Microservices #2

Training overview

1. **Go basics**

   Syntax, data structures, interfaces, ...

2. **Go basics**

   Best practices, profiling, docker, rest APIs, ...

3. **Microservices**

   Monoliths, containers, Kubernetes, packaging, ...

4. **Microservices**

   CI/CD, skaffold, logging, monitoring, troubleshooting, ...

5. **Workshop**

   Building microservices with Go

Part 3

- Distributed computing
- Containers
- Docker
- Exercise 1

Part 4

- Kubernetes
- Exercise 2
- Skaffold
- Helm
- Testing

Distributed computing

What is a distributed system?

- Components on different networked computers
- Communicates and coordinates by passing messages

Why is it relevant?

- Microservice architecture is a distributed system
- Comes with its own set of challenges

Challenges?

1. Lack of a global clock

- Transactions inserted at different times
- Which one is first or latest?

2. Synchronization

- Multiple instances process the same transactions?

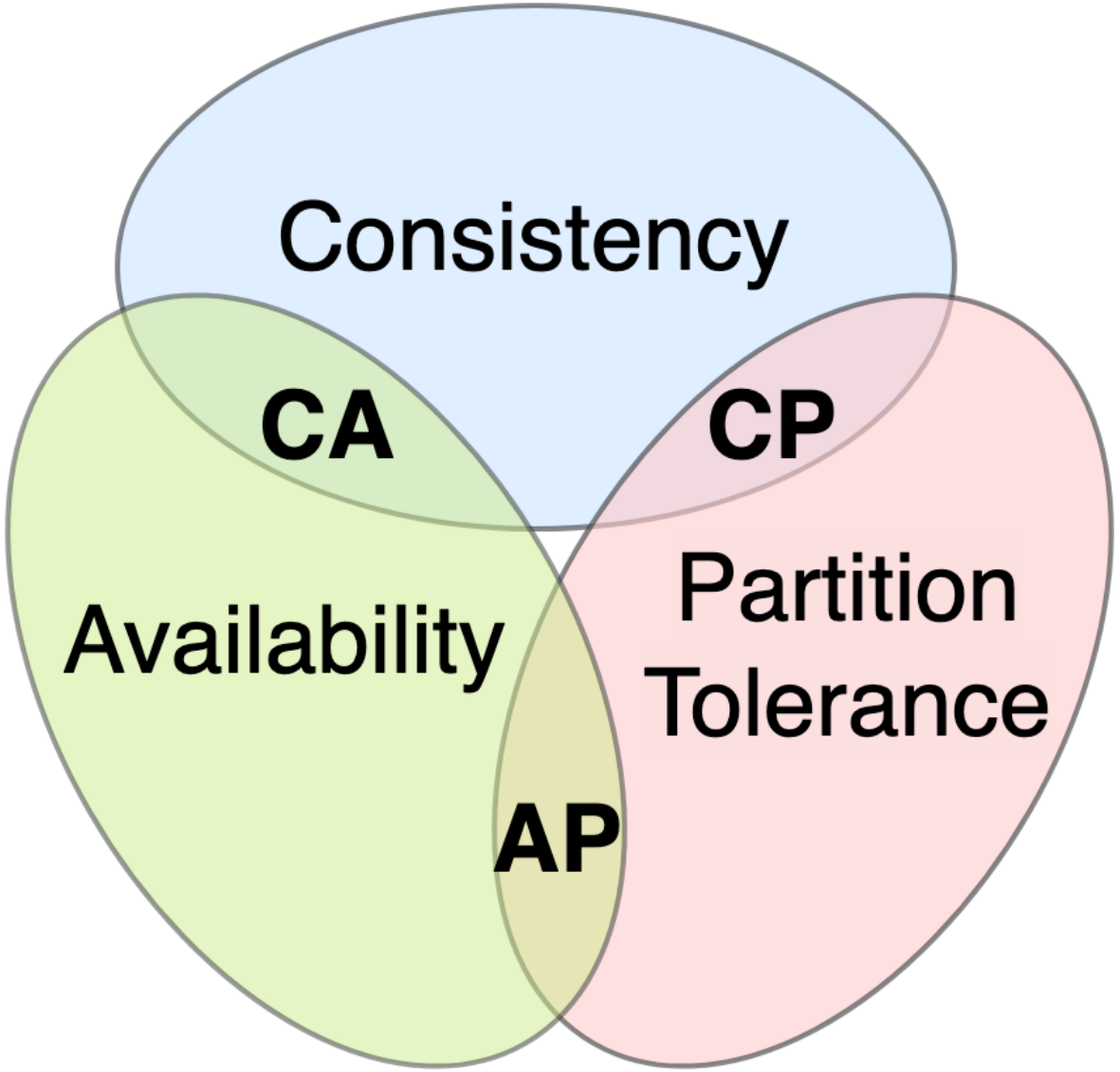
3. Independent failures

- What if one of the nodes fail?
- Or what if multiple nodes fail at the same time?

CAP theorem - 1/5

Any distributed system can only provide two out of three guarantees:

1. Consistency
2. Availability
3. Partition tolerance



CAP theorem - 2/5

Consistency

- Every read receives the most recent write, or an error
- All nodes see the same data, at the same time

