Cloud & Containers
- Everything you need to know

Jack Henschel

2023-03-06

Overview

- **I. What is Cloud Computing?**
  - Motivation, Benefits, Drawbacks
- **II. How to use “the Cloud”?**
  - Deployment and access models
- **III. What are Containers?**
  - OS primitives and orchestration layers

I. Cloud Computing

What is Cloud Computing?

*“**Cloud computing** is a model for enabling **ubiquitous, convenient, on-demand network access** to a **shared pool** of **configurable computing resources** that can be **rapidly provisioned** and **released** with minimal management effort or service provider interaction.”*
— NIST

Essential Characteristics:

- **On-demand self-service**
- **Broad network access**
- **Resource pooling**
- **Rapid elasticity**
- **Measured service**

https://www.nist.gov/publications/nist-definition-cloud-computing

Why do we need a Cloud?

A “Cloud” decouples hardware and software by means of an API

- Virtual “machines” can be managed *programmatically* with API calls
- This also applies to other resources: access permissions, storage, containers etc.
- **Operators** vs. **Users**:
  - Operators are *responsible for the API*, but are not concerned with applications
  - Users *consume the API*, but are not concerned with the underlying hardware

Cloud Service Models

Flexibility, Control, Cost

SaaS
Software-as-a-Service

Translation
Machine Learning
Blob Storage

Containers
Serverless
Sandboxes

PaaS
Platform-as-a-Service

IaaS
Infrastructure-as-a-Service

Routers & Firewalls
Virtual Machines
Disk-based Storage

Ease of use, level of abstraction

Benefits of a Cloud

- **Pay-as-you-go pricing model**
- **Simplified (removed) resource management**
- **Scale quickly and effortlessly**
- **Flexible deployment options**
- **Improved resource utilization** (thanks to multiplexing)

Drawbacks of a Cloud

- **Abstraction layers (can) make troubleshooting difficult**
- **Vendor lock-in** (when using specific services or features)
- **Increased security risk** (?)

II. How to use “the Cloud”?

How to use a Cloud?

- Cloud enviroments strongly benefit from **declarative deployments**
  → **Reproducability** and **easy scalability**
- **Resources** in cloud environments are **expensive**! (CPU, memory, storage, network)
  → Use only what you need, scale up if necessary, scale down if possible
- Strategically decide if you want to use **cloud-specific features**
  → (substantially) **cheaper**, but at the cost of **vendor lock-in**
- **Access control** must be ***zero-trust*** (*because everything is online*)

What is cloud-native?

"Cloud native technologies empower organizations to build and run scalable applications in ***modern, dynamic environments*** *such as* ***public***, ***private***, *and* ***hybrid clouds***. ***Containers***, ***microservices***, ***immutable infrastructure***, *and* ***declarative APIs*** *exemplify this approach.*
These techniques enable ***loosely coupled systems*** *that are* ***resilient***, ***manageable***, *and* ***observable***.
Combined with ***robust automation***, *they allow engineers to make* ***high-impact changes frequently*** *and* ***predictably.***" — CNCF

- **Resiliency**: failures are treated as the norm; applications take advantage of dynamic environment ("adapt") and can recover from failure
- **Agile:** quick deployment and update cycles leveraging cloud-native infrastructure
- **Observability:** allow operators to reason about the application state
- **Operability:** it is easy to manage the application during its lifetime (not just during deployment); facilitated by using industry standard tools, no weird hacks to keep the application happy

What is cloud-native NOT about?

These are **not defining features** of cloud-native:

- Running applications in a public cloud (example: *“lift-and-shift”*)
- Running applications in containers
- Using a fancy container orchestrator, e.g. Kubernetes
- Microservices (*monolithic applications can also be cloud-native*)
- Infrastructure as Code (*Chef, Puppet etc. are IaC but not cloud-native*)

Best practices

*“Cloud-native is a term describing software designed to run and scale reliably and predictably on top of potentially **unreliable** cloud-based infrastructure."*
— Duncan Winn

