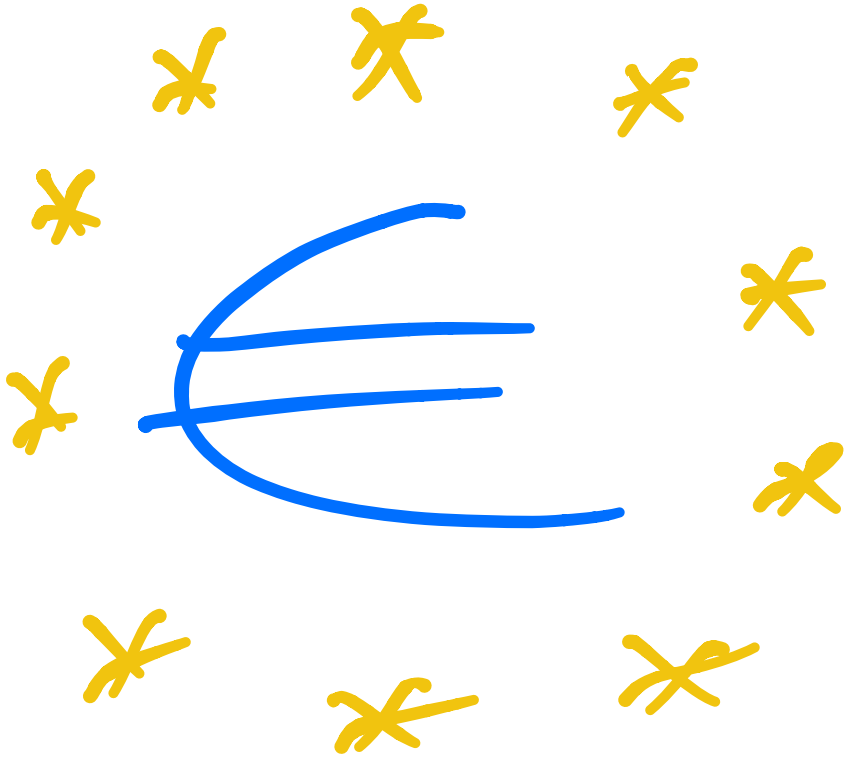
I see, and I remember.

I do, and I understand.