CAP theorem - 3/5

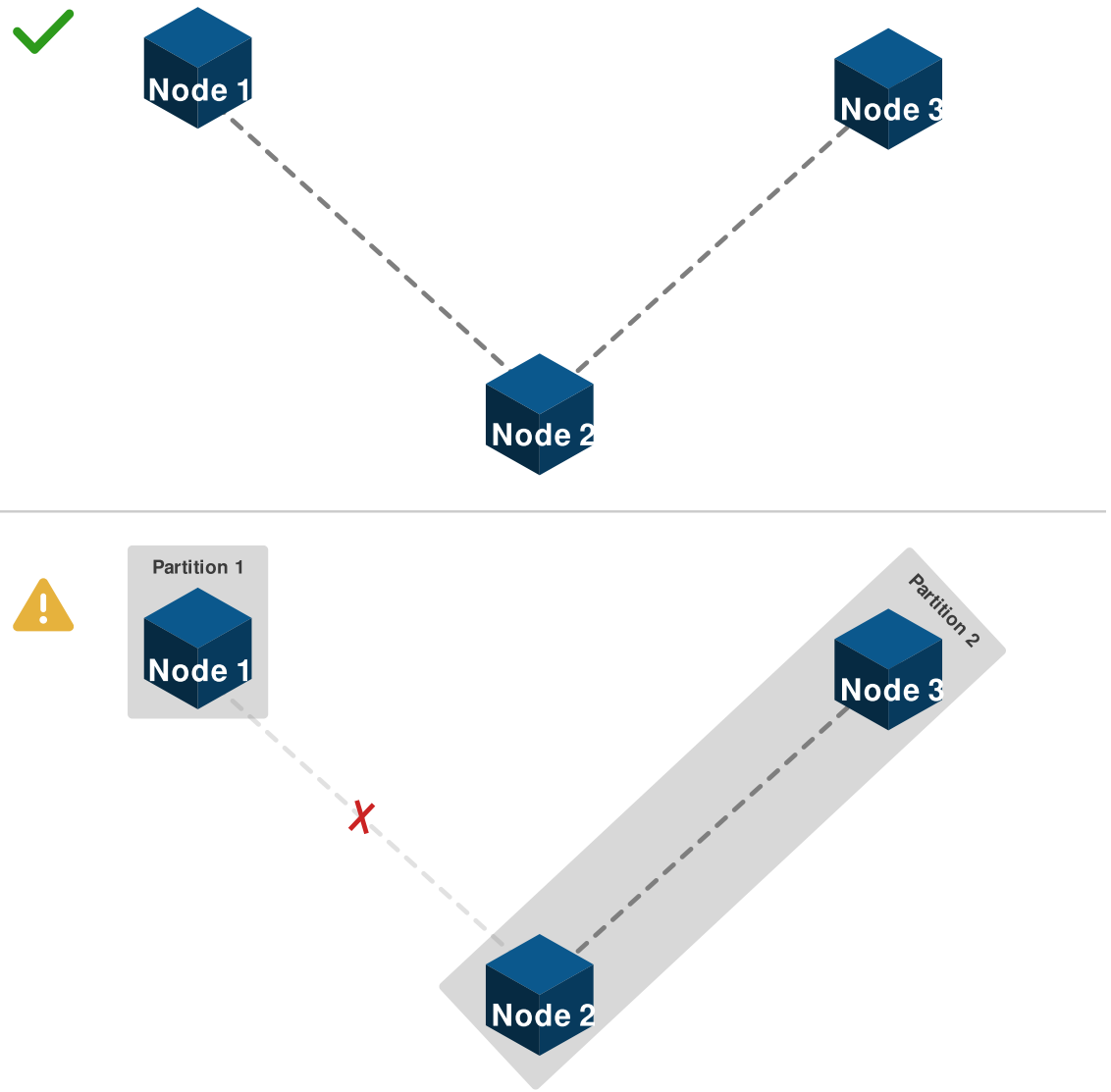
Availability

- Every request receives a (non-error) response
- No guarantee that it contains the most recent write

CAP theorem - 4/5

Partition tolerance

- The system continues to operate, on network partition failures
- Even if any number of messages are dropped (or delayed)



CAP theorem - 5/5

Summary

- Distributed systems need to deal with network partition failures
- When this happens either consistency or availability is lost

Eventual consistency - 1/2

- Consistency model to achieve high-availibility
- While maintaining a high degree of consistency
- Changes are propagated asynchronously
- Eventually, all systems see the same data

CAP?

What do we think?

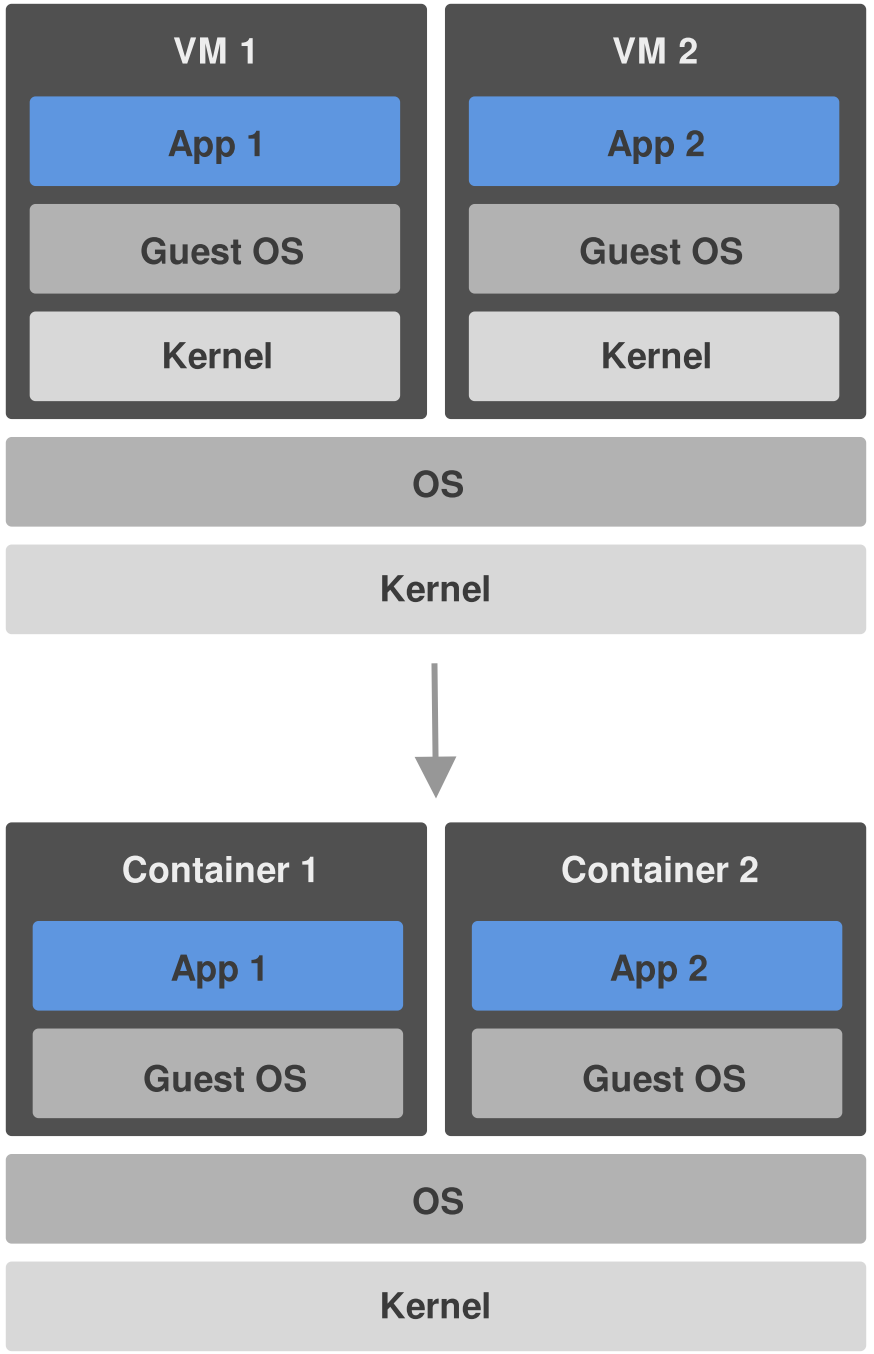
Eventual consistency - 2/2

- Answer is: AP (Availability and Partition tolerance)
- Provides high-availablibility and partition tolerance over consistency
- Consistency is achieved over time, not immediately

Containers

What is a container?