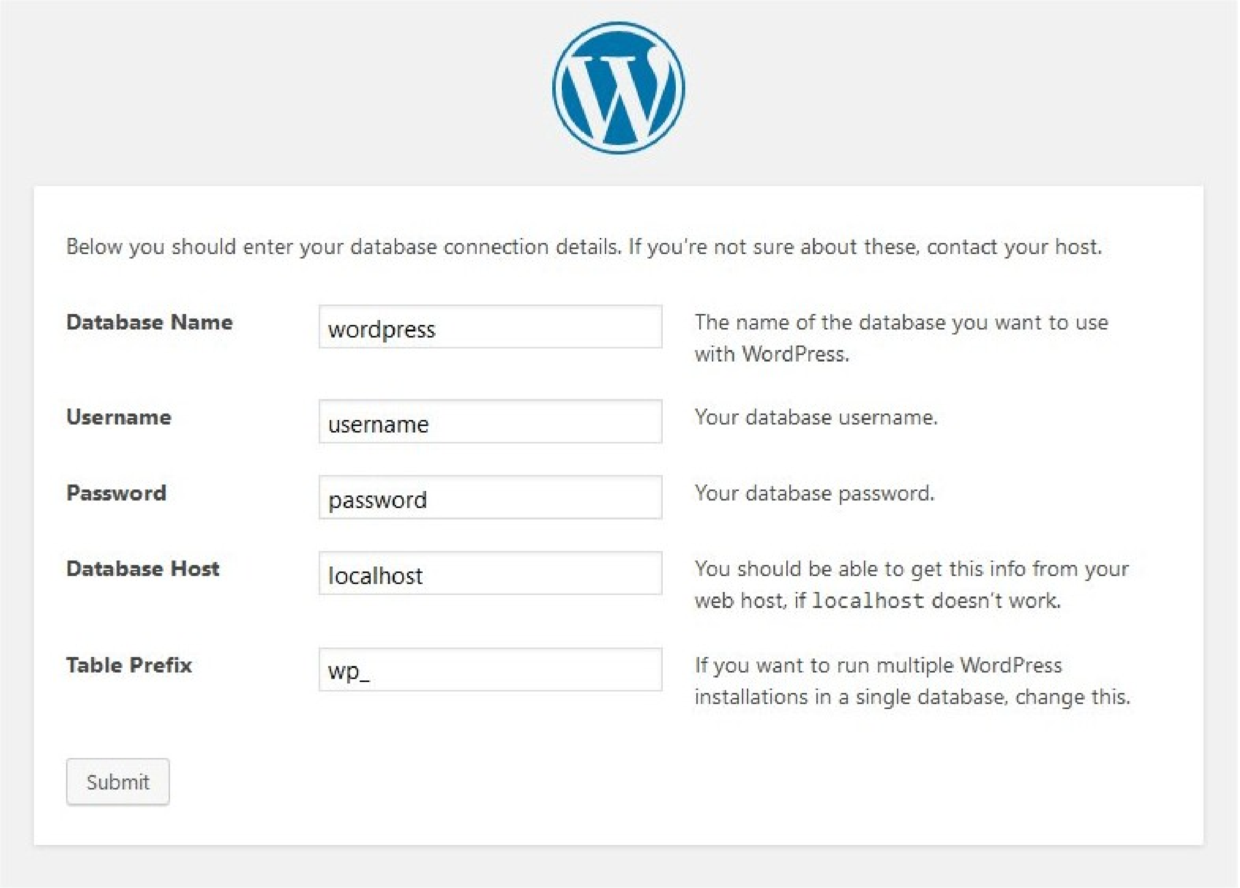
- Implement **retry logic** and make actions **idempotent**
- Don’t assume that application state will be **persisted** in memory
- Make as **few assumptions** as possible about the **runtime environment**
- Ensure that the application can be **configured declaratively** and **restarted reproducible**
- Don’t hard-code configuration (→ ideally use **service discovery**)
- Allow **multiple-readers/multiple-writers**

Practical examples (what not to do)

Interactive configuration is taboo

→ use declarative configuration from files or environment variables instead

(and make sure not to persist them in the DB!)