– Chinese proverb

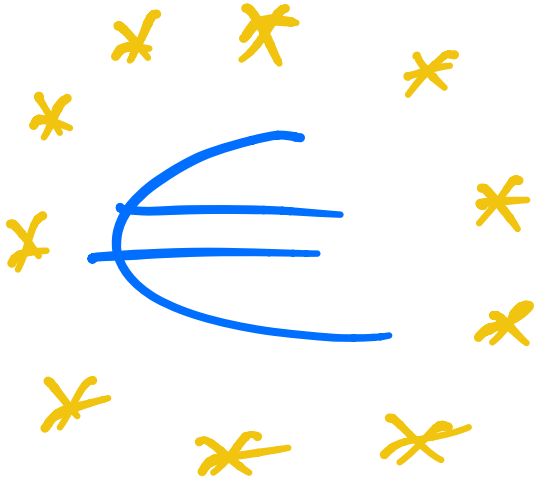
Good Practices

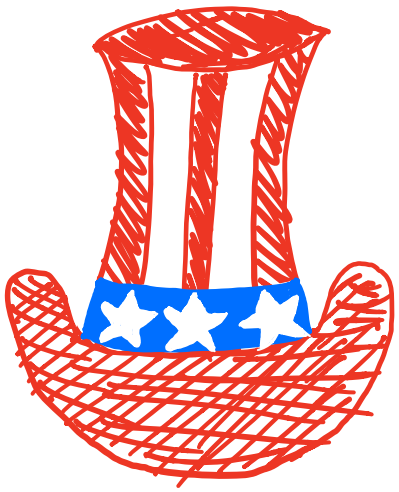
Good European Citizen



EU vs US

Citize



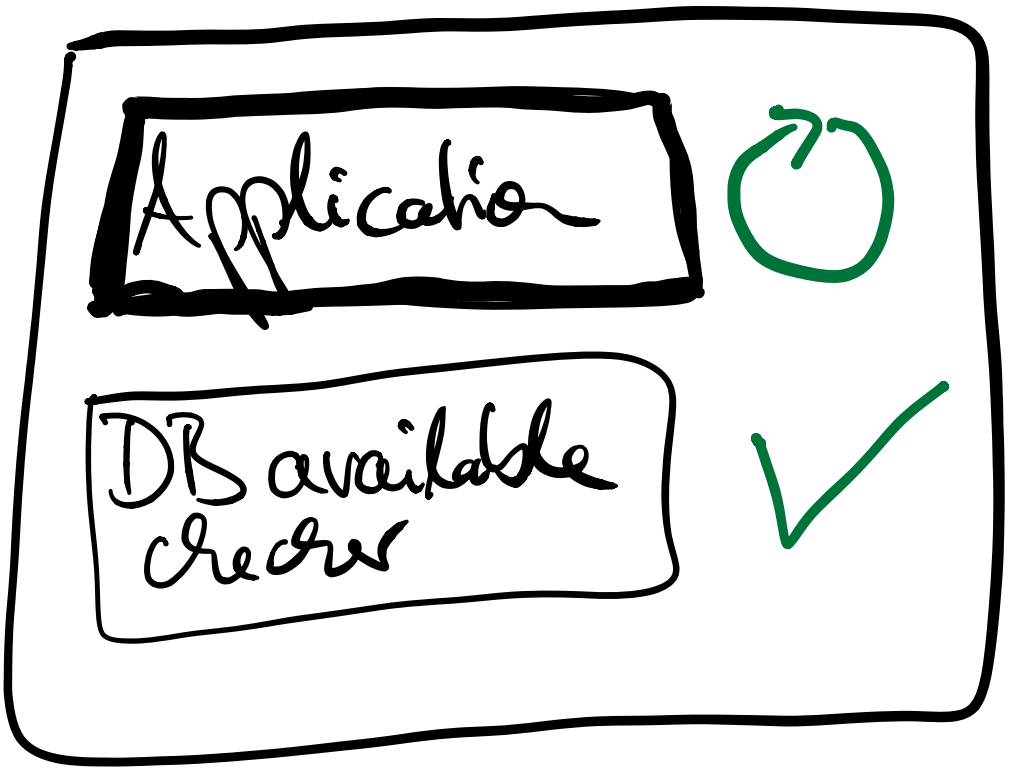


Good Cloud Native Citizen

Structural Patterns

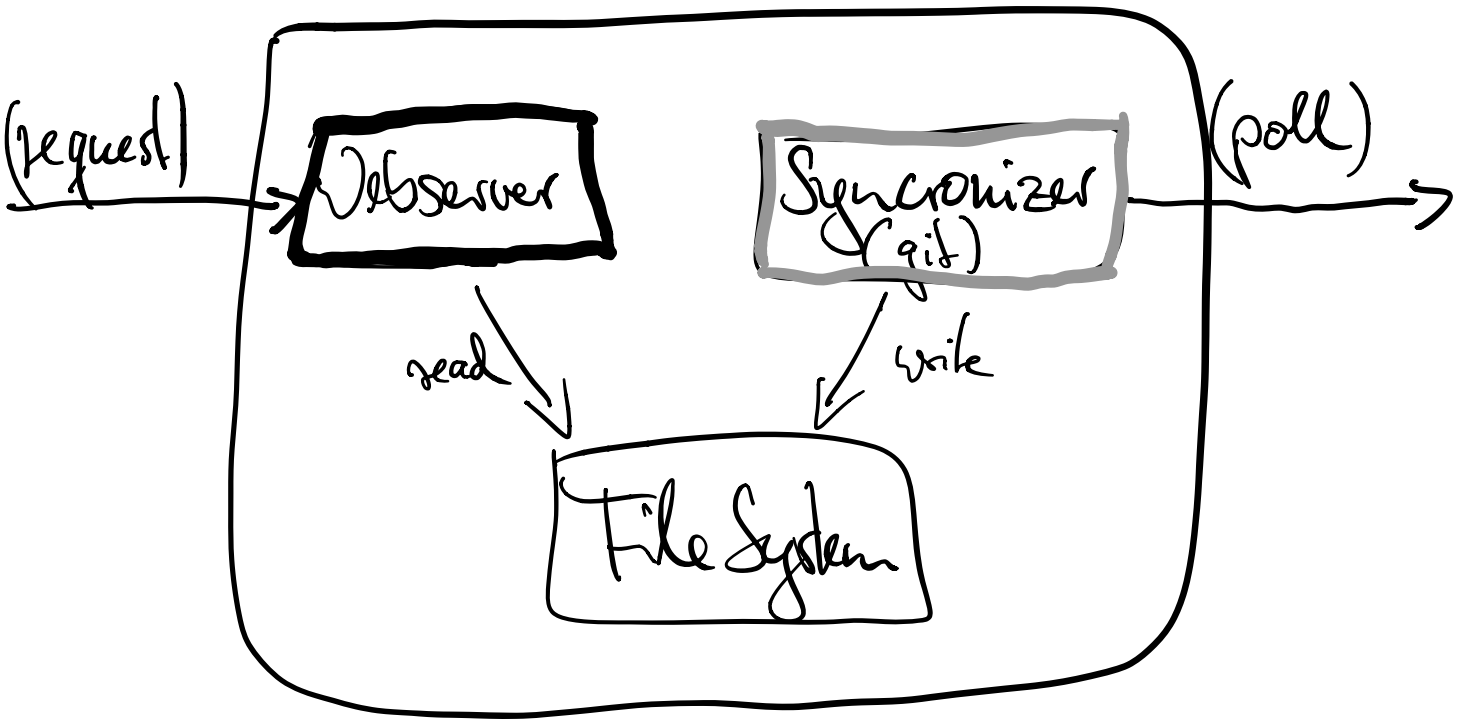
* init container
* Sidecar classic
* Sidecar adapter
* Sidecar ambassador