- OS-level virtualization
- Share the host kernel
- Similar to virtual machines (more lightweight)
- No emulation involved



Advantages

- Lightweight isolation, improved security
- Fast startup time, less resource usage
- Solves packaging problems (e.g. dependency hell)
- Improved portability ("it works on my machine!")

Comparison

Virtual machine

Container

History

- 1979 - Unix V7, introduction of chroot
- 2000 - FreeBSD Jails, proper isolation
- 2006 - Process containers, later renamed to cgroups
- 2008 - LXC, container management with cgroups and namespaces
- 2013 - Docker, originally based on LXC
- 2015 - OCI, standardization of container runtime
- 2017 - Containerd, donated to CNCF
- 2018 - Firecracker, lightweight VMs, better isolation

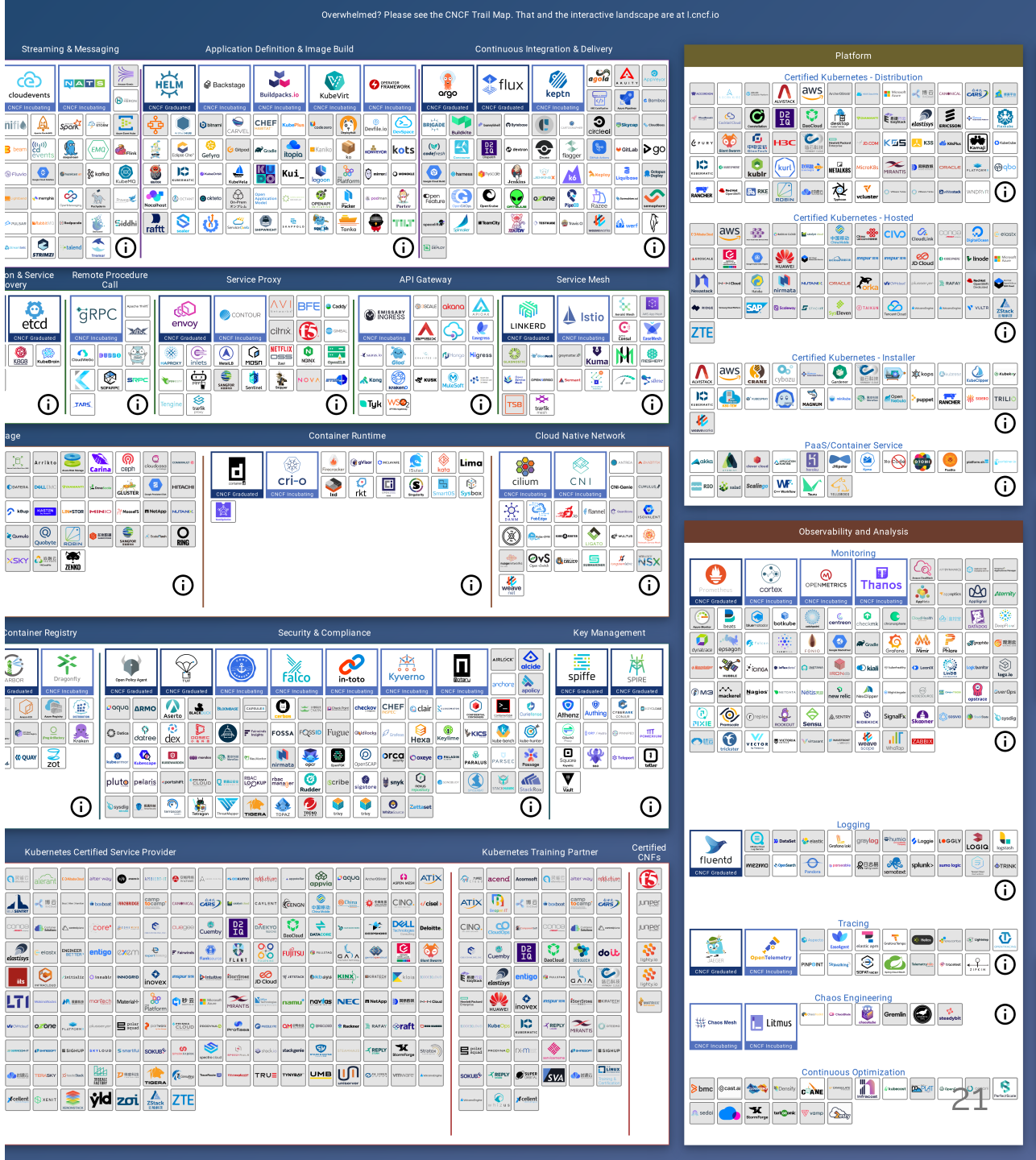
CNCF - 1/2

- Cloud Native Computing Foundation (part of the Linux Foundation)
- Hosts many cloud-native/Kubernetes open source projects
- In different maturity levels:
  - **Sandbox**; experimental, not production ready
  - **Incubation**; runs successfully in production by a few early adopters
  - **Graduated**; stable, production ready and widely used

CNCF - 2/2

Graduated projects

- Argo CI/CD
- CoreDNS
- Containerd
- Envoy Service Proxy
- Kubernetes
- Prometheus
- *And more...*



OCI

- Open Container Initiative (part of the Linux Foundation)
- Mission: minimal open standards and specifications for container technology
- Current specifications:
  - Runtime Specification
  - Image Specification
  - Distribution Specification

Container images - 1/3

- Image can be build from instructions
- Instructions are defined in a Dockerfile
- Each instruction creates a new layer
- Layers are cached, speeding up builds

Container images - 2/3

Build flow

1. Dockerfile is parsed
2. Context is sent to the daemon (e.g. source code)
3. Each instruction is executed, creating new layers
4. Image is stored locally

Container images - 3/3

Dockerfile

```
FROM golang:1.20-alpine

WORKDIR /dist

COPY . .

RUN go build -o /dist/server ./cmd/matrix-gin

CMD ["/dist/server"]
```

Breakdown - 1/5

```
FROM golang:1.20-alpine
#    ^^^^^^ ^^^^^^^^^^^
#    Image  Tag
```

Summary

Start with `golang:1.20-alpine` as the base image, where:

- `golang` is the image name
- `1.20-alpine` is the image tag
- Registry is empty so defaults to: `docker.io`

Breakdown - 2/5

```
WORKDIR /dist
#       ^^^^^
#       Path name
```

Summary

- Set `/dist` as the working directory
- For future `COPY` and `RUN` instructions

Breakdown - 3/5

```
COPY . .
#  Src Dest
```

Summary

- Copy the current directory, to the working directory

Breakdown - 4/5

```
RUN go build -o /dist/server ./cmd/matrix-gin
#   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#   Command
```

Summary

- Runs `go build` in the current directory
- Output is stored in `/dist/server`
- After execution, a new layer is created

Breakdown - 5/5

```
CMD ["/dist/server"]
#    ^^^^^^^^^^^^^^^^^
#    Command with arguments
```

Summary

- Container should run `/dist/server` when started
- Can be overridden by the `docker run` command
- Or with the `ENTRYPOINT` instruction

Demonstration

Exercise 1

- Create a simple webserver with Gin Gonic
- Implement endpoint **GET** `/`
  - Respond with HTTP status 200
  - Respond with JSON body: `{"status": "OK"}`
- Create a Dockerfile to build and run the image

Review

What is the image size?

We can do better

- Our image is quite large
- Contains the Go compiler and all dependencies
- We only need the compiled binary

Improvement 1 / multi-stage builds

- Docker images can be built in multiple stages
- Each stage can use a different base image
- Only the final stage is used to create the final image
- Intermediate stages are discarded

Multi-stage build

Example

```
FROM golang:1.20-alpine AS builder

.. # build steps

FROM gcr.io/distroless/static-debian11
#    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#    Tiny base image for Go

COPY --from=builder /dist/server /server
#           ^^^^^^^ ^^^^^^^^^^^^ ^^^^^^^
#           Stage   Source       Target

CMD ["/server"]
```

Improvement 2 / cache go-modules

- Go modules are downloaded on every build
- This can be slow and is not cached

Example

```
FROM golang:1.20-alpine AS builder

COPY go.mod go.sum ./

RUN go mod download

.. # build steps
```

Improvement 3 / run without root

- By default, containers and images are run as root
- This is not recommended for security reasons
- Even though attack surface is limited
- We can create a non-root user and run as that

Run without root

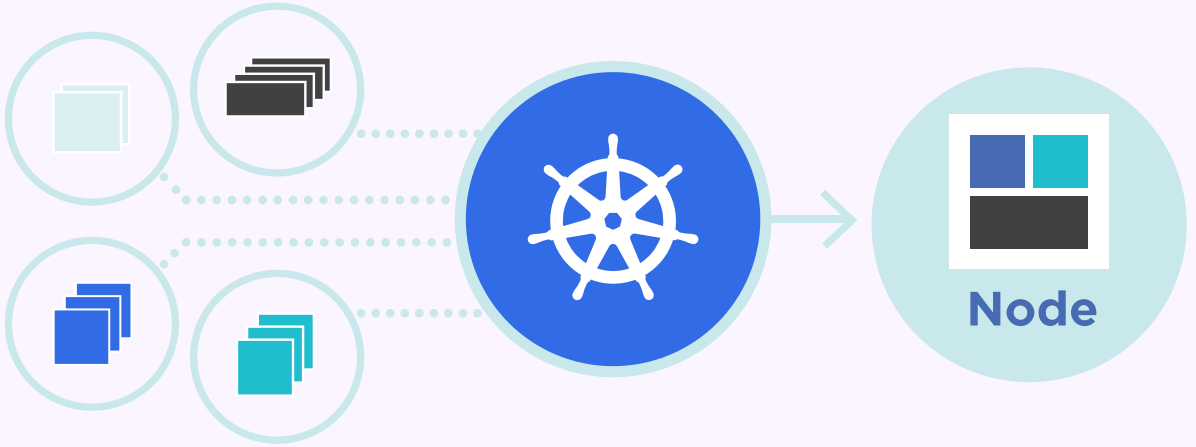
Example

```
FROM gcr.io/distroless/static-debian11

.. # packaging steps

RUN groupadd -r appuser && useradd -r -g appuser appuser
#   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#   Create a new group and user

USER appuser
#    ^^^^^^^
#    Run as user
```

Kubernetes

Summary

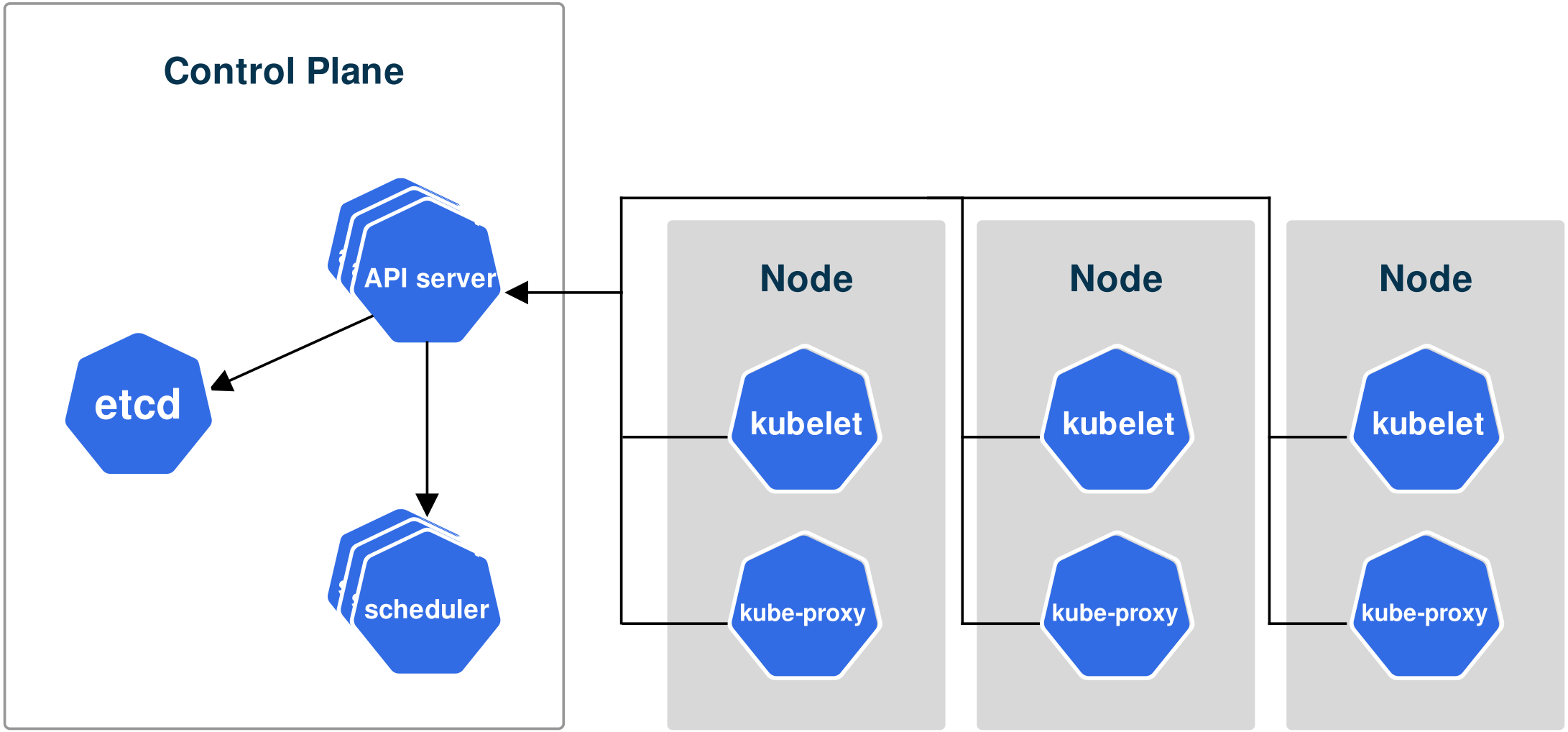
- Container orchestration system
- Originally from Google
- Donated to the CNCF