Practical examples (what not to do)

Do not rely on specific runtime / environment

(or at least document all dependencies and assumptions of your app)

→ makes it easy to run the application in different environments (local development vs. production) and makes it more portable

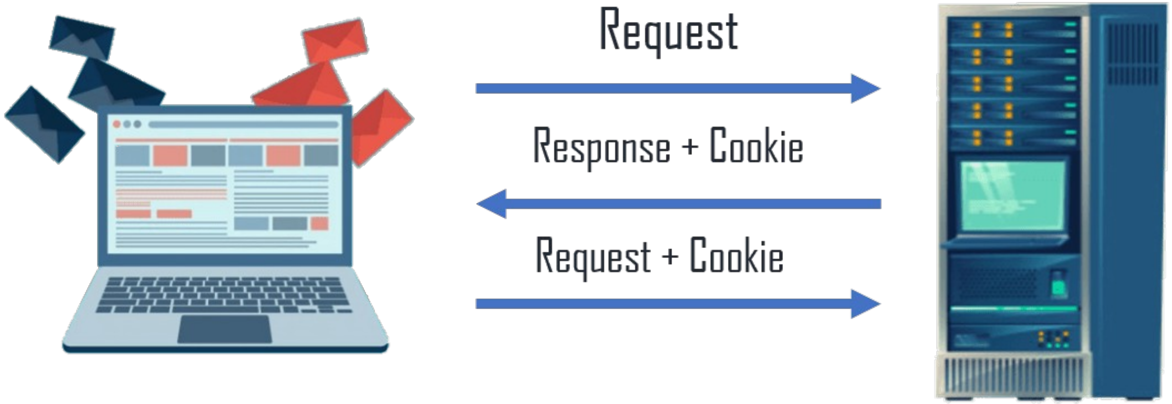

```
# Install well-known SSH host keys for GitLab
COPY ./contrib/ssh /etc/ssh

# install extra plugins and ensure permissions are correct everywhere after adding our own files
# Plugins go to /opt/openshift/plugins in the image, then on first run they'll be copied to /var/lib/jenkins
RUN /usr/local/bin/install-plugins.sh /opt/openshift/extra_plugins_no_overwrite.txt && \
    /usr/local/bin/install-plugins.sh /opt/openshift/extra_plugins_overwriten_at_startup.txt && \
    chown -R 1001:0 /opt/openshift && \
    /usr/local/bin/fix-permissions /opt/openshift && \
    /usr/local/bin/fix-permissions /var/lib/jenkins
```

Practical examples (what not to do)

Avoid "process-local" state

All state must be persisted in external systems (database, shared storage, shared cache)

→ enables horizontal scaling and fault tolerance

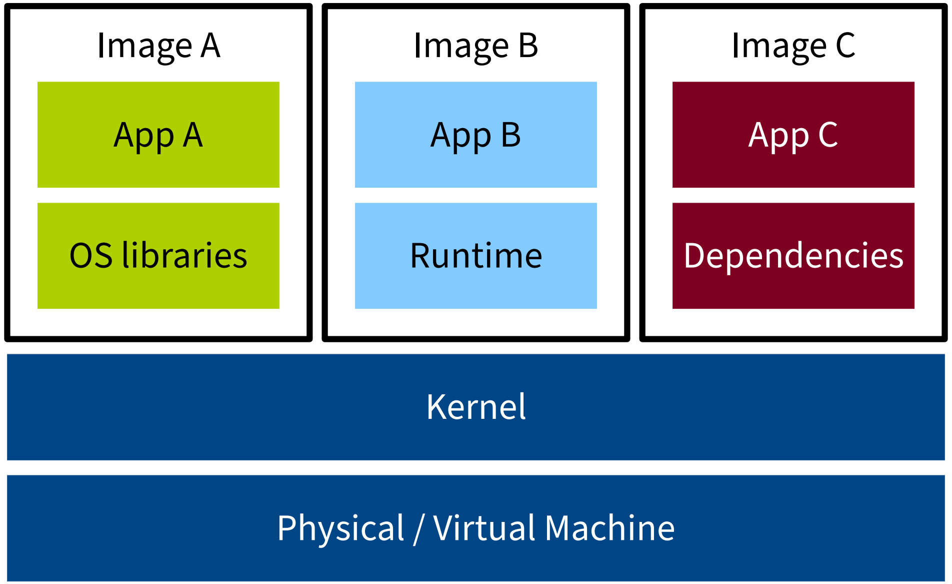


III. Containers

What are Containers?