Init Container



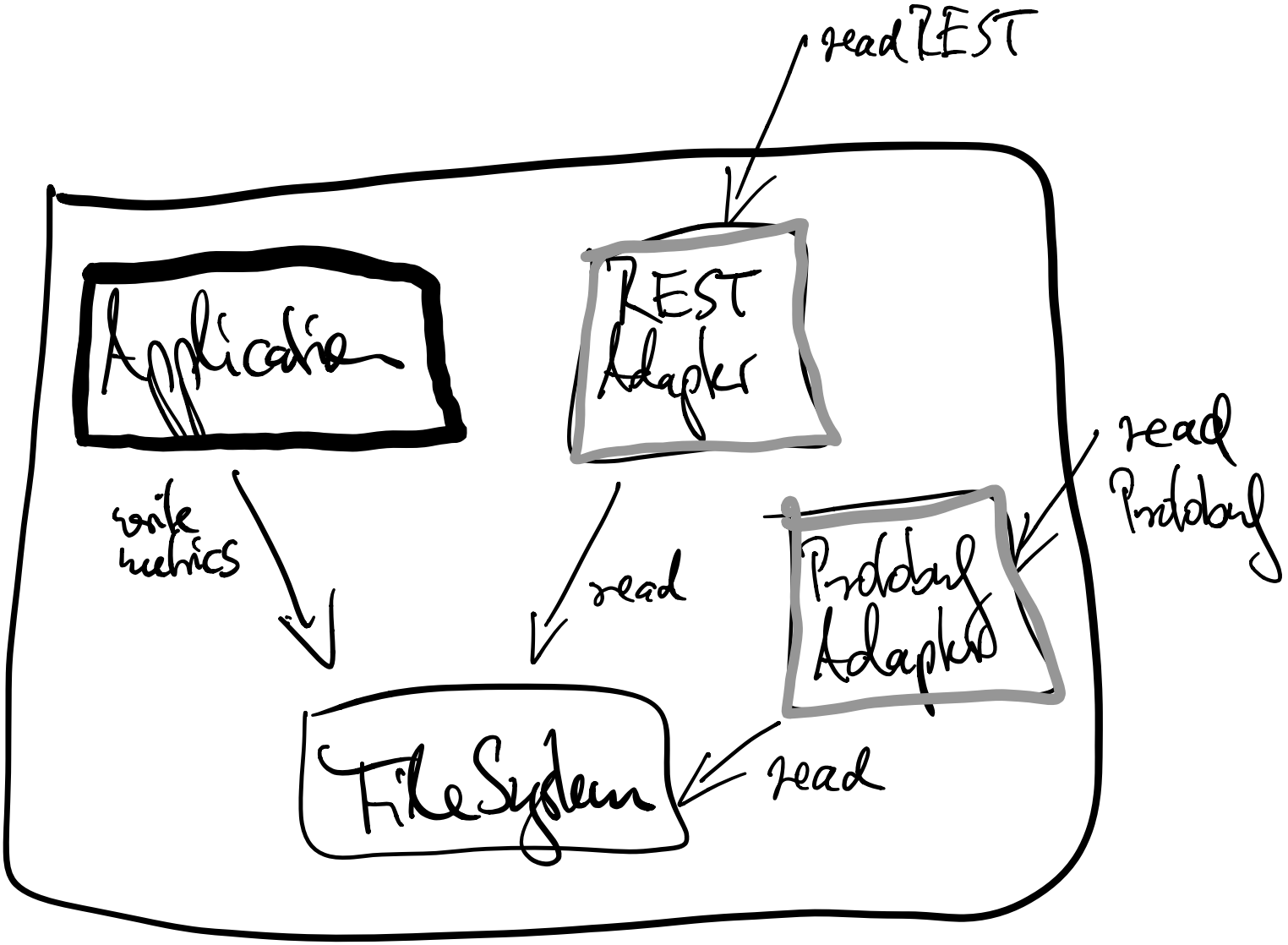
* main application can start after init container finishes
* Separation of concerns
  Separate lifecycle (initialization logic)