Features

- **Supports multiple container runtimes**
  containerd, CRI-O
- **And many infrastructures**
  on-promise, hybrid, AWS, GCP, Azure, etc.
- **Written in Go**
  So it has to be awesome

Kubernetes Cluster



Components - 1/5

API server

- The main entry point for all API requests
- Handles authentication and authorization
- Validates and configures data for the cluster

Components - 2/5

Scheduler

- Watches for newly created unassigned pods
- Selects a node for the pod to run on
- Based on resource requirements and other constraints

Components - 3/5

Etcd

- Distributed key-value store, kubernetes its database
- Stores cluster state and configuration (e.g. pods, services, secrets)
- Build on Raft, so fault-tolerant (`n2+1`) and highly available

Components - 4/5

Kubelet

- Primary agent that runs on each node
- Runs containers in pods and manages their lifecycle
    - Using the container runtime (e.g. containerd, CRI-O, etc.)

Components - 5/5

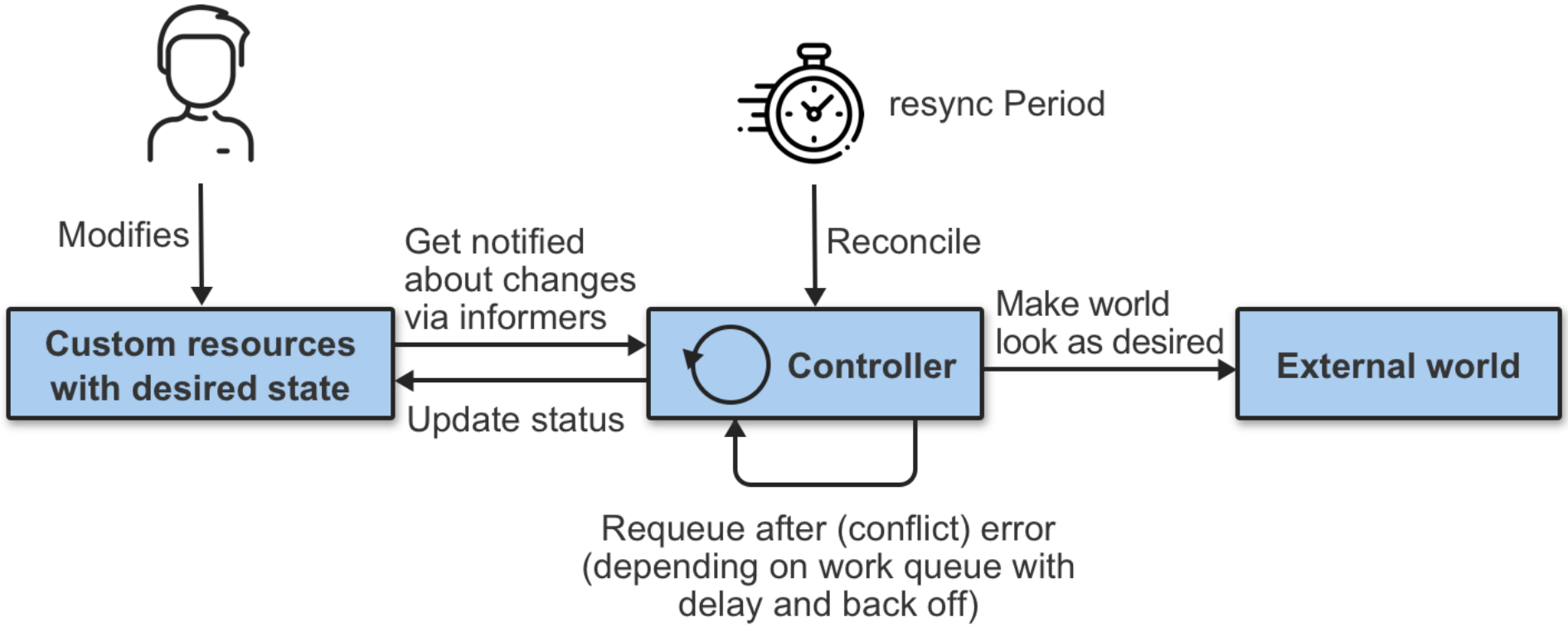
Kube-proxy

- Network proxy that runs on each node
- Provides network connectivity for services
    - Using a network driver (e.g. iptables, ipvs, etc.)