A.k.a. ***"OS-level virtualization"***

Multiple processes share the **same kernel**, but have an **isolated environment**: compute, memory, storage, networking



A brief history of containers

There has always been a desire to divide a single computer into smaller, flexible units:

~1960: Multi-tasking operating systems: *time-sharing*

1982: Restricting filesystem access of a process: *Unix chroot*

2000: Additional isolation and security: FreeBSD *Jails*

2002: First-class citizen: Solaris *Zones*

2008: Linux kernel gains equivalent features: *Linux Containers (LXC)*

2013: Docker launches easy-to-use tools for interacting with containers

Containers for everyone

Docker allows managing the *entire* life cycle of a container:

- **building** an image from a set of instructions (Dockerfile)
- **sharing** this container image over the Internet (DockerHub)
- **creating**, **running** and **deleting** containers based on images (Docker Daemon & CLI)

→ **modern containers**!

Containers provide a higher level of abstraction for the application lifecycle (starting/stopping, upgrades, replication, scaling)

Containers for everyone

Dockerfile:

```
FROM node:18-alpine
COPY src/ /app
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

Application lifecycle:

```
docker image build -t my-app .

docker container run my-app \
 --name=app-1 --publish 3000:3000

docker container logs app-1

docker container stop app-1

docker container rm app-1
```

Container Orchestration

We have so many containers … now what?

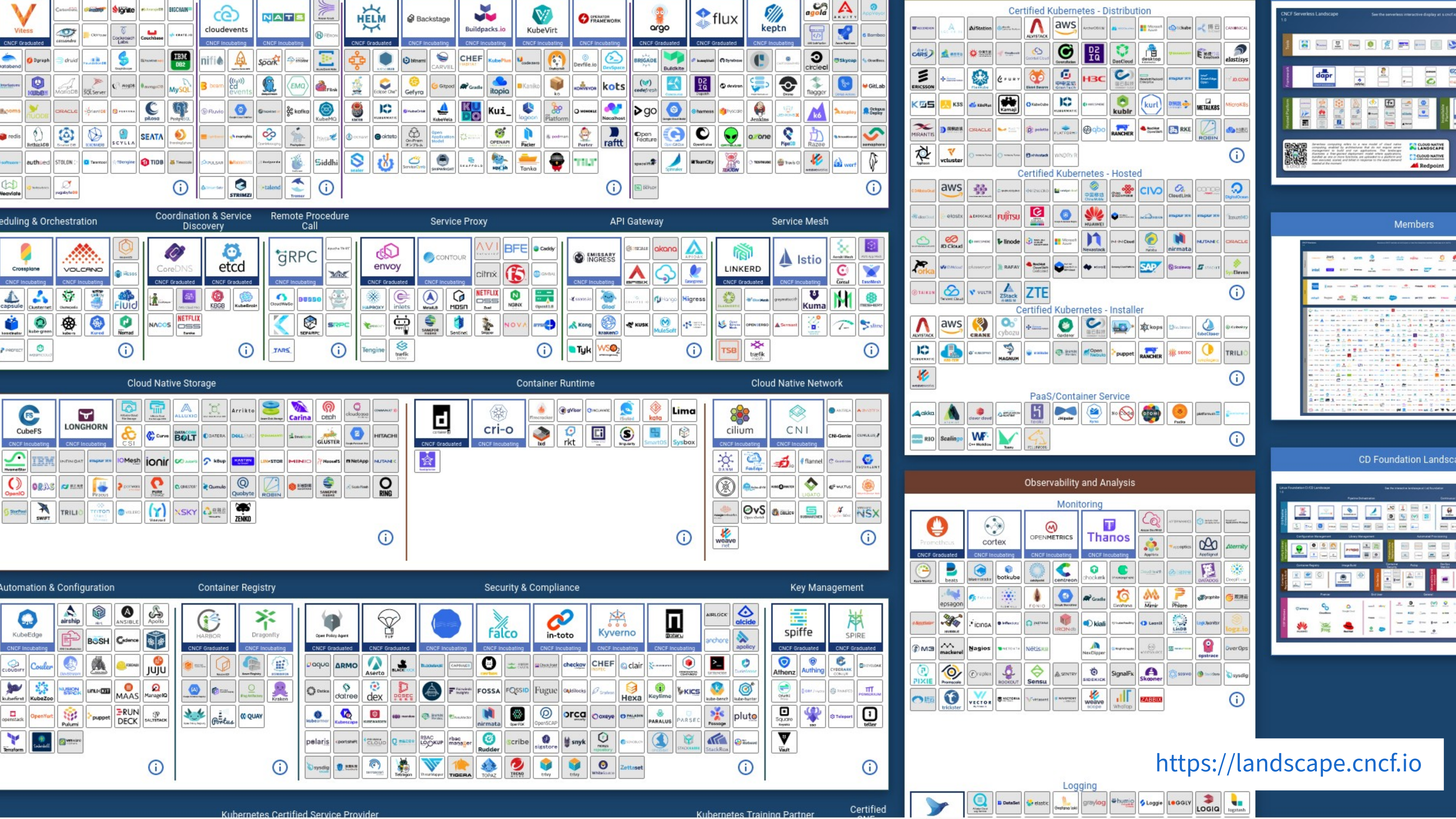
An **Orchestrator** automates the deployment, management, scaling, and networking of containers

2014-2017: *Orchestration Wars* (Apache Mesos, Docker Swarm, HashiCorp Nomad, Kubernetes)

Kubernetes became the **de-facto** standard (on-premises and public cloud)

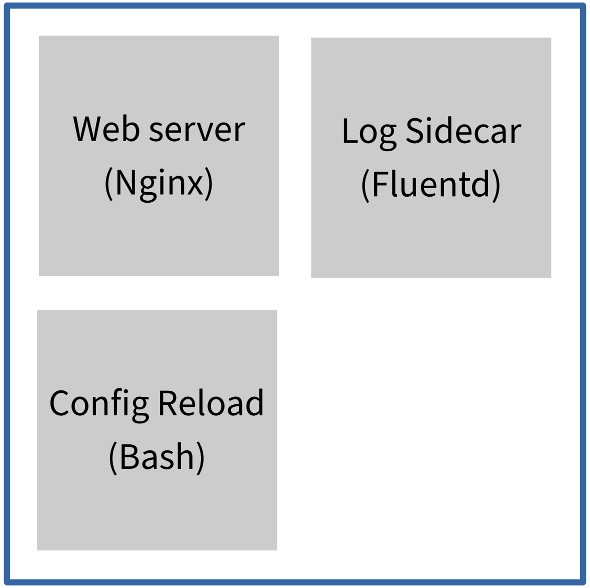
What makes Kubernetes so powerful?

- **Simple, declarative** and **extensible API**
- **Continuous resource consiliation**
- **Design** and **experience based on Google's internal Borg task scheduler**