Sidecar classic



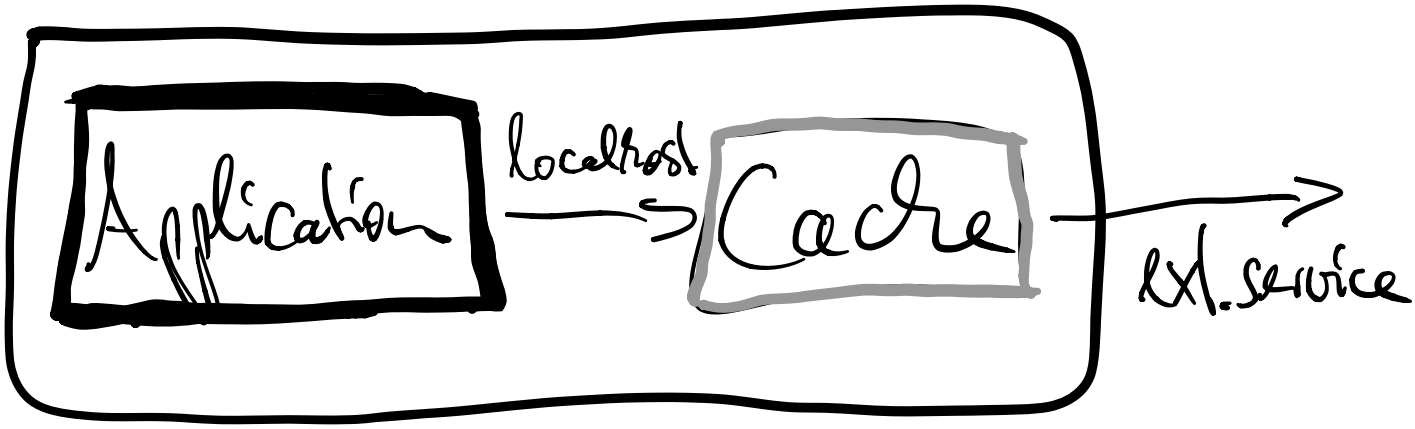
* extend/enhance main
* Single purpose
* no knowledge of the outside world

