# SCALE 13x
# Container Management at Google Scale

Tim Hockin <thockin@google.com>
Senior Staff Software Engineer
@thockin

# SCALE 13x

# Container Management at Google Scale

Tim Hockin <thockin@google.com>
Senior Staff Software Engineer
@thockin

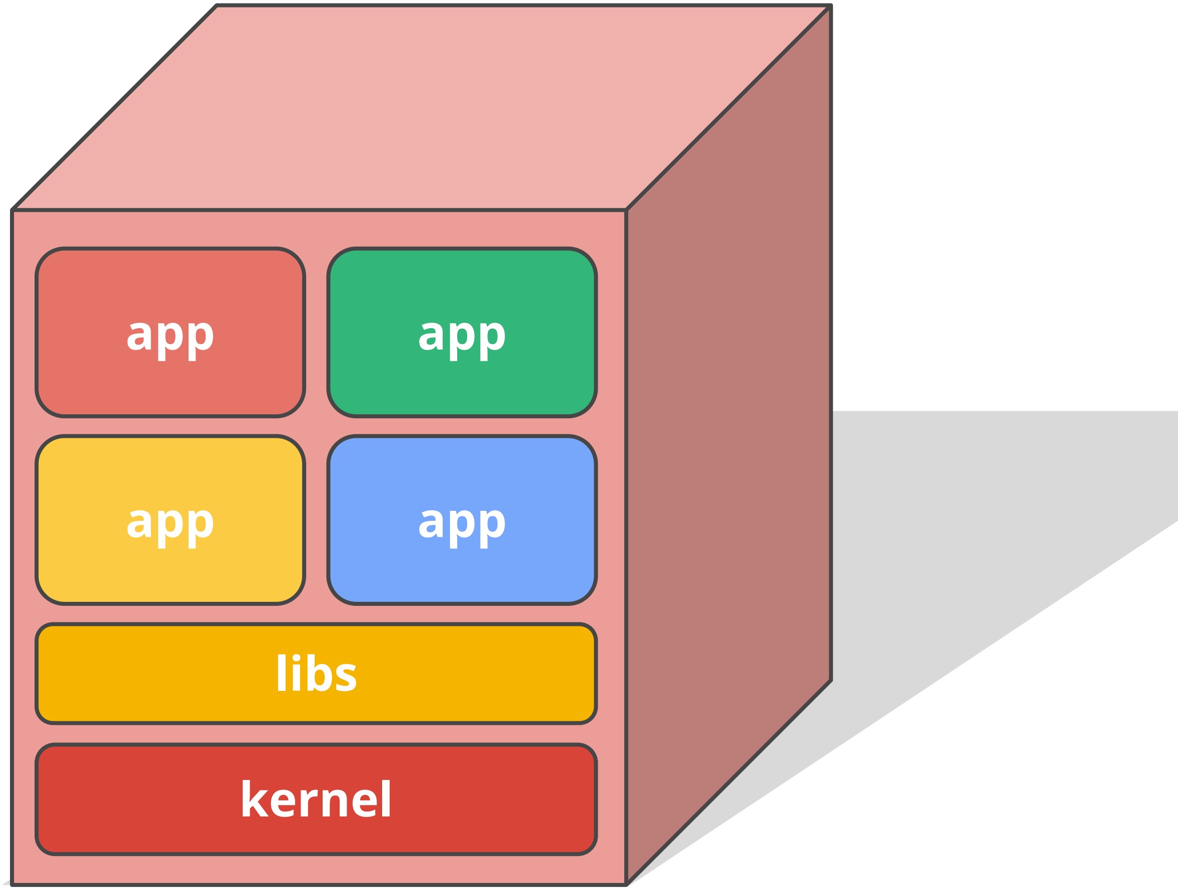
# Old Way: Shared machines

No isolation

No namespacing

Common libs

Highly coupled apps and OS



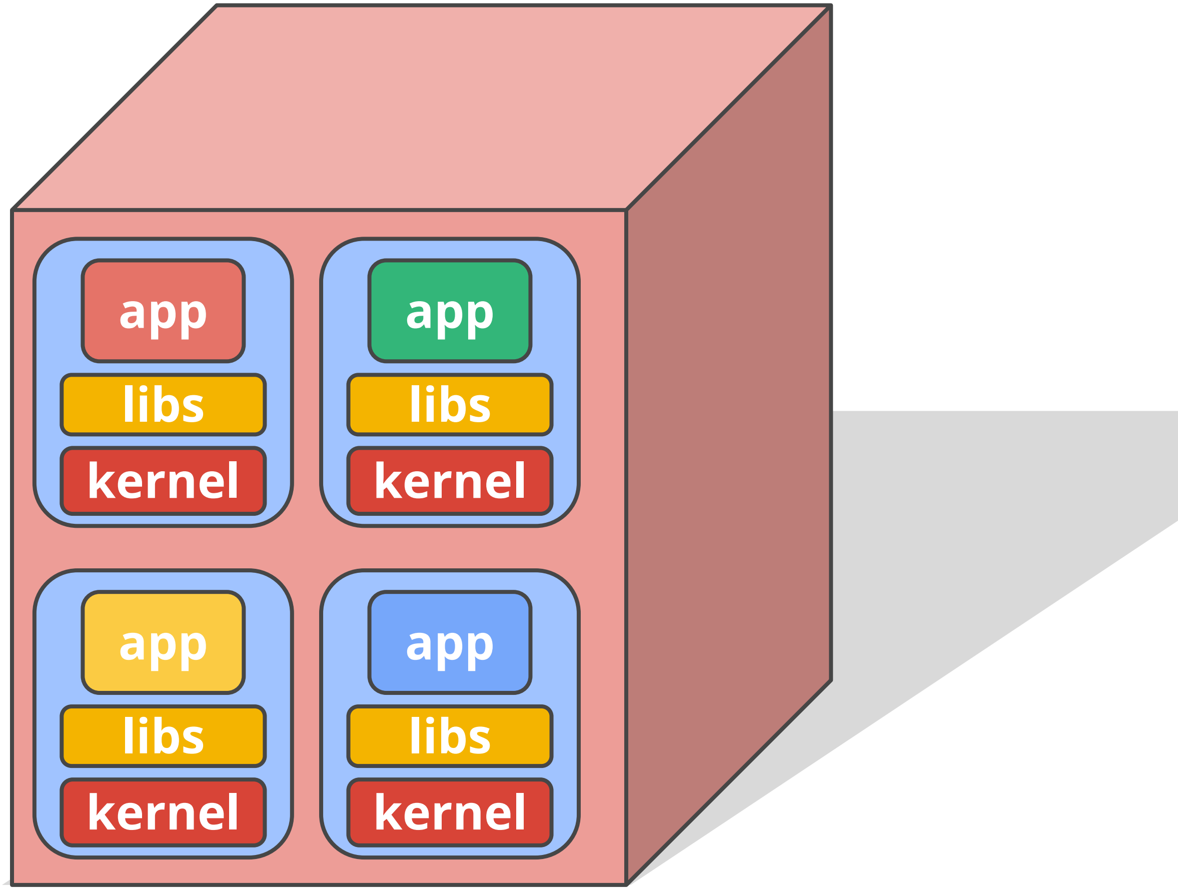
# Old Way: Virtual machines

Some isolation

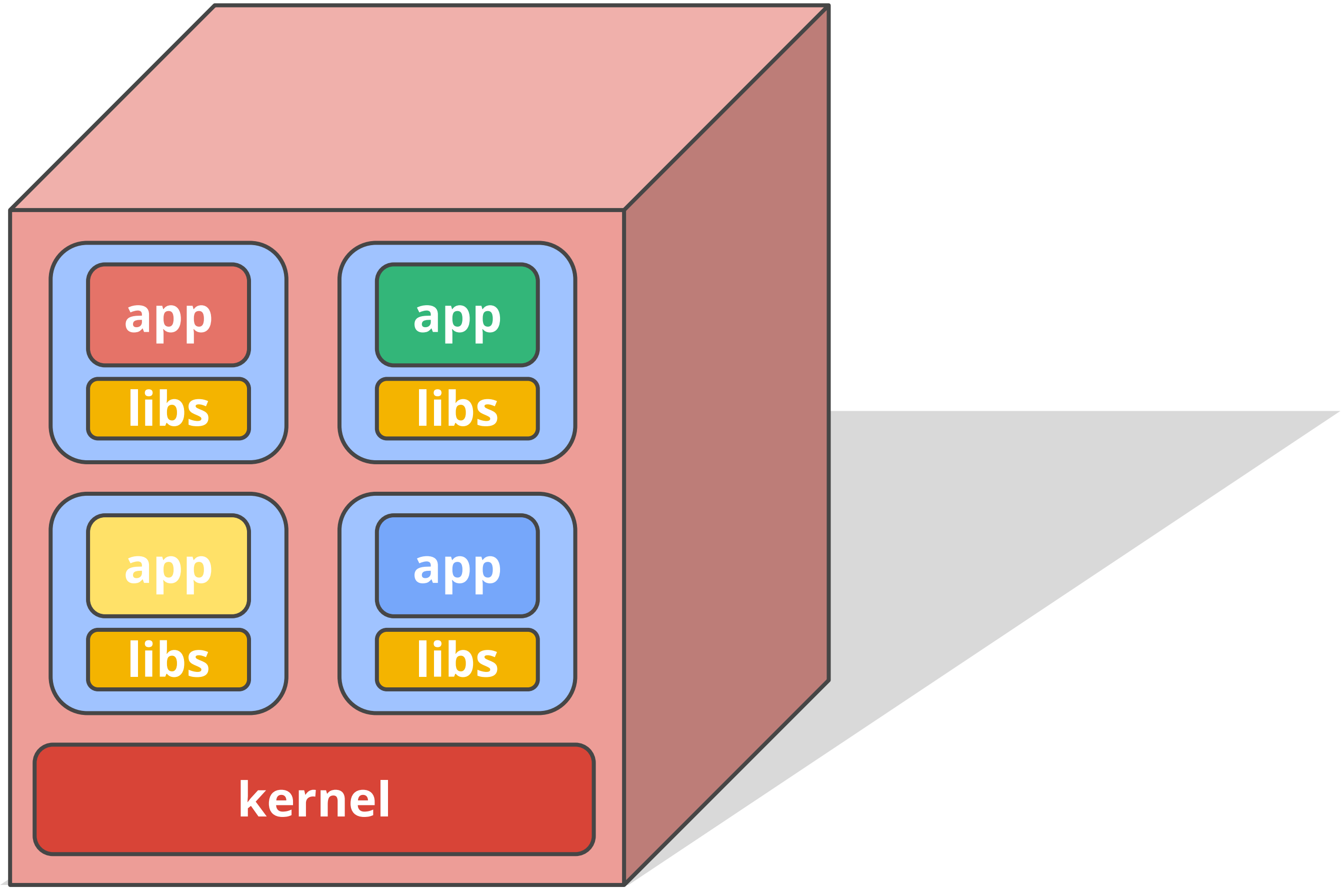
Expensive and inefficient

Still highly coupled to the guest OS

Hard to manage



# New Way: Containers



# But what ARE they?

Lightweight VMs

- no guest OS, lower overhead than VMs, but no virtualization hardware

Better packages

- no DLL hell

Hermetically sealed static binaries

- no external dependencies

Provide Isolation (from each other and from the host)

- Resources (CPU, RAM, Disk, etc.)
- Users
- Filesystem
- Network

# How?

Implemented by a number of (unrelated) Linux APIs:

- **cgroups:** Restrict resources a process can consume
  - CPU, memory, disk IO, ...
- **namespaces:** Change a process's view of the system
  - Network interfaces, PIDs, users, mounts, ...
- **capabilities:** Limits what a user can do
  - mount, kill, chown, ...
- **chroots:** Determines what parts of the filesystem a user can see

Google has been developing and using **containers** to manage our applications for **over 10 years.**

Images by Connie Zhou

**Everything** at Google runs in containers:

- Gmail, Web Search, Maps, ...
- MapReduce, batch, ...
- GFS, Colossus, ...
- Even GCE itself: VMs in containers

Shipping Containers At Clyde, by Steve Gibson

**Everything** at Google runs in containers:

- Gmail, Web Search, Maps, ...
- MapReduce, batch, ...
- GFS, Colossus, ...
- Even GCE itself: VMs in containers

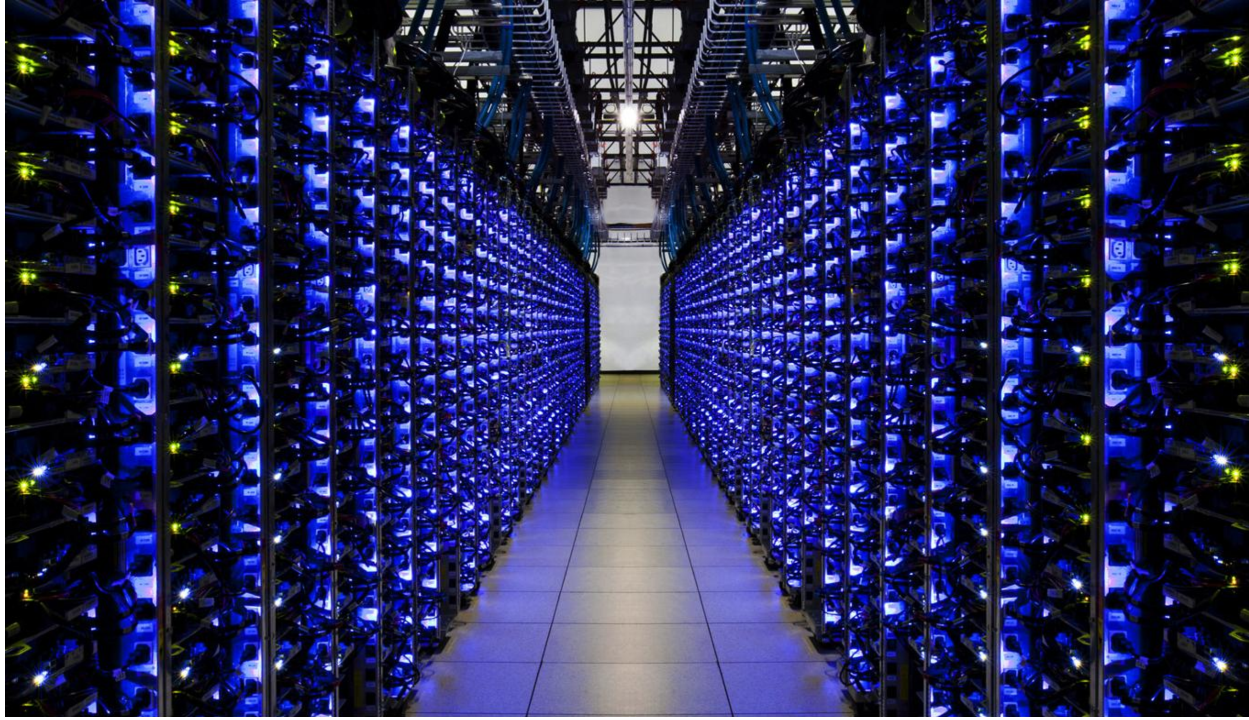
We launch over **2 billion** containers **per week**.

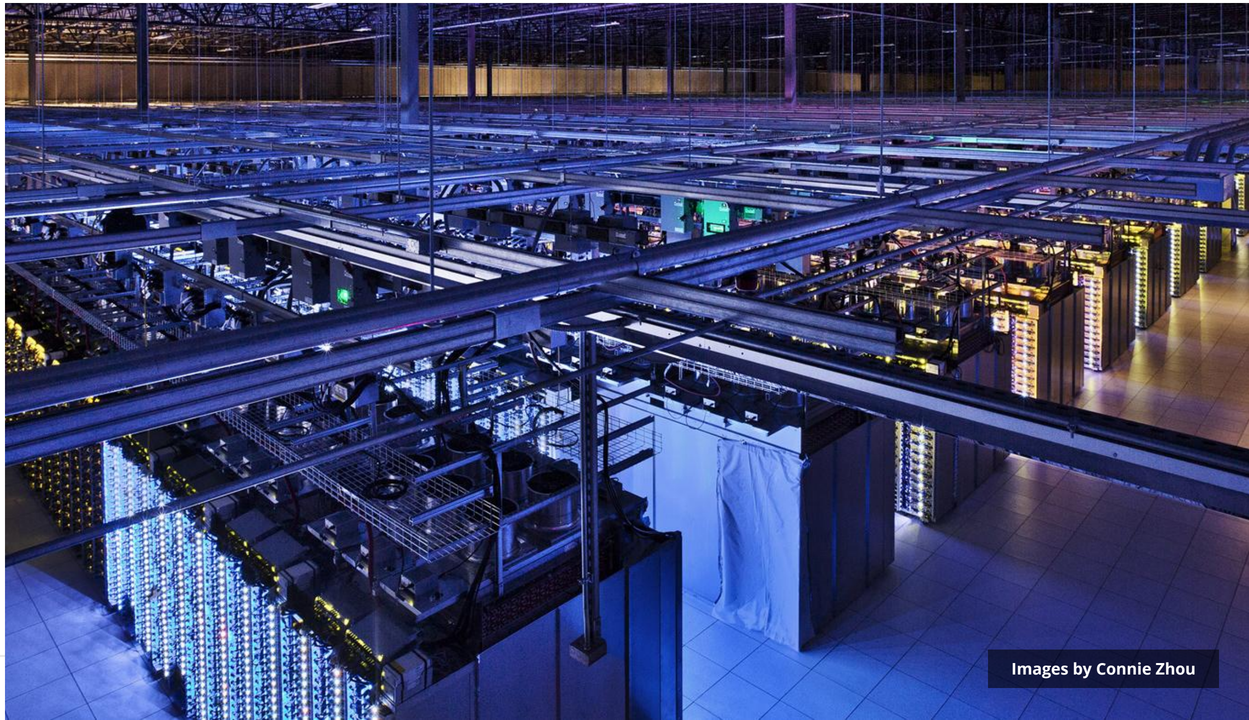
Shipping Containers At Clyde, by Steve Gibson

# Why containers?

- Performance
- Repeatability
- Isolation
- Quality of service
- Accounting
- Visibility
- Portability

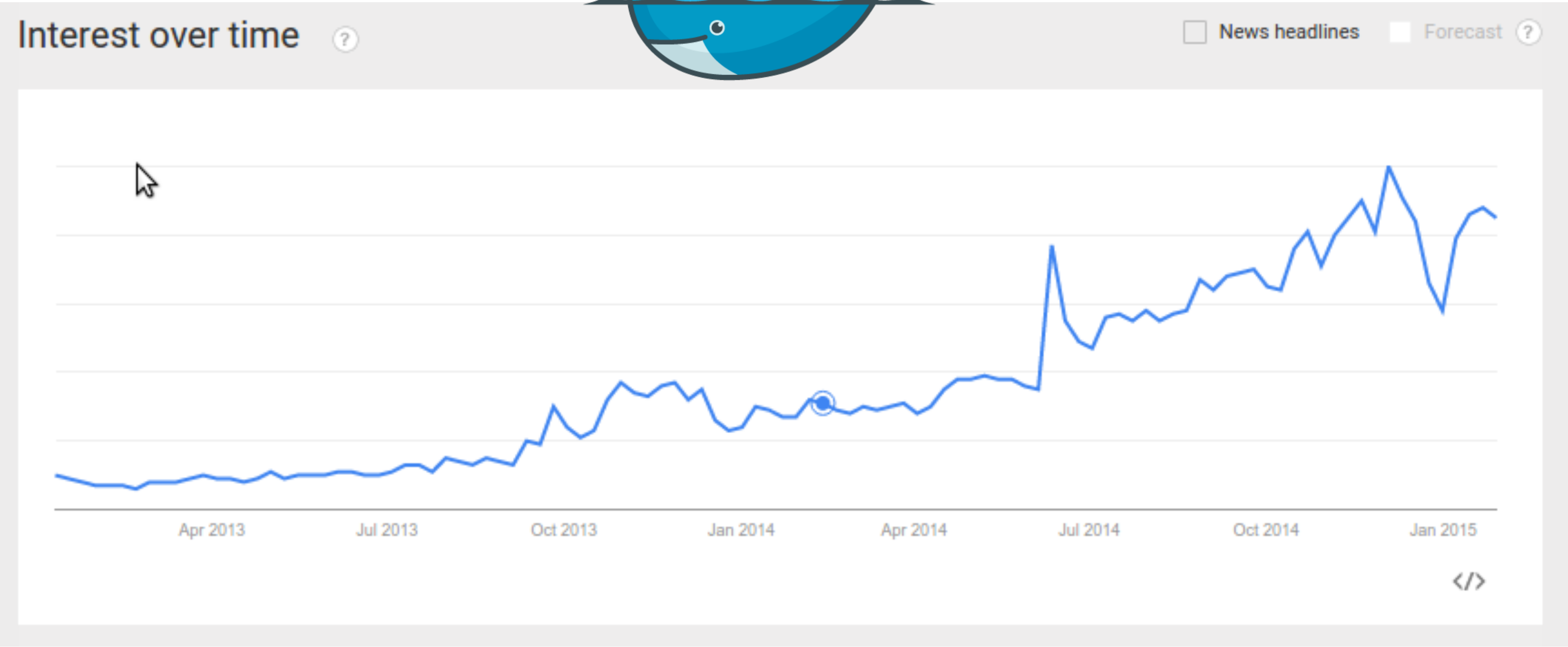
A **fundamentally different** way of managing **applications**





Images by Connie Zhou

# Docker




Source: Google Trends

# But what IS Docker?

An implementation of the container idea

A package format

An ecosystem

A company

An open-source juggernaut

A phenomenon

Hoorah! The world is starting to adopt containers!