- **Flexible** and **extensible** (container runtime, networking etc.)
- **Active** and **open community**
- **Comprehensive software ecosystem**


Scheduling & Orchestration
Coordination & Service Discovery
Remote Procedure Call
Service Proxy
API Gateway
Service Mesh
Cloud Native Storage
Container Runtime
Cloud Native Network
Automation & Configuration
Container Registry
Security & Compliance
Key Management
Kubernetes Certified Service Provider
Kubernetes Training Partner
Certified Kubernetes - Distribution
Certified Kubernetes - Hosted
Certified Kubernetes - Installer
PaaS/Container Service
Observability and Analysis
Monitoring
Logging
Members
CD Foundation Landscape
CNCF Serverless Landscape
https://landscape.cncf.io

Kubernetes 101: Pods

- **Smallest unit of compute in Kubernetes**
- **Composed of multiple containers (one process/function per container)**
- **Shared networking stack**
- **All containers in the pod run on the same host**
- **Ephemeral!**



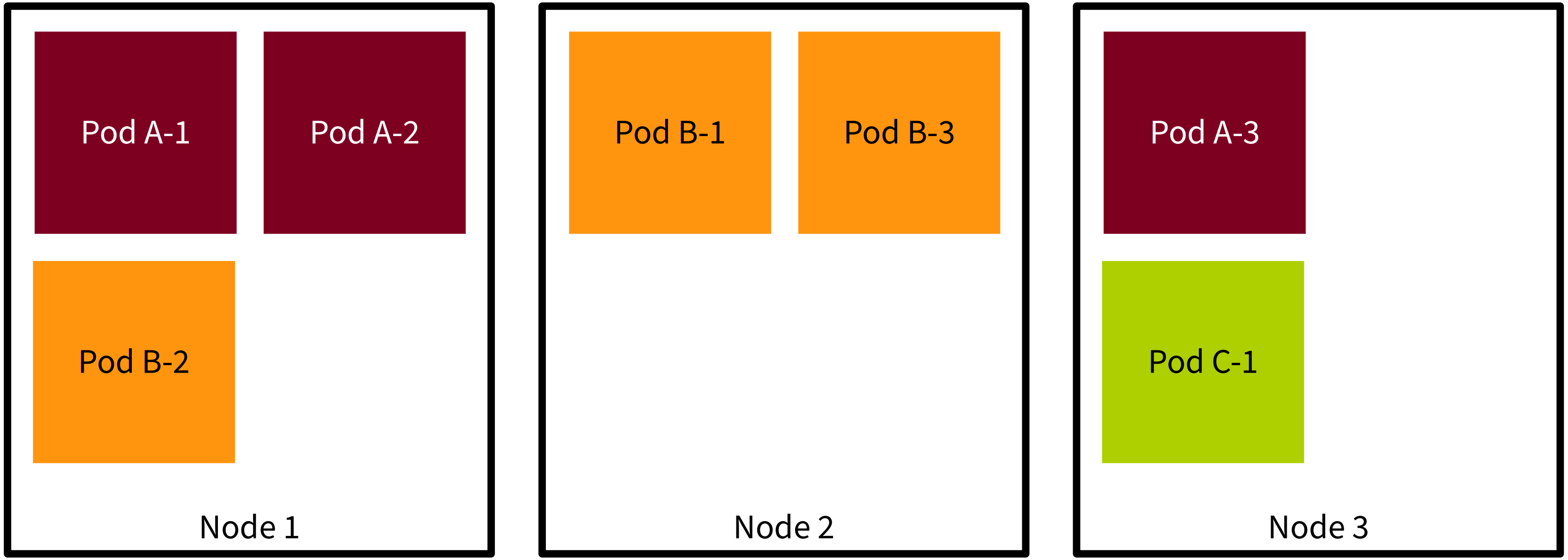
Kubernetes 101: Pods