Sidecar Adapter



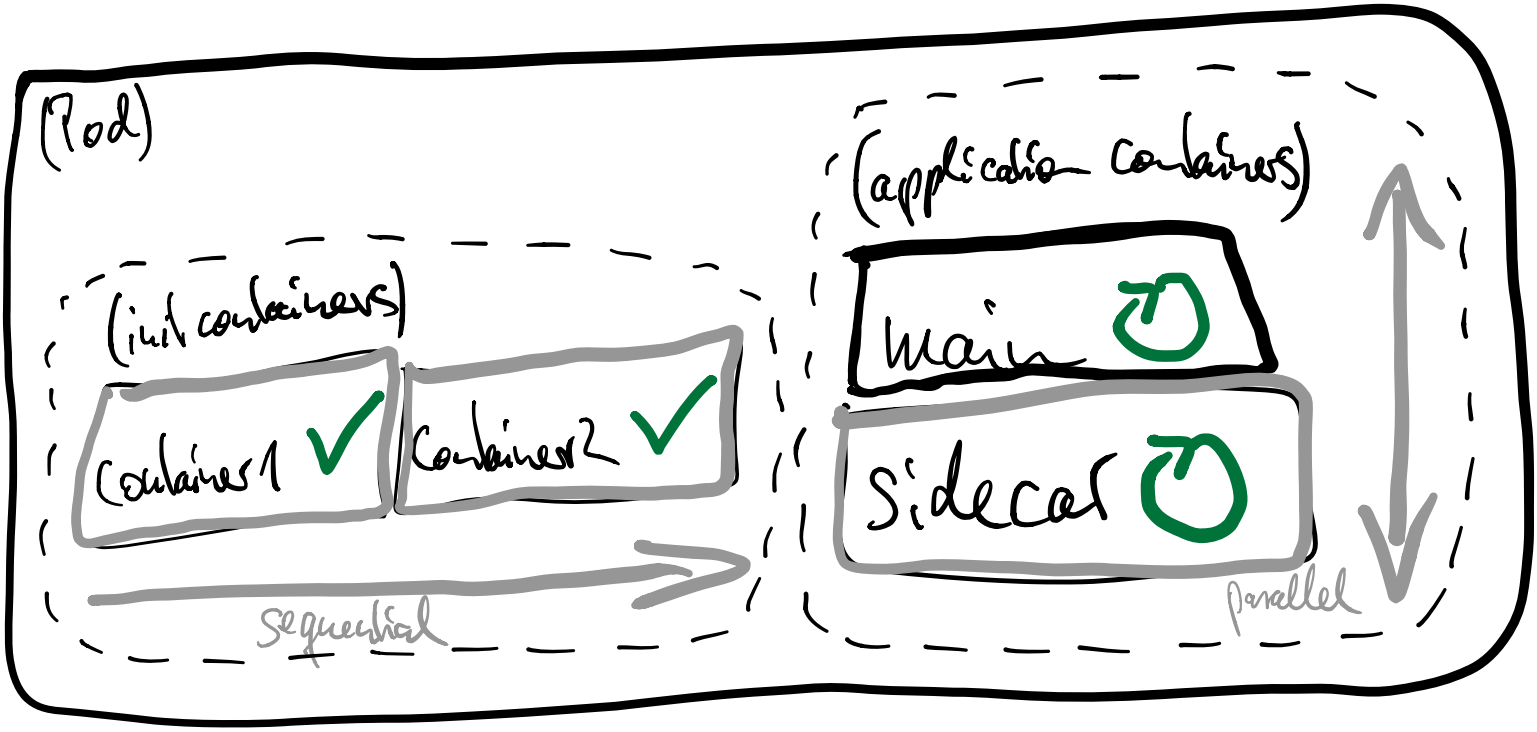
* provide unified interface to the outside world
* conform to a standard

Sidecar Ambassador



* hide complexity
* decouple external dependencies
* uniform interface to the inside world

Combine init + Sidcar



* init: sequential

* Sidecar: parallel

Monitoring

* Logs
* Metrics
* Traces

Good Practices:

* The RED Pattern (Requests - Errors - Duration)
  - > request driven, debug services

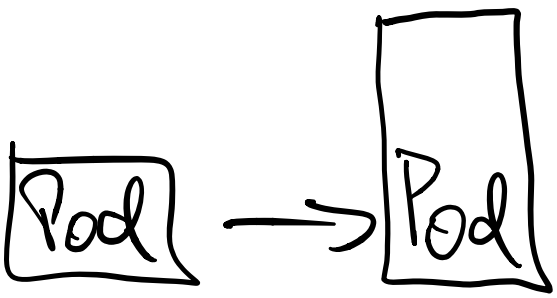
* The USE Pattern (Utilization - Saturation - Errors)
  (90% disk full) (10 jobs waiting) (25 ops failed)
  - > focus on resources, physical components, bottlenecks
    (CPU, MEM, network interfaces)