Resources & controllers - 1/2

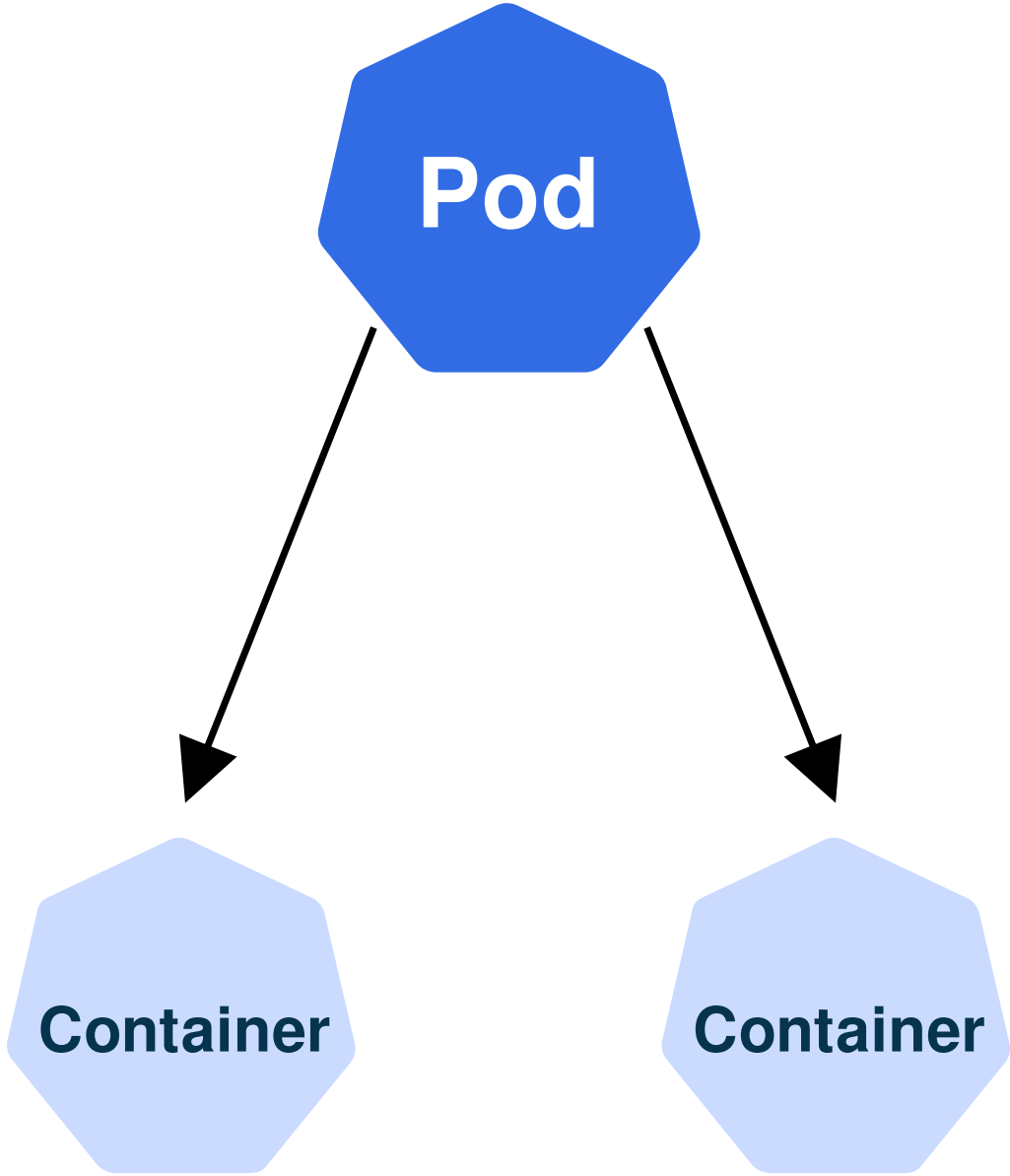
- State is stored in resources, managed declaratively
- Resources are created, updated and deleted using the API
- Controllers watch for changes and reconcile the state

Resources & controllers - 2/2



Pods - 1/2

- Set of one or more containers
- Shared namespaces and filesystem volumes
- Scheduled on the same node
- Typically managed by a controller



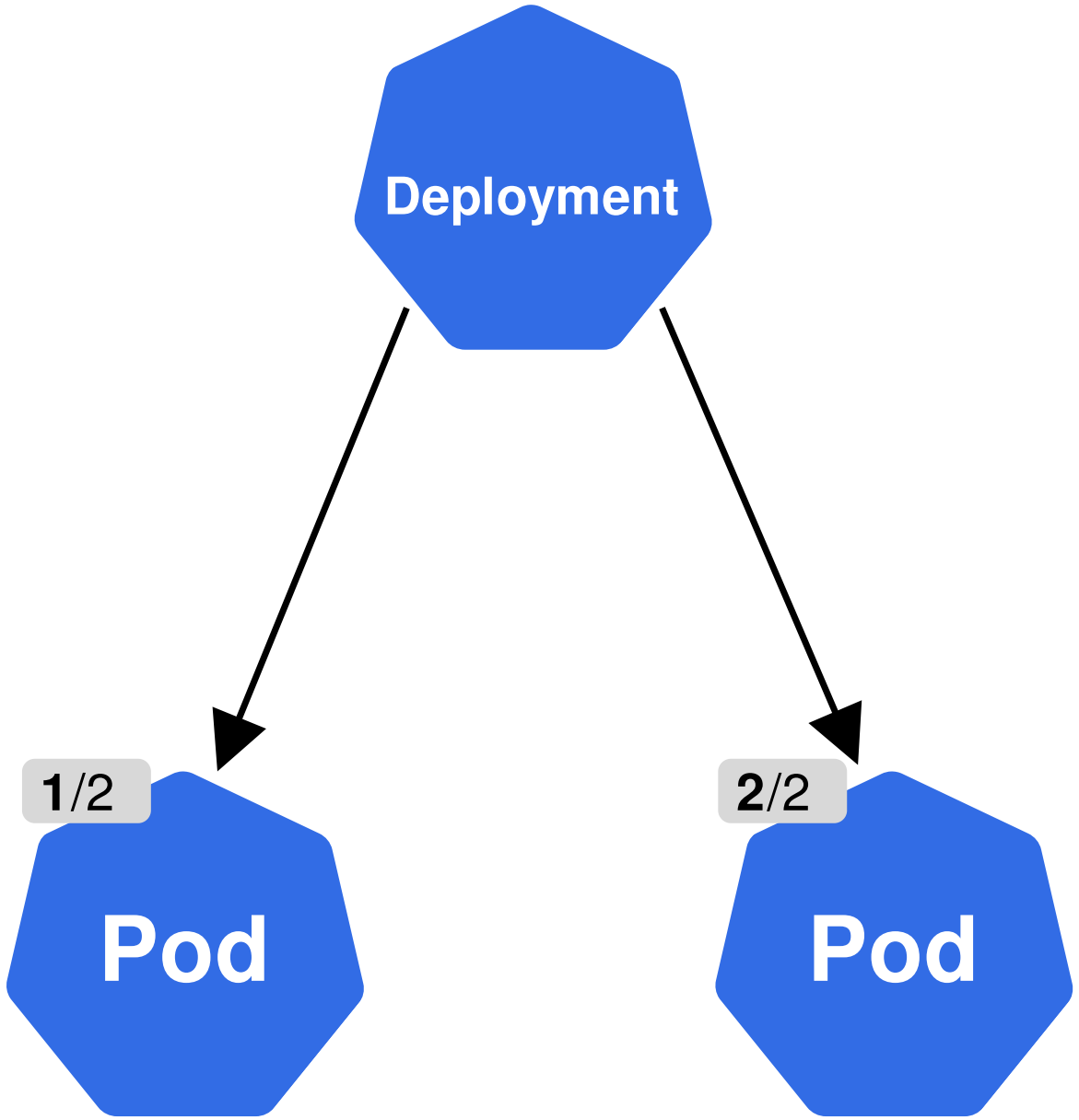
Pods - 2/2

Example

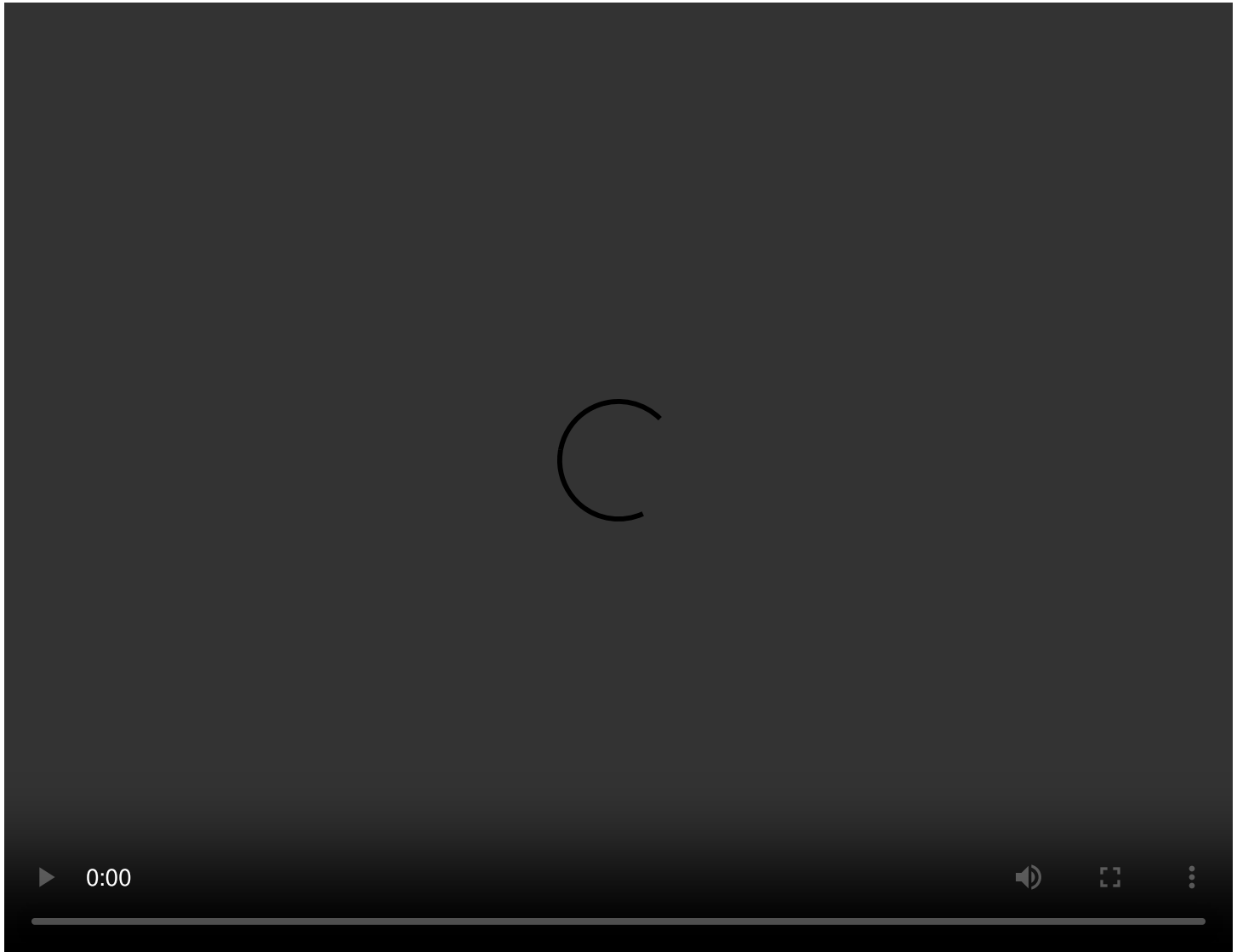
```
apiVersion: v1
kind: Pod
metadata:
  name: hello
  namespace: default
spec:
  containers:
    - name: hello
      image: busybox:1.28
      command: ['sh', '-c', 'echo "Hello world" && sleep 3600']
      restartPolicy: OnFailure
```

Deployments - 1/3

- High-level resource (controller)
- Builds on ReplicaSets (manages desired number of pods)
- Implements rolling updates and rollbacks



Deployments - 2/3



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
  labels:
    app: hello
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      labels:
        app: hello
    spec:
      containers:
      - name: busybox
        image: busybox:1.28
```

Deployments - 3/3

- Selector is used to match pods to the deployment
- Rules can be simple labels or more complex expressions
- The `template` key is used to define the pod spec

Setup Kubernetes

1. **Install k3d**

   Mac OS: `brew install k3d`

   Windows: install `.exe` from https://github.com/k3d-io/k3d/releases

2. **Create cluster**

   Run: `k3d cluster create training`

3. **Verify installation**

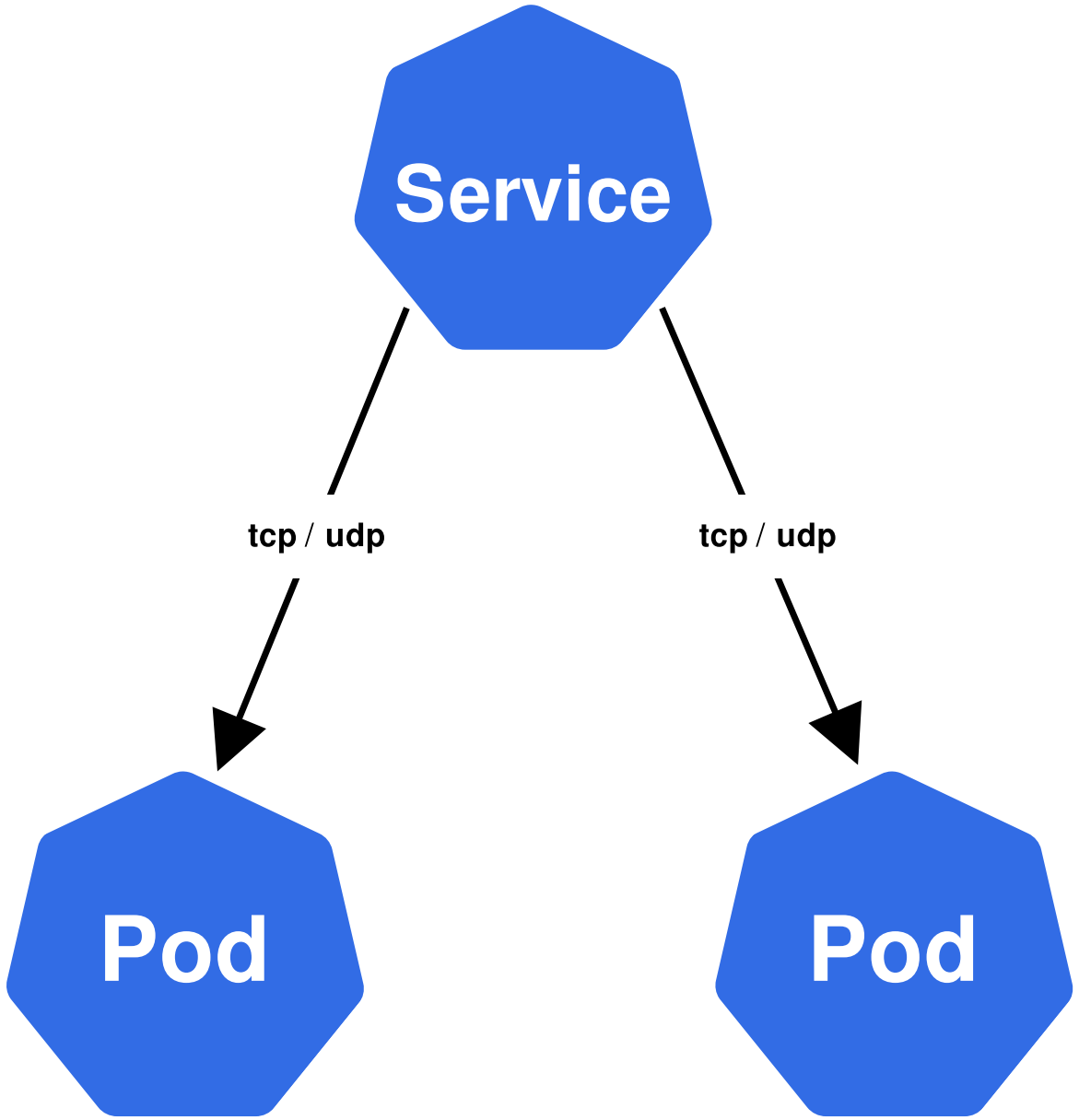
   Run: `kubectl get nodes`

Create deployment

- Create a simple deployment
- Use your existing image
- Deploy the resource to the cluster
- Verify if it works
  - With `kubectl get deployments`
  - And `kubectl get pods -l app=hello`

Services - 1/3

- Pods get IP addresses assigned
- Provides load balancing and service discovery
- Can be exposed to the internet (based on type)



Services - 2/3

Service types:

- **ClusterIP** (default)
  Service only reachable from within the cluster
- **NodePort**
  Service exposed on a static port on each node
- **LoadBalancer**
  Service exposed using a cloud provider load balancer
- **ExternalName**
  Maps service to a DNS name (CNAME record)

Services - 3/3

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-world
spec:
  selector:
    app: hello
#   ^^^^^^^^^^^^^
#   Same selector as deployment uses
  ports:
    - port: 8000
#     ^^^^^^^^^^
#     Port exposed by the service
      targetPort: 8000
#     ^^^^^^^^^^^^^^^^
#     Port exposed by the container
```

Try it yourself

- Create a service for your deployment
- Either create a new file (e.g. `service.yml`) or add it to the same file
  - Multiple resources can be defined, split by: `---`
- Deploy the resource to the cluster
- Verify if it works
  - With `kubectl get services`

Configuration

- Use `ConfigMap` or `Secret` resources
- ConfigMaps for non-sensitive data
- Secrets for sensitive configuration data

Example

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: http-config
data:
  key: value
```

Questions?

CRDs - 1/2

- Not everything can be done with the built-in resources
- What if we want a PostgreSQL database for example?
- This is where CRDs (Custom Resource Definitions) come in
- They allow us to define our own resources
    - And controllers to manage them

CRDs - 2/2

- Examples of CRDs are:
  - **Cert Manager**; for managing TLS certificates
  - **Prometheus Operator**; for managing Prometheus instances
  - **PostgreSQL Operator**; for managing PostgreSQL clusters

Skaffold

- Making changes, building and deploying can be tedious
- Skaffold automates this process
- For a single or multiple microservices

Setup skaffold

Mac OS

```
brew install skaffold
```

Windows

Install `.exe` from https://github.com/GoogleContainerTools/skaffold/releases/

Demonstration

Setup skaffold

- Initialize skaffold (`skaffold init`)
- Verify if it works (`skaffold dev`)

Helm

- Package manager for Kubernetes
- Manages install/upgrade lifecycle
- Repositories
- Charts

Deploy a database

- Install the **cloudnative-pg** operator
- Deploy your own PostgreSQL cluster using the operator
- Verify if it works with `kubectl get clusters`

Go to

training.brainhive.nl

Exercise 2

Summary

- We are going to build the Internet Book DataBase (IBDb)
- A simple REST API for storing book reviews
- We will use the PostgreSQL database we just deployed
- And the project we created in the previous assignment

OpenAPI spec - 1/2

Let's start by checking out the OpenAPI specification.

Go to:

https://tinyurl.com/book-review-api

OpenAPI spec - 2/2

- Quick summary
- Any questions?

Go to

training.brainhive.nl

Review

Testing

How can we test this?

- Any suggestions?

Mocks - 1/4

- Instead of directly using GORM we can create an interface
- Something like the `BookStore` interface for example

Example

```
type BookStore interface {
  GetBook(id int) (*Book, error)
  GetReviews(bookId int) ([]Review, error)
  CreateReview(rating int, comment string) error
}
```

Mocks - 2/4

- This interface can be mocked (basically "fake" the implementation)
- We can then use the mock in our tests
- This way we can test our code without a database

Example

```
type BookStore interface {
  GetBook(id int) (*Book, error)
  GetReviews(bookId int) ([]Review, error)
  CreateReview(rating int, comment string) error
}
```

Mocks - 3/4

- Mocks can be generated by using `mockgen` based on our interface
- We can automate this by using `go generate`

Example

```
//go:generate mockgen . UserStore -destination=mocks_test.go
```

Demonstration