```
# kubectl run my-server --image docker.io/library/nginx
# kubectl get pod my-server -o yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-server
spec:
  containers:
  - image: docker.io/library/nginx
    imagePullPolicy: Always
    name: my-server
  nodeName: test-jack-r7smtryv6dfb-node-0
status:
  conditions:
  - lastTransitionTime: "2023-02-03T15:24:42Z"
    status: "True"
    type: ContainersReady
  phase: Running
  podIP: 10.100.180.137
```

Kubernetes 101: Deployments

Abstraction to continously run and scale ephemeral pods (of the same kind)

Each pod is fully independent and isolated from the others



Other ways to scale pods on Kubernetes: *StatefulSets*, *DaemonSets*

Kubernetes 101: Deployments

```
# kubectl create deployment my-cache --image docker.io/library/redis --replicas=3
# kubectl get deploy my-cache -o yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-cache
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-cache
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: my-cache
    spec:
      containers:
      - image: docker.io/library/redis
        name: redis
status:
  availableReplicas: 3
```

Kubernetes 101: Services

Provides a stable endpoint (IP) for a collection of pods (usually replicas of a Deployment)

Services are a pure network abstraction

Pods are selected via labels

Endpoint can be internal (*ClusterIP*) or external (*LoadBalancer*)

```
# kubectl create service clusterip my-cache
--tcp=6379
# kubectl get svc my-cache -o yaml
apiVersion: v1
kind: Service
metadata:
  name: my-cache
spec:
  clusterIP: 10.254.52.144
  ports:
  - name: "6379"
    port: 6379
    protocol: TCP
    targetPort: 6379
  selector:
    app: my-cache
  type: ClusterIP
status:
  loadBalancer: {}
```

Kubernetes 101: Ingress

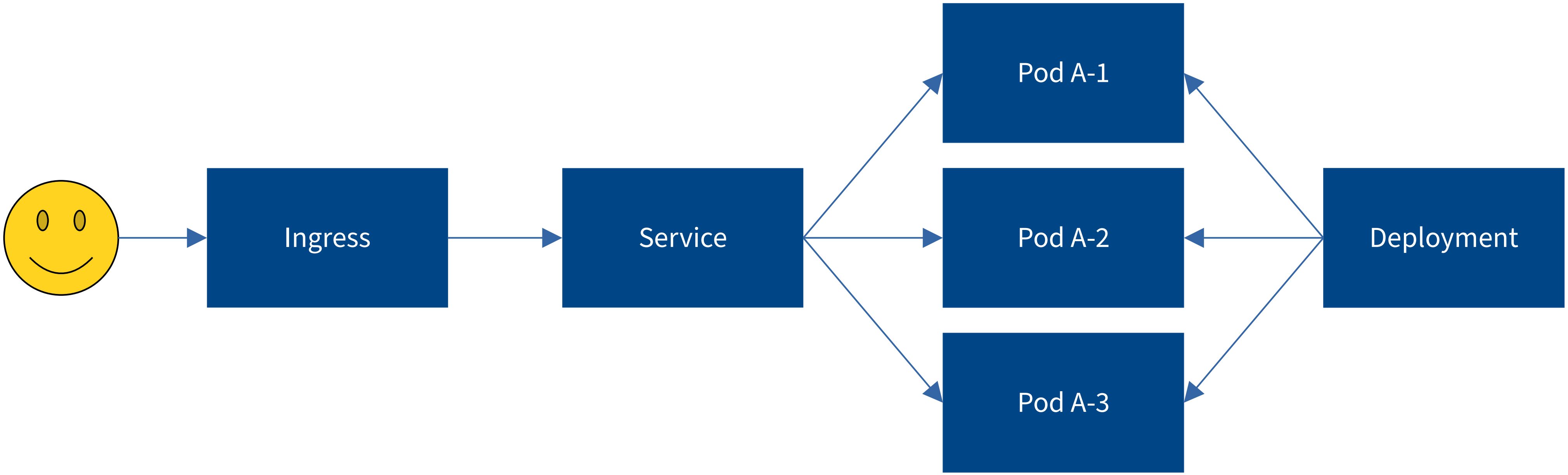
Provides an external entrypoint for HTTP services

Layer 7 based → "smart"

Offers routing, API gateway, TLS termination, certificate renewal ...