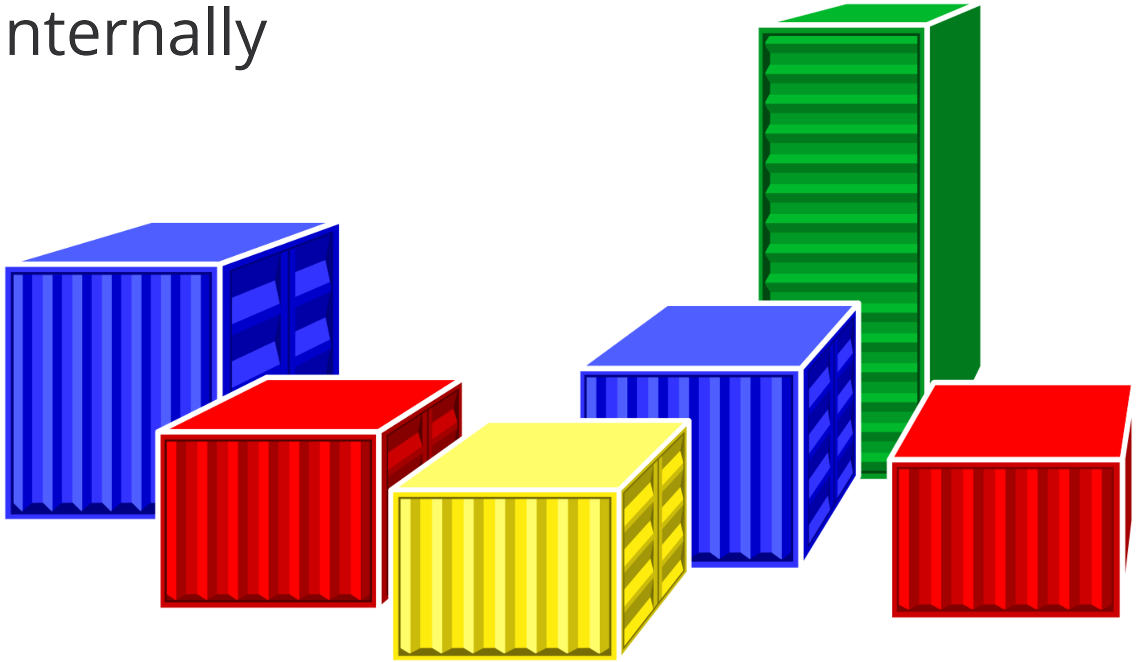
# LMCTFY

Also an implementation of the container idea (from Google)

Also open-source

Literally the same code that Google uses internally

“Let Me Contain That For You”



# LMCTFY

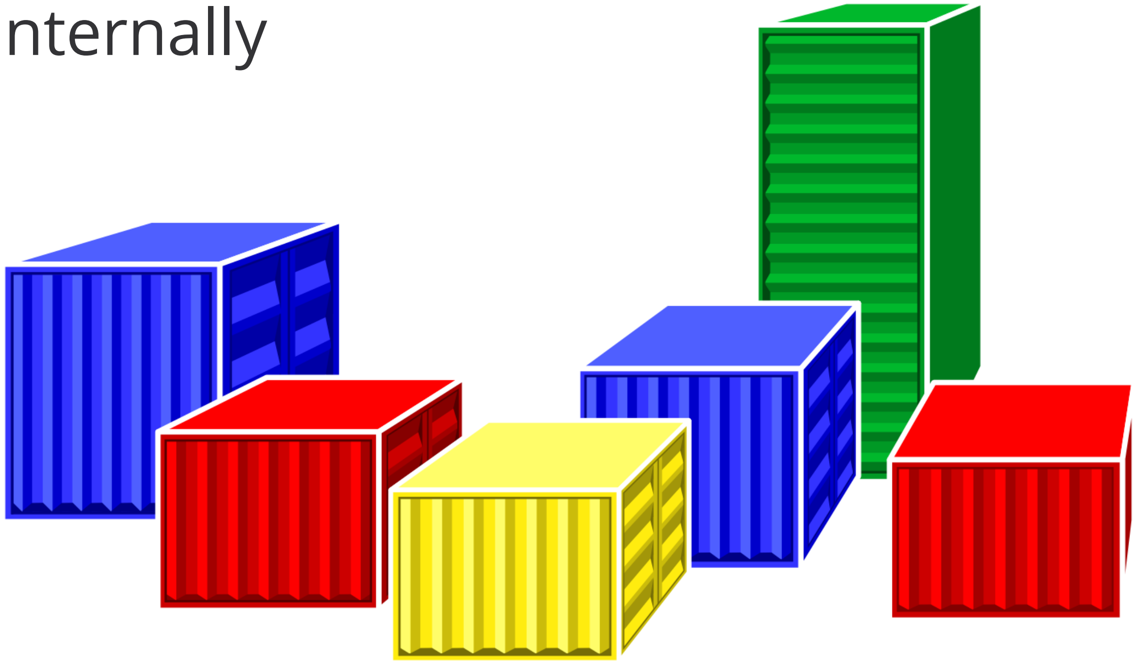
Also an implementation of the container idea (from Google)

Also open-source

Literally the same code that Google uses internally

“Let Me Contain That For You”

**Probably NOT what you want to use!**



# Docker vs. LMCTFY

Docker is primarily about namespacing: control what you can see

- resource and performance isolation were afterthoughts

LMCTFY is primarily about performance isolation: jobs can not hurt each other

- namespacing was an afterthought

Docker focused on making things simple and self-contained

- “sealed” images, a repository of pre-built images, simple tooling

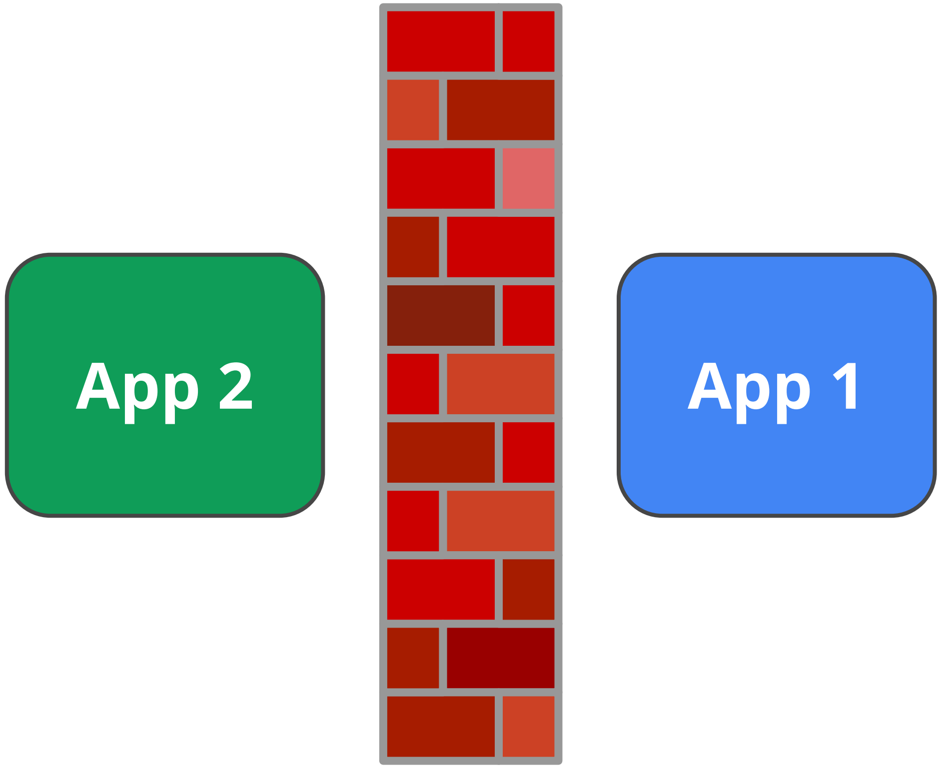
LMCTFY focused on solving the isolation problem very thoroughly

- totally ignored images and tooling

# About isolation

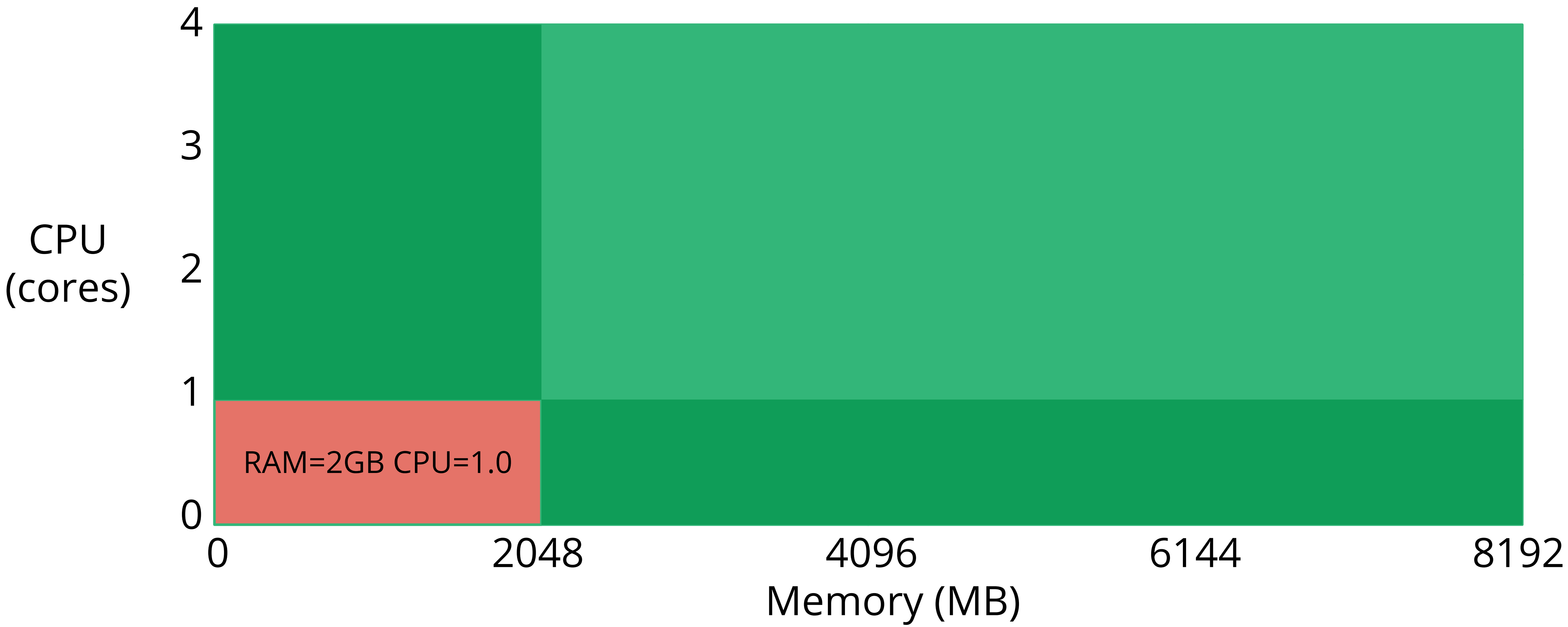
Principles:

- Apps must not be able to affect each other’s perf
  - if so it is an **isolation failure**
- Repeated runs of the same app should see ~equal perf
- Graduated QoS drives resource decisions in real-time
- Correct in all cases, optimal in some
  - reduce unreliable components
- SLOs are the lingua franca

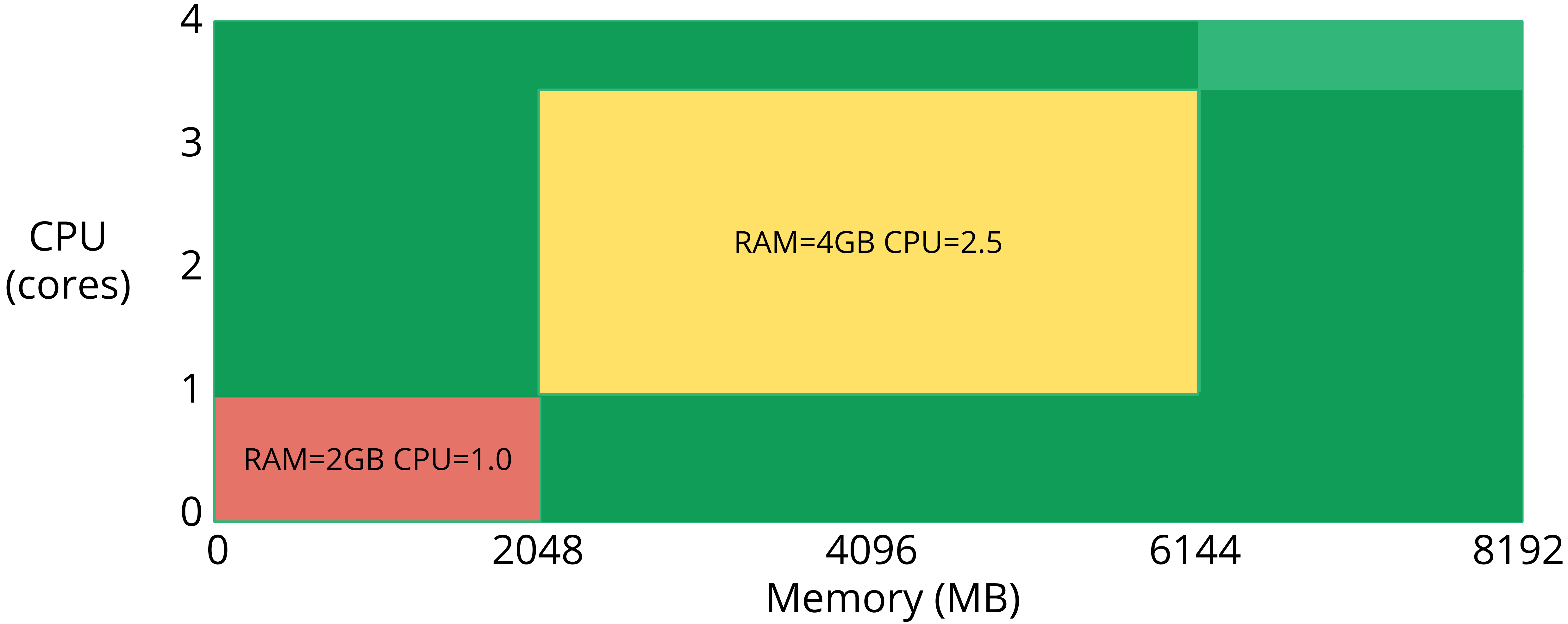


# Strong isolation

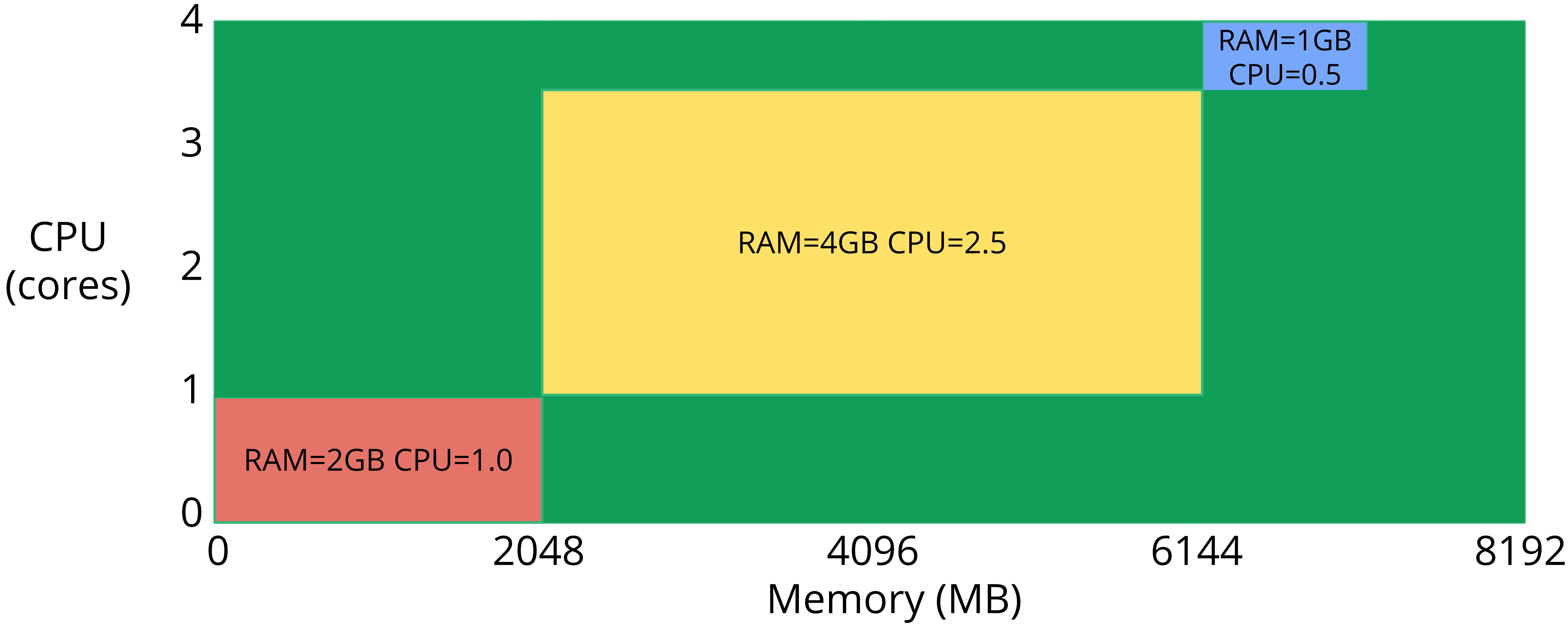
# Strong isolation



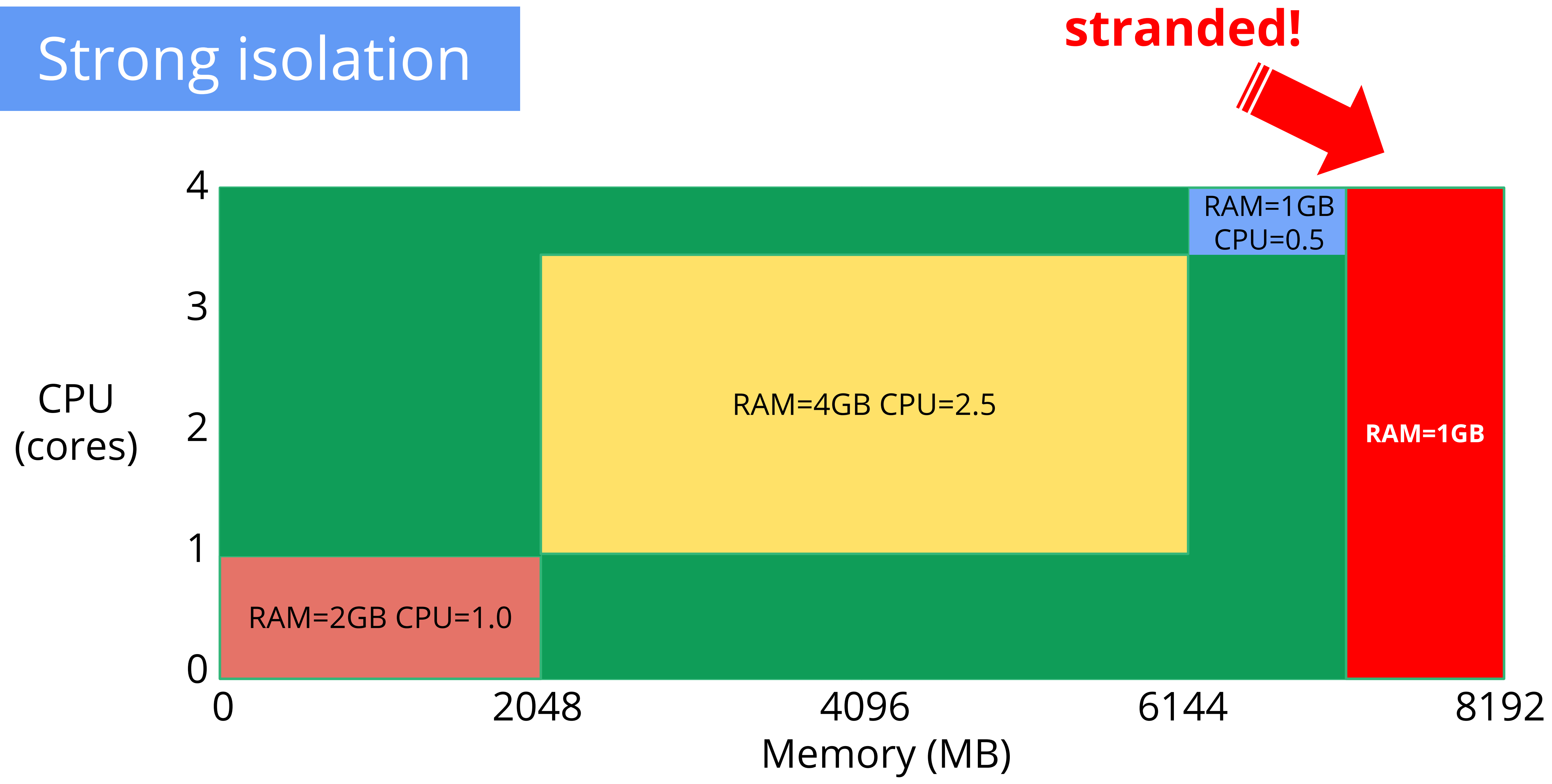
# Strong isolation



# Strong isolation



# Strong isolation



# Strong isolation

Pros:

- Sharing - users don’t worry about interference (aka the noisy neighbor problem)
- Predictable - allows us to offer strong SLAs to apps

Cons:

- Stranding - arbitrary slices mean some resources get lost
- Confusing - how do I know how much I need?
  - analog: what size VM should I use?
  - smart auto-scaling is needed!
- Expensive - you pay for certainty

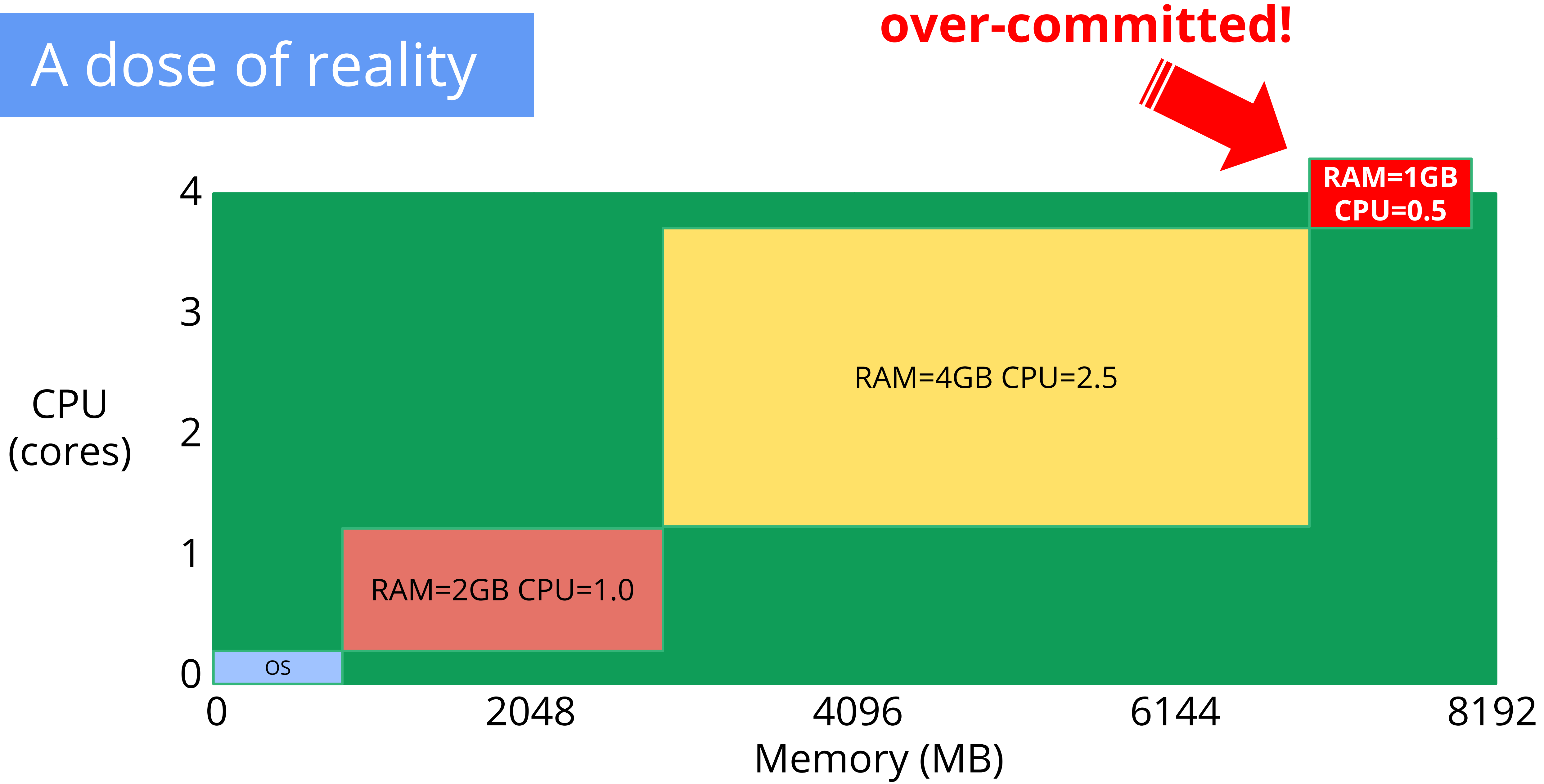
In reality this is a multi-dimensional bin-packing problem: CPU, memory, disk space, IO bandwidth, network bandwidth, ...

# A dose of reality

The kernel itself uses some resources “off the top”

- We can estimate it statistically but we can’t really **limit** it

# A dose of reality



# A dose of reality

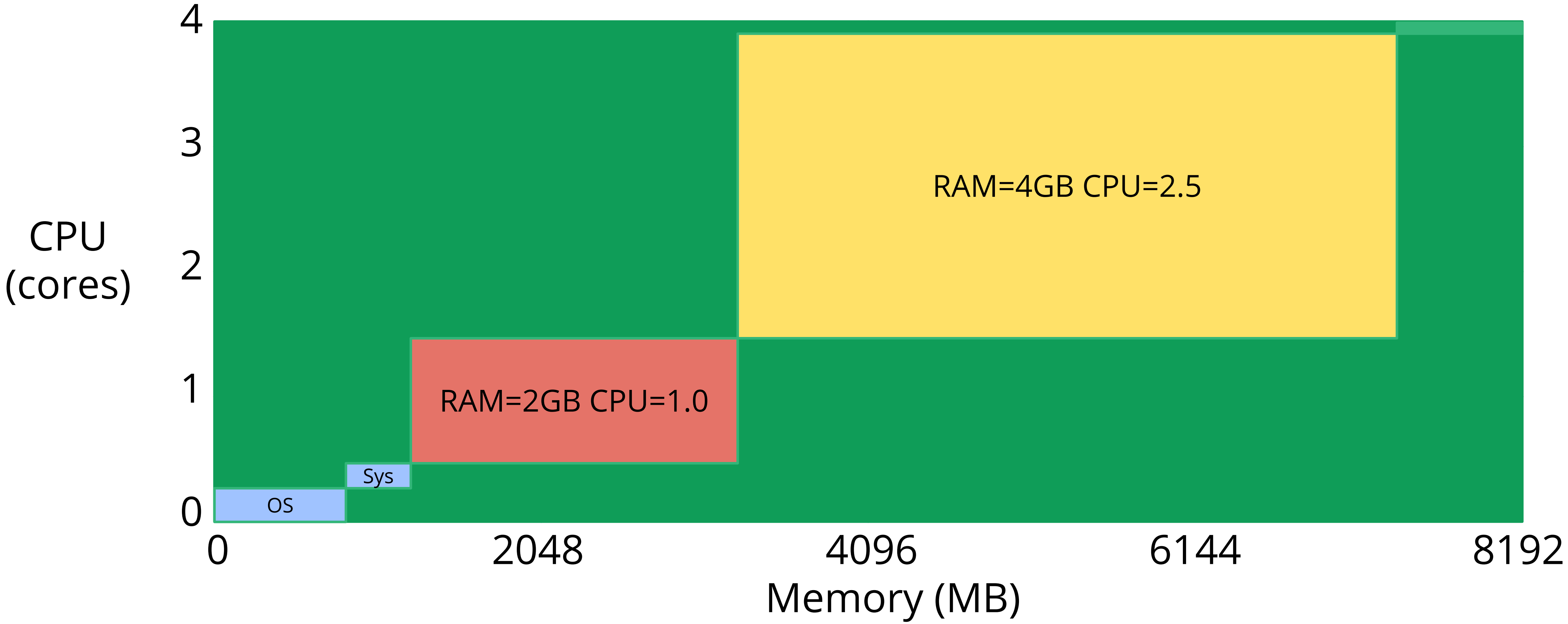
The kernel itself uses some resources “off the top”

- We can estimate it statistically but we can’t really **limit** it

System daemons (e.g. our node agent) use some resources

- We can (and do) limit these, but failure modes are not always great

# A dose of reality



# A dose of reality

The kernel itself uses some resources “off the top”

- We can estimate it statistically but we can’t really **limit** it

System daemons (e.g. our node agent) use some resources

- We can (and do) limit these, but failure modes are not always great

If ANYONE is uncontained, then all SLOs are void.  We pretend that the kernel is contained,  but only because we have no real choice.  Experience shows this holds up most of the time.  Hold this thought for later...

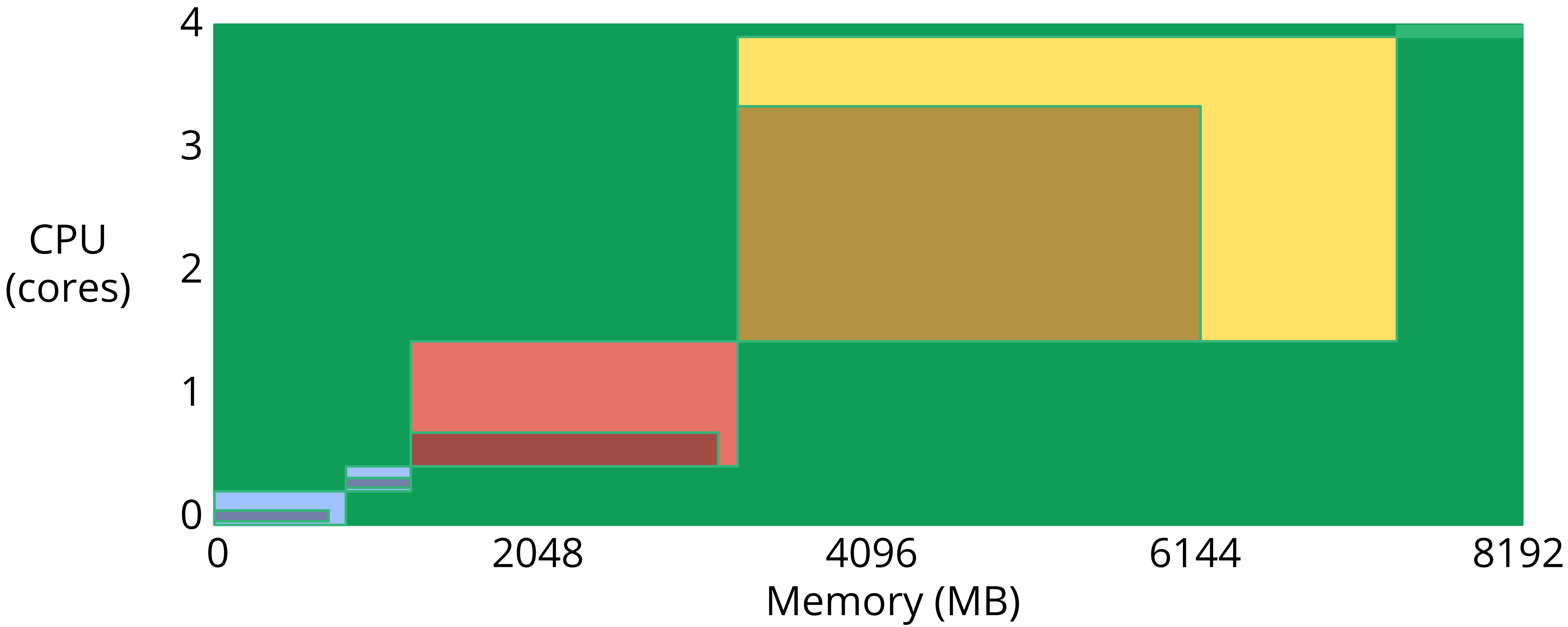
# Results

Overall this works VERY well for latency-sensitive serving jobs

Shortcomings:

- There are still some things that can not be easily contained in real time
  - e.g. cache (see CPI[2])
- Some resource dimensions are **really** hard to schedule
  - e.g. disk IO - so little of it, so bursty, and **SO SLOW**
- Low utilization: nobody uses 100% of what they request
- Not well tuned for compute-heavy work (e.g. batch)
- Users don't really know how much CPU/RAM/etc. to request

# Usage vs bookings



# Making better use of it all

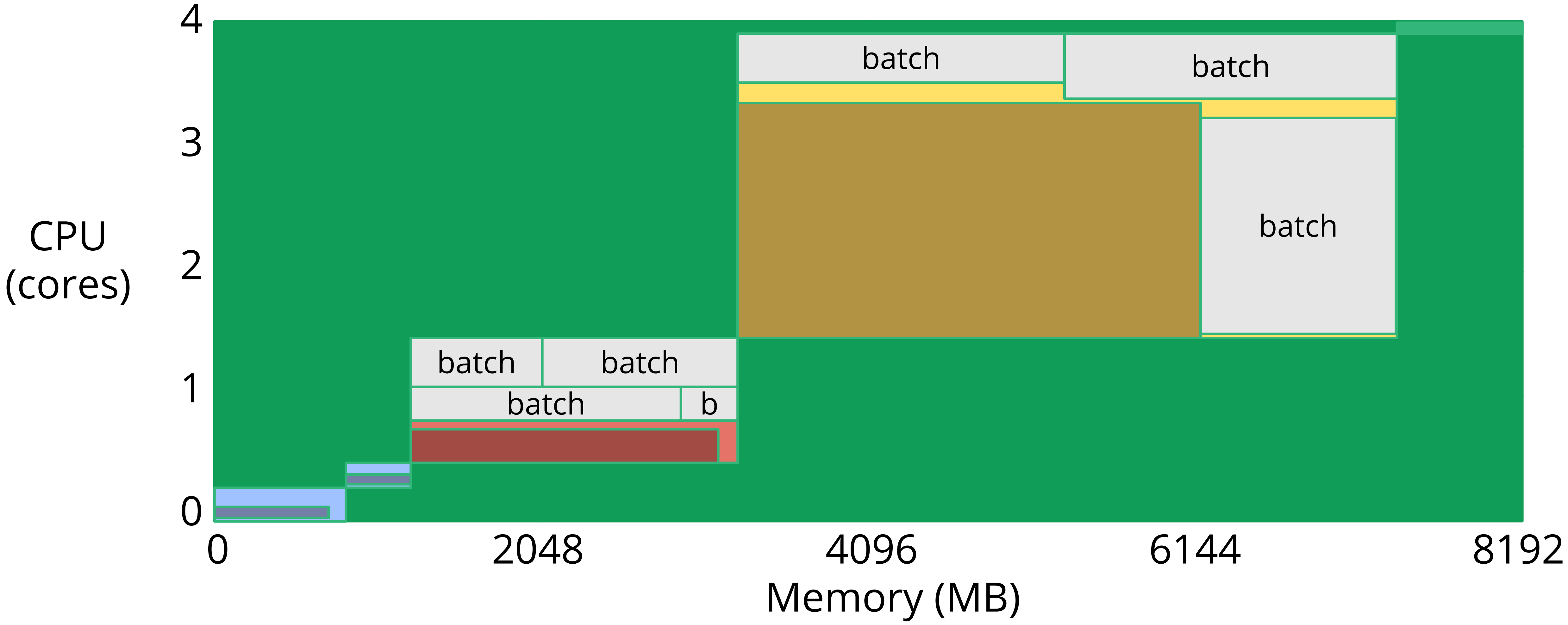
Proposition: Re-sell unused resources with lower SLOs

- Perfect for batch work
- Probabilistically “good enough”

Shortcomings:

- Even more emphasis on isolation failures
  - we can’t let batch hurt “paying” customers
- Requires a lot of smarts in the lowest parts of the stack
  - e.g. deterministic OOM killing by priority
  - we have a number of kernel patches we want to mainline, but we have had a hard time getting upstream kernel on board

# Usage vs bookings



# Back to Docker

Container isolation today:

- ...does not handle most of this
- ...is fundamentally voluntary
- ...is an obvious area for improvement in the coming year(s)

# More than just isolation

Scheduling: Where should my job be run?

Lifecycle: Keep my job running

Discovery: Where is my job now?

Constituency: Who is part of my job?

Scale-up: Making my jobs bigger or smaller

Auth{n,z}: Who can do things to my job?

Monitoring: What's happening with my job?

Health: How is my job feeling?

...

# Enter Kubernetes

Greek for *"Helmsman"*; also the root of the word *"Governor"*

- Container orchestrator
- Runs Docker containers
- Supports multiple cloud and bare-metal environments
- Inspired and informed by Google's experiences and internal systems
- **Open source**, written in **Go**

Manage applications, not machines

# Design principles

**Declarative > imperative**: State your desired results, let the system actuate

**Control loops**: Observe, rectify, repeat

**Simple > Complex:** Try to do as little as possible

**Modularity**: Components, interfaces, & plugins

**Legacy compatible**: Requiring apps to change is a non-starter

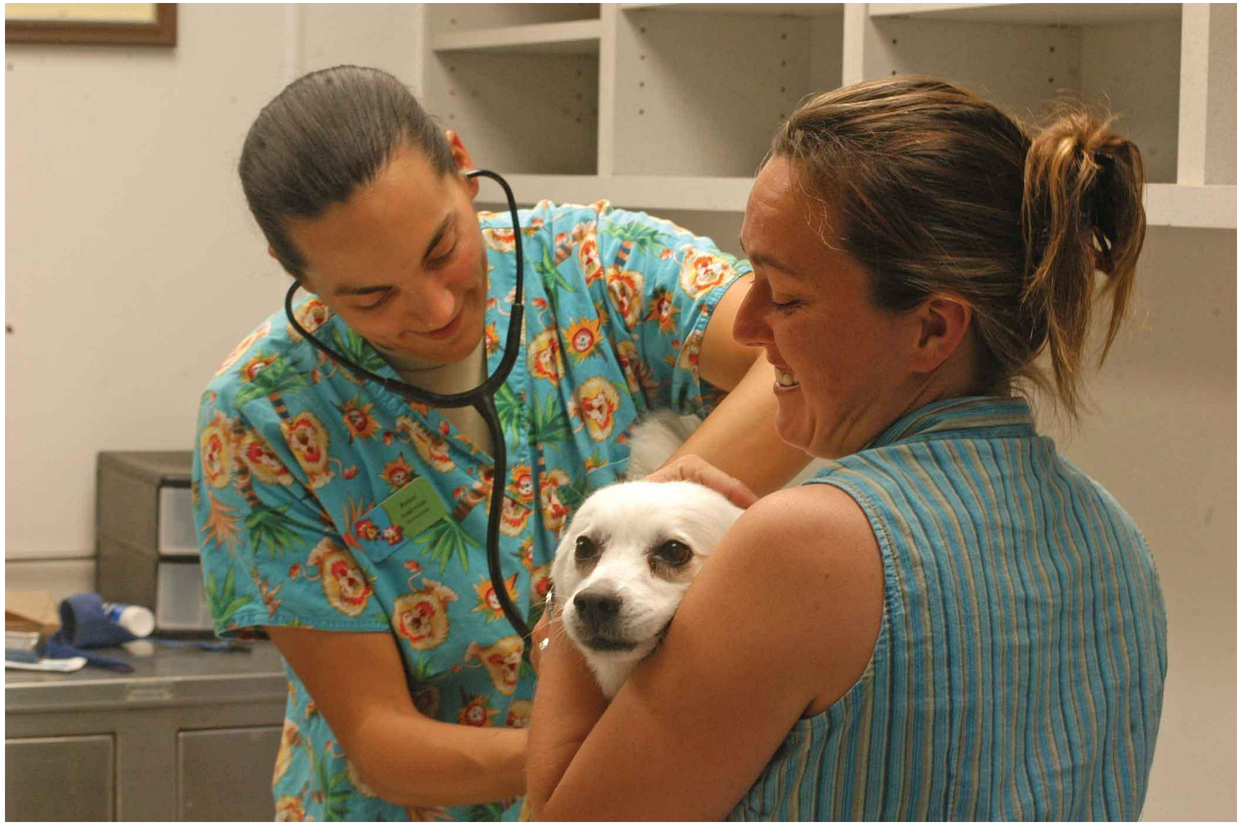
**Network-centric**: IP addresses are cheap

**No grouping**: Labels are the only groups

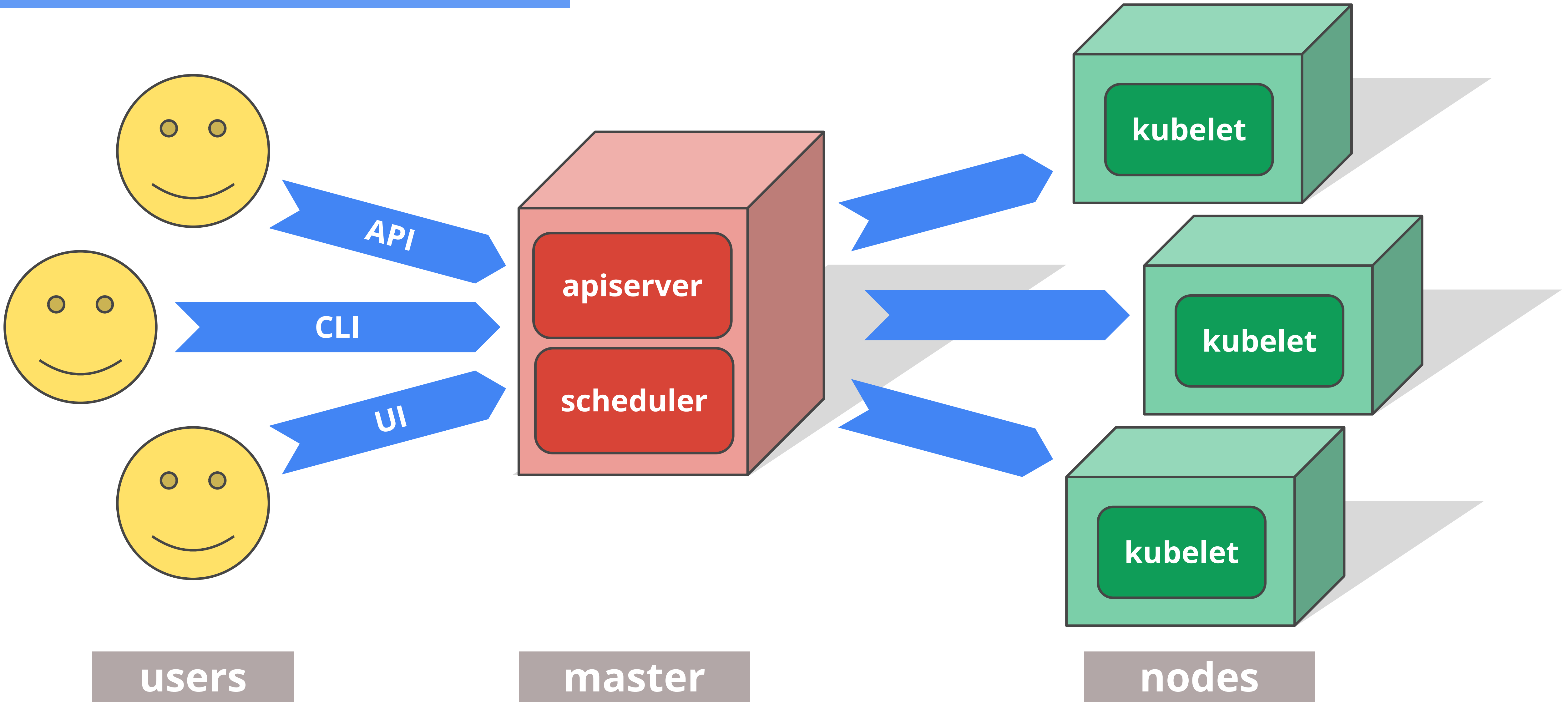
**Cattle > Pets**: Manage your workload in bulk

**Open > Closed**: Open Source, standards, REST, JSON, etc.

# Pets vs. Cattle



# High level design



# Primary concepts

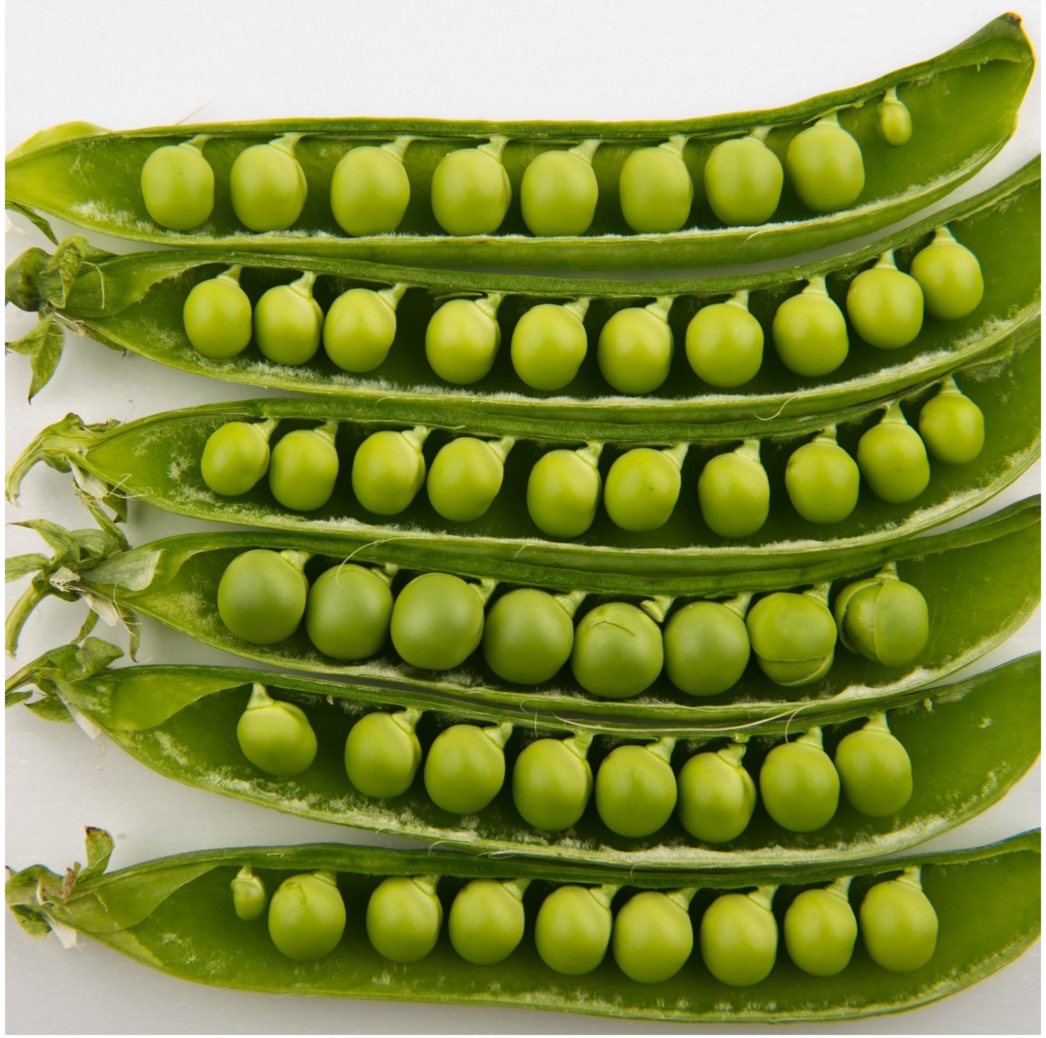
**Container**: A sealed application package (Docker)
**Pod**: A small group of tightly coupled Containers
example: content syncer & web server

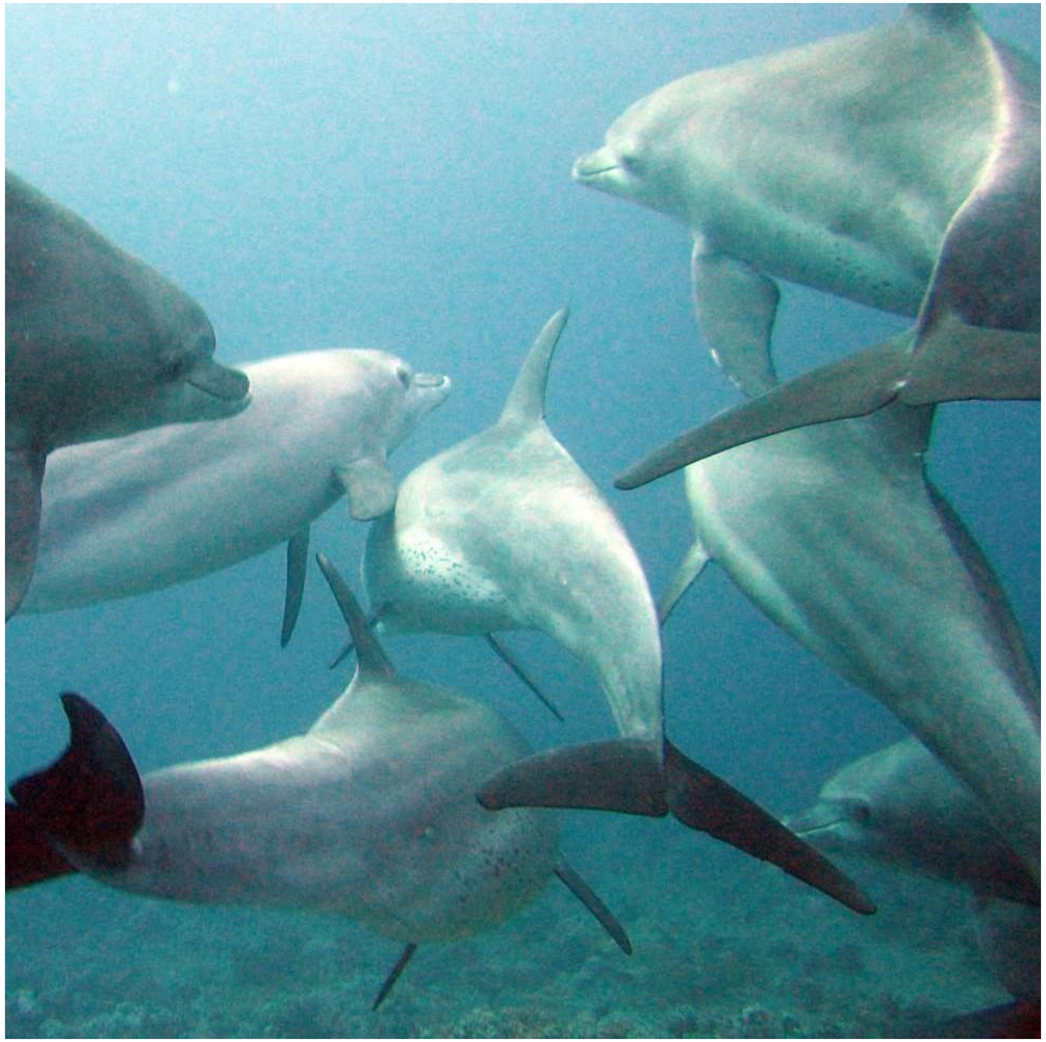
**Controller**: A loop that drives current state towards desired state
example: replication controller

**Service**: A set of running pods that work together
example: load-balanced backends

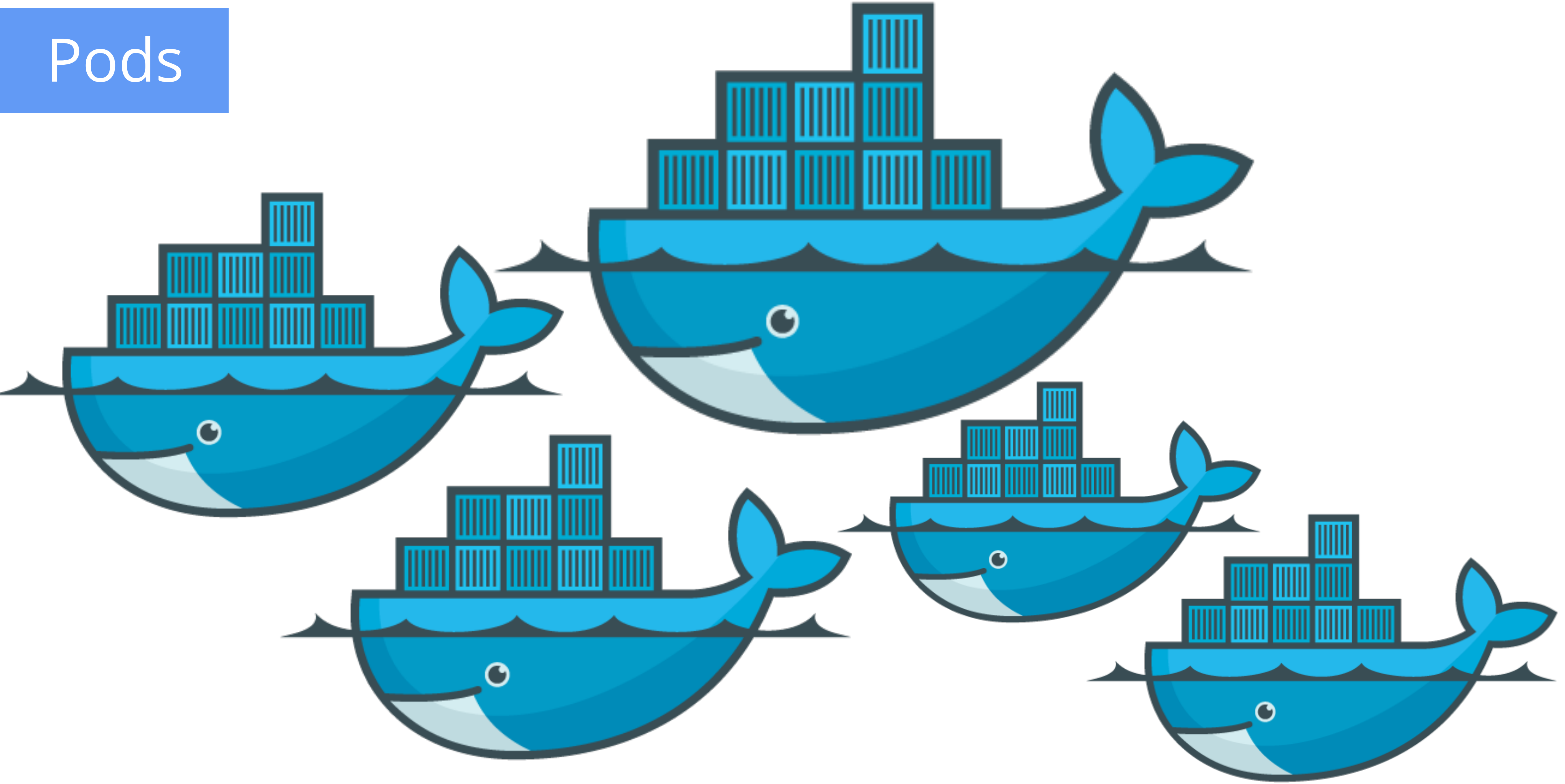
**Labels**: Identifying metadata attached to other objects
example: phase=canary vs. phase=prod
**Selector**: A query against labels, producing a set result
example: all pods where label phase == prod

# Pods





# Pods



# Pods

**Small group** of containers & volumes

**Tightly** coupled

The atom of cluster scheduling & placement

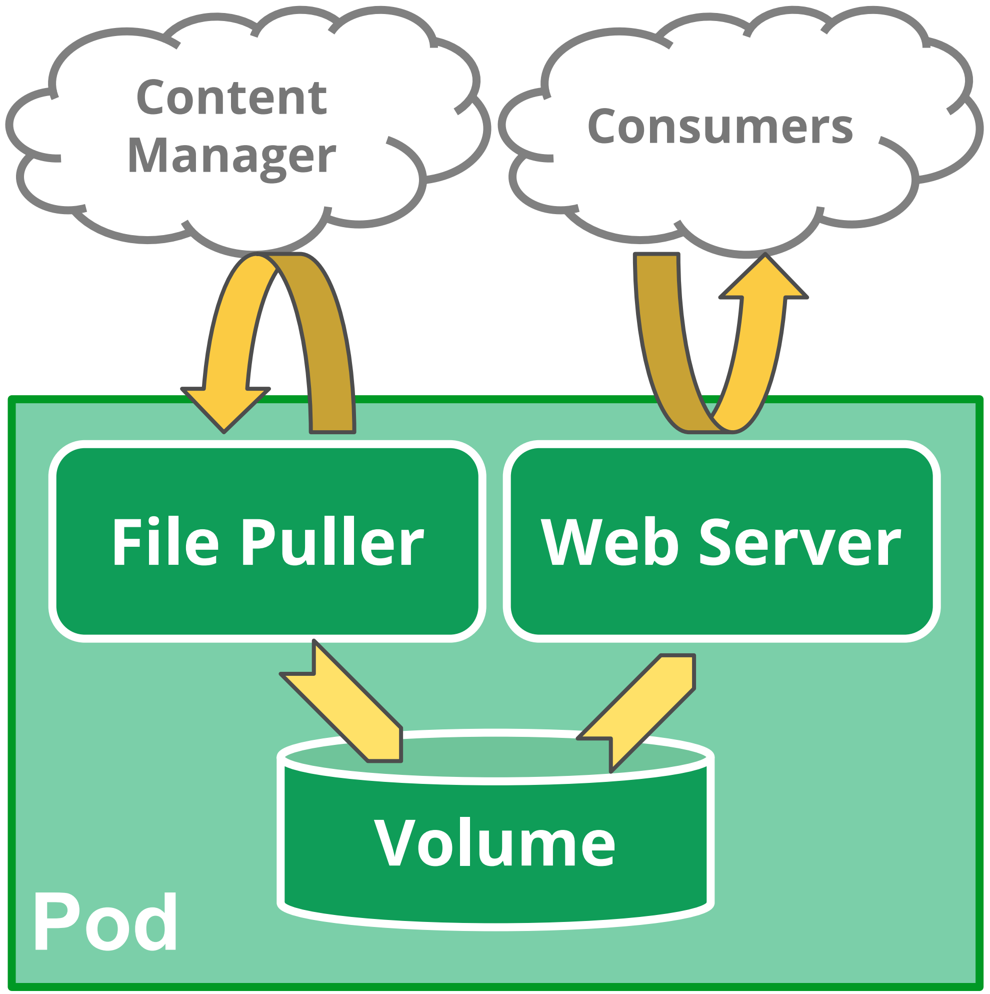
Shared namespace

- **share IP** address & localhost

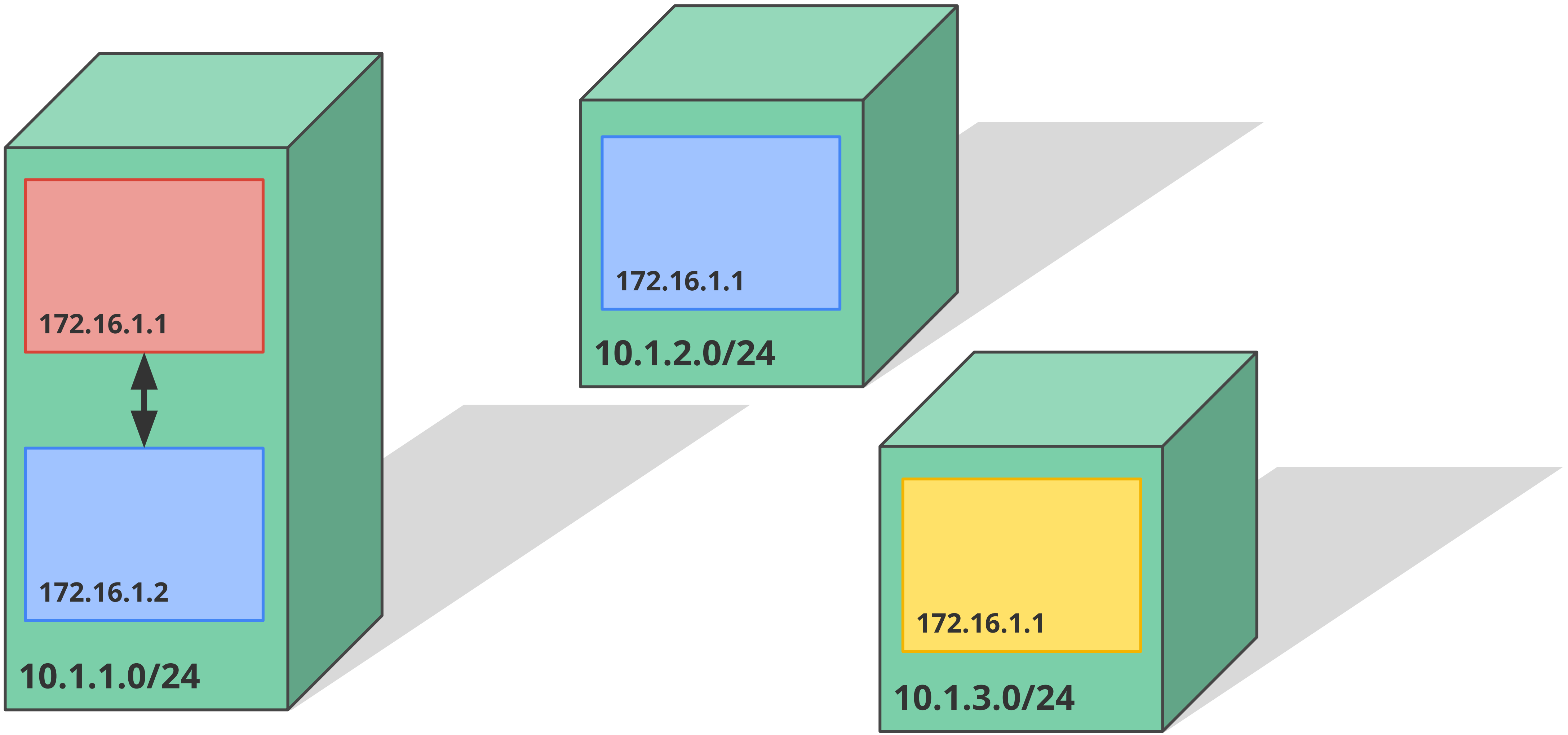
Ephemeral

- can die and be replaced

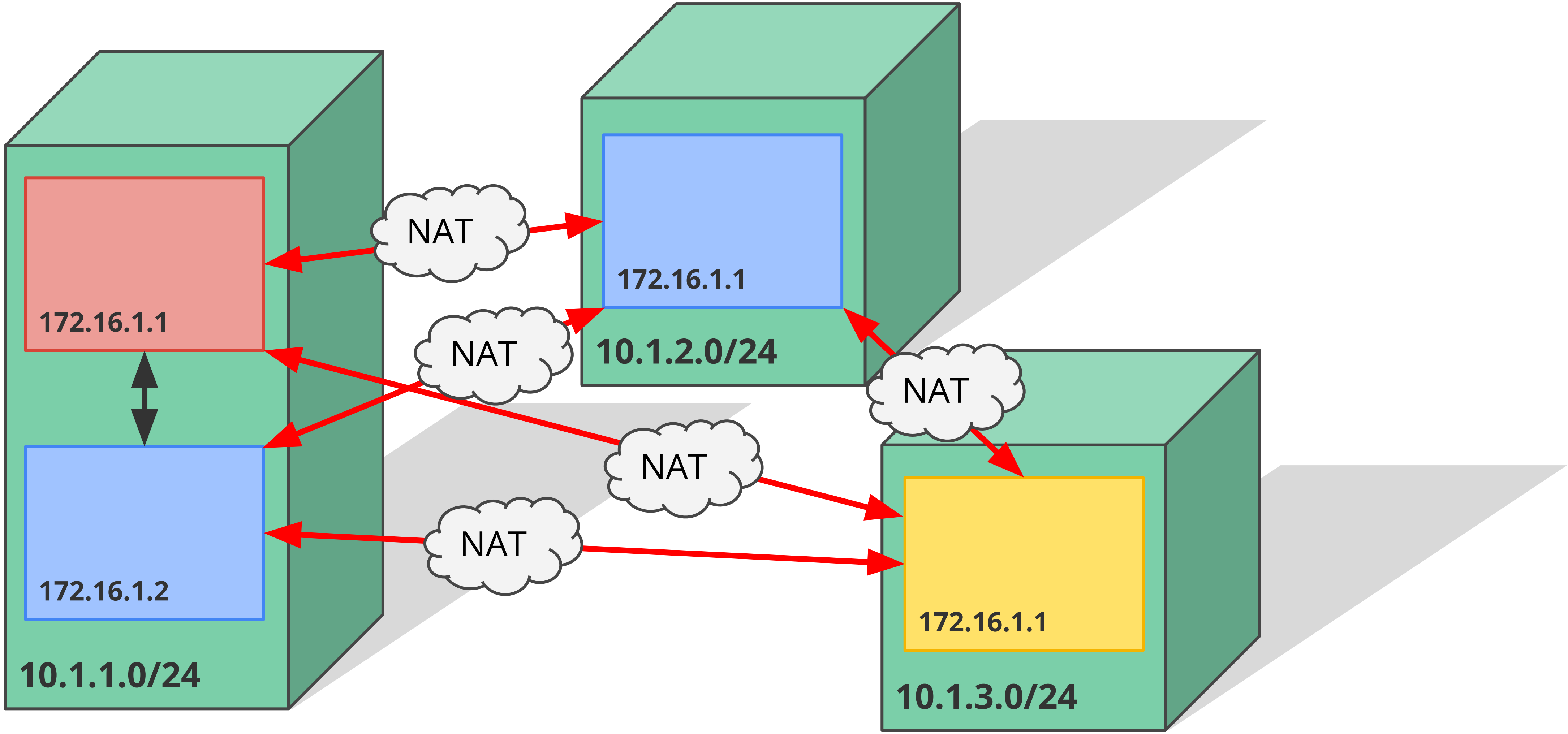
**Example: data puller & web server**



# Docker networking



# Docker networking



# Pod networking

Pod IPs are **routable**

- Docker default is private IP

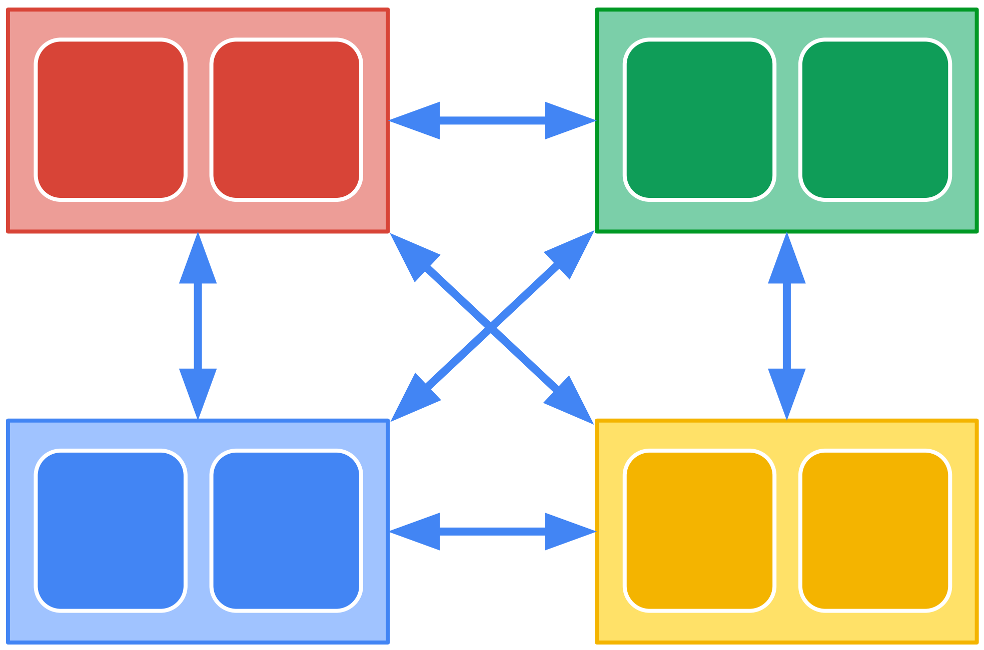
Pods can reach each other without NAT

- even across nodes