Scaling

* Node (cluster autoscaler - provider)

* Pod

- HPA: 

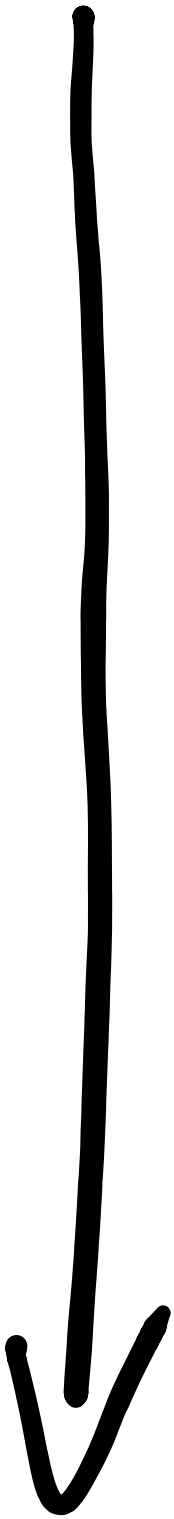
- VPA: 

DevOps

* CICD Pipeline
* Push vs Pull

DevOps

CI CD Pipeline



Code

↓

Build

↓

Test

↓

Package

↓

Configure/Release

↓

Deploy

↓

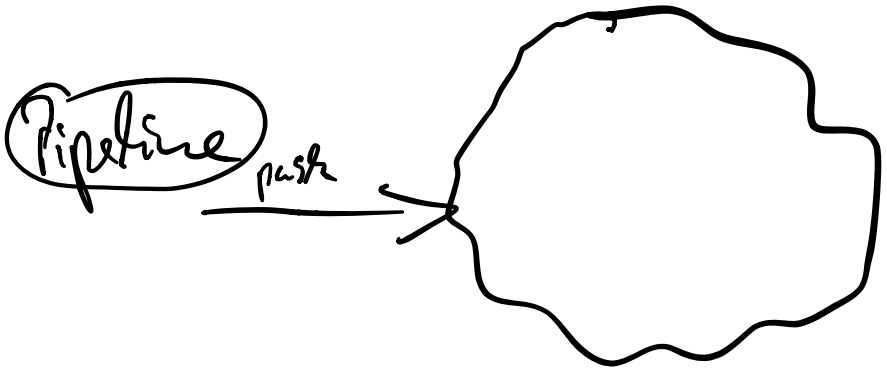
Monitor

* Automated
* Efficient
* Predictable

DevOps

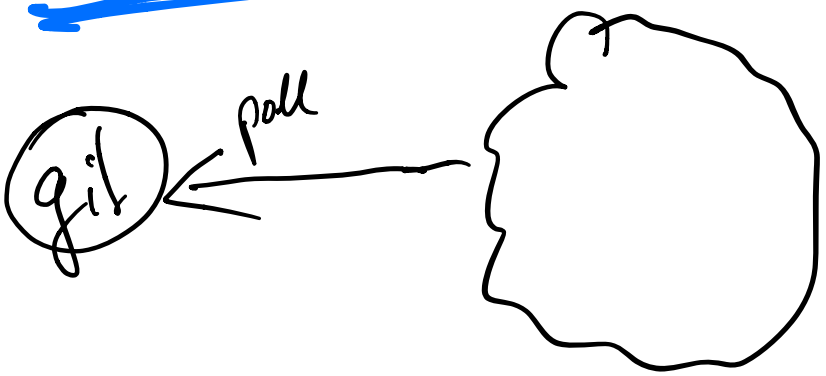
Push vs Pull

Push



* traditional
* the truth is in the running system
* imperative (commands)

Pull



* config as code
* the truth is in repository
* declarative (controllers)
* faster lifecycle (rollouts/rollbacks)

Volumes

* azureFile
* azureDisk

* static
* dynamic

Persistant Volume

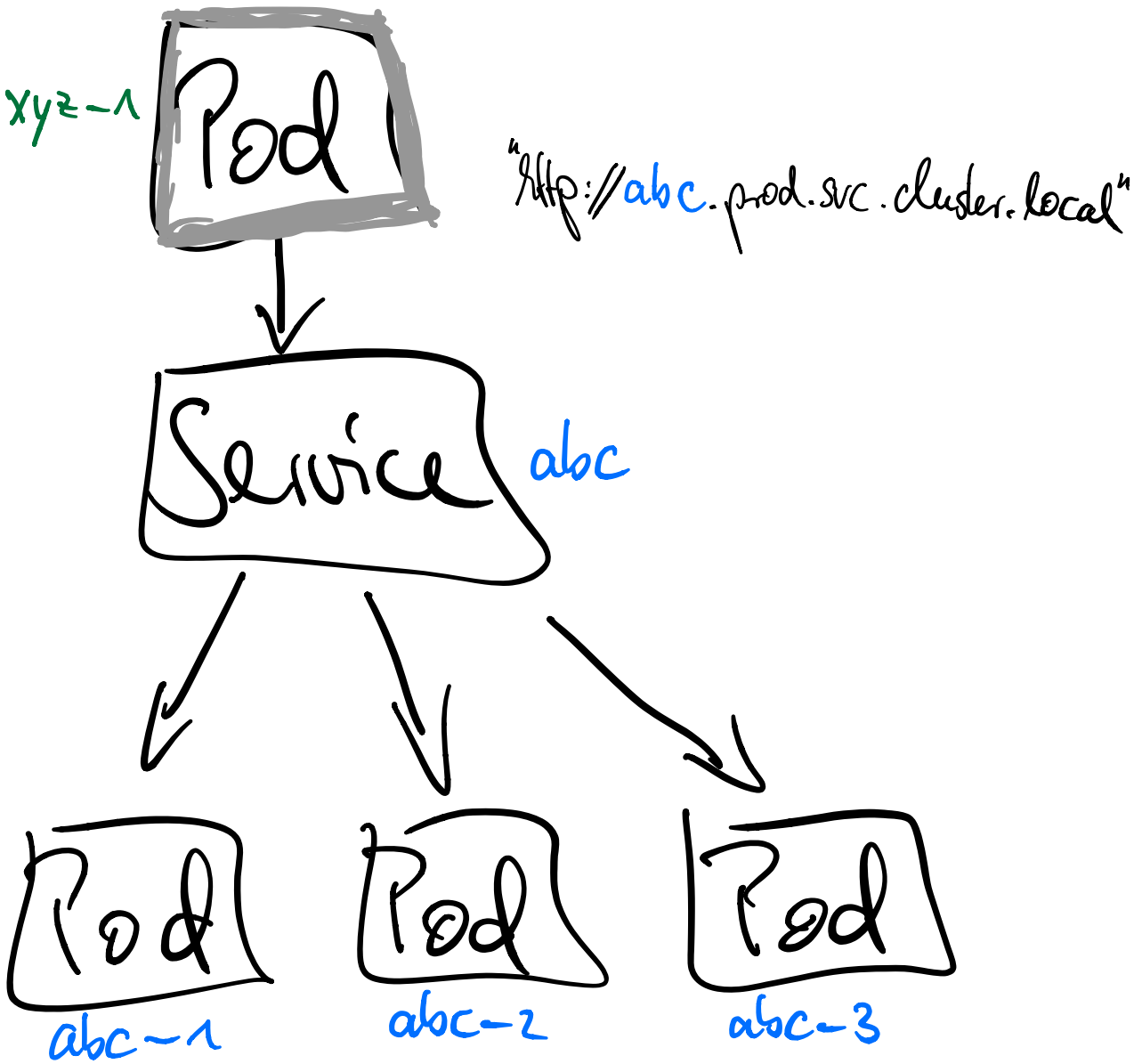
Persistant Volume Claim

Pod

Networking

* Internal Service Discovery
* External Service Discovery

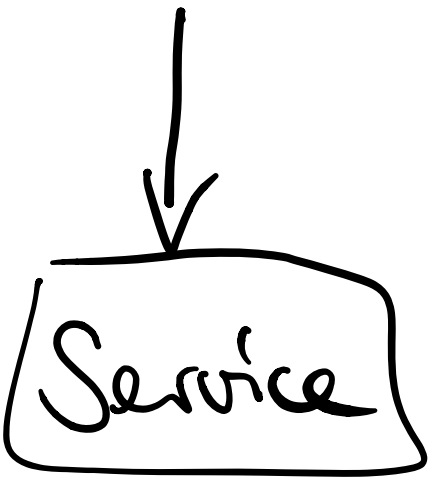
Kubernetes Internal Service Discovery



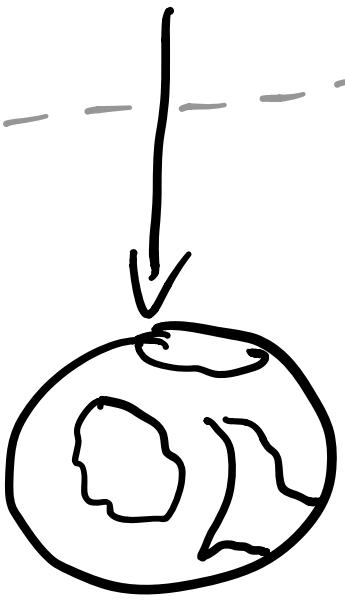
Networking
External Service Discovery

* kind: Service
* spec.type: ExternalName

http://test.database



http://myaccount.mlab.com:56789/mydb



Security

* Build Pipeline
* Container Content
* Container Runtime
* Operating System
* Orchestration Platform

Security

Build Pipeline

description:

- typically the least secure, security in development slows developers down
- malicious source code changes
- malicious changes to build scripts (expose credentials)

good practice:

- remove libraries and unneeded packages - reduces attack surface
- check libraries against the Common Vulnerabilities and Exposures (CVE)
- Code Analysis
- scan containers for security vulnerabilities

Container Contents

description:

- what is installed inside the container?
- open ports, security bugs in installed software

good practice:

- don't run container processes as 'root', narrow down roles and access privileges
- check for hard-coded passwords, keys, and other sensitive items in the container

Container Runtime

description:

- Think of Docker Daemon. PodSandbox.

good practice:

- limit external endpoints and who has access to the container
- use trusted image repository
- containers have a 'time to live' - new vulnerabilities may have been discovered.
- one image may instantiate into a thousand running containers - regularly patch and replace images and keep containers in sync
- enable AlwaysPullImages: images are always pulled with the correct authorization and cannot be re-used by other pods without first providing credentials.

Operating System

description:

- If an attack on the host platform succeeds it's pretty much game over for that cluster of containers, and may give malicious code sufficient access to attack other systems.

good practice:

- Host OS/Kernel Hardening: patch management

Orchestration Platform

description:

- incredibly complex
- focus on scalability and ease of management — not security
- insecure default configurations
- permission escalation
- code injection vulnerabilities

good practice:

- use latest Kubernetes version (critical bug fixes and new security features)
- isolate resources using namespaces
- use network policies: configure how components communicate among each other and with outside endpoints (pods, containers, service, namespaces) https://ahmet.im/blog/kubernetes-network-policy/
- use Network plugin (custom network interface - CNI): TLS, NSG, VNET
- use secrets: passwords, tokens or keys should always be stored in Kubernetes secrets objects (KeyVault)
- Authentication: use an identity provider (basic authentication, certificates, bearer token - Active Directory, OAuth)
- Authorization: use RBAC to limit access to Kubernetes Resources (ServiceAccount, Role, RoleBinding), avoid giving admin access as much as possible
- disable default service account: default service account is automatically assigned to new pods - has a wide range of permissions
- use PodSecurityPolicy: e.g. enforce pods are not able to run in privileged context, pods cannot bind to hostNetwork, pods must run as a particular user
- use securityContext in Pods/Deployments: runAsNonRoot, readOnlyRootFileSystem, allowPrivilegeEscalation, capabilities (drop:["all"], add["NET_BIND_SERVICE"])
- use ResourceQuota to limits resource consumption for each namespace (CPU, MEM, how many pods)

Monitoring

description:

- discover the unexpected stuff
- learn what is effective, track what's really happening in your environment, and detect what's broken.

good practice:

- enable Kubernetes audit logging: record a sequence of activities that users or system components perform
- enable Kubernetes logging: understand what is happening inside your cluster as well as debug problems and monitor activity

Discussion