```
# kubectl create ingress my-http \
--rule="example.com/*=my-server:80"
# kubectl get ingress my-http -o yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-http
spec:
  rules:
  - host: example.com
    http:
      paths:
      - backend:
          service:
            name: my-server
            port:
              number: 80
        path: /
```

Overview – *what have we just built?*

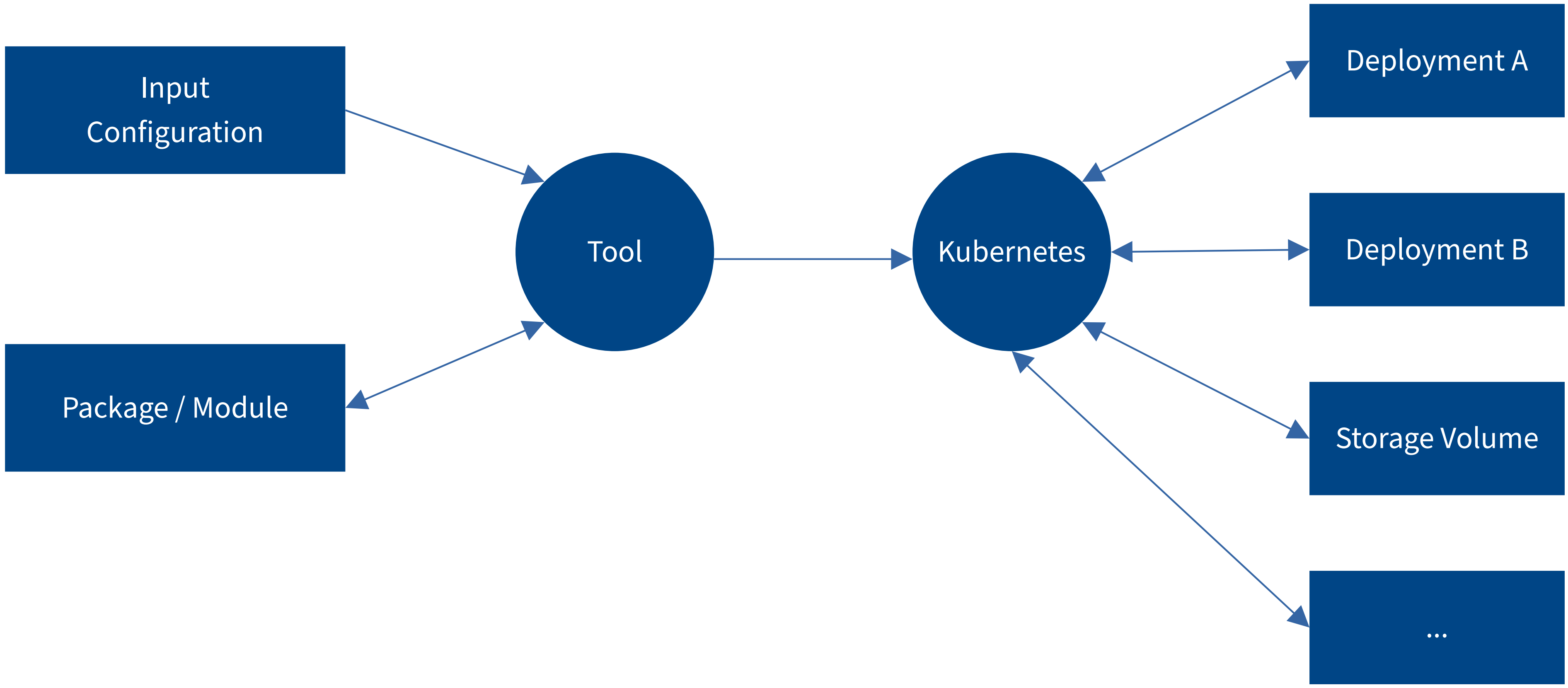


Deploying on Kubernetes

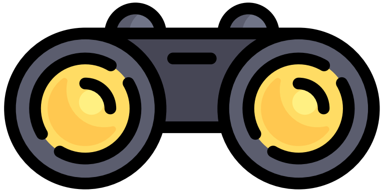
- **kubectl**
- **Kustomize**
- **Helm**
- **Flux**
- **ArgoCD**

+ many more that wrap around these tools (Terraform, Pulumi, Ansible ...)

Deploying on Kubernetes: Under the Hood



Cloud-native Observability



- Cloud-native **environments change rapidly** – static configuration à la Nagios doesn’t cut it
- The monitoring tool needs to be able **to automatically (re-)configure** itself
  → ***Service discovery***
- Different dimensions:
  metrics, **logs**, **tracing**, **alerting**

Metrics & Prometheus

Time-series database + query language + automatic ingestion

- **OpenMetrics:** the *exposition format*

```
http_requests_total{instance="pod-abc123",method="post",code="200"} 1027
http_requests_total{instance="pod-abc123",method="post",code="400"}  420
```

- Prometheus regularly *scrapes* this exported data (HTTP request + ingestion)
- Extremely efficient storage of time-series values (1-2 bytes per **sample**)

PromQL example

https://cern.ch/csc

https://indico.cern.ch/e/iCSC-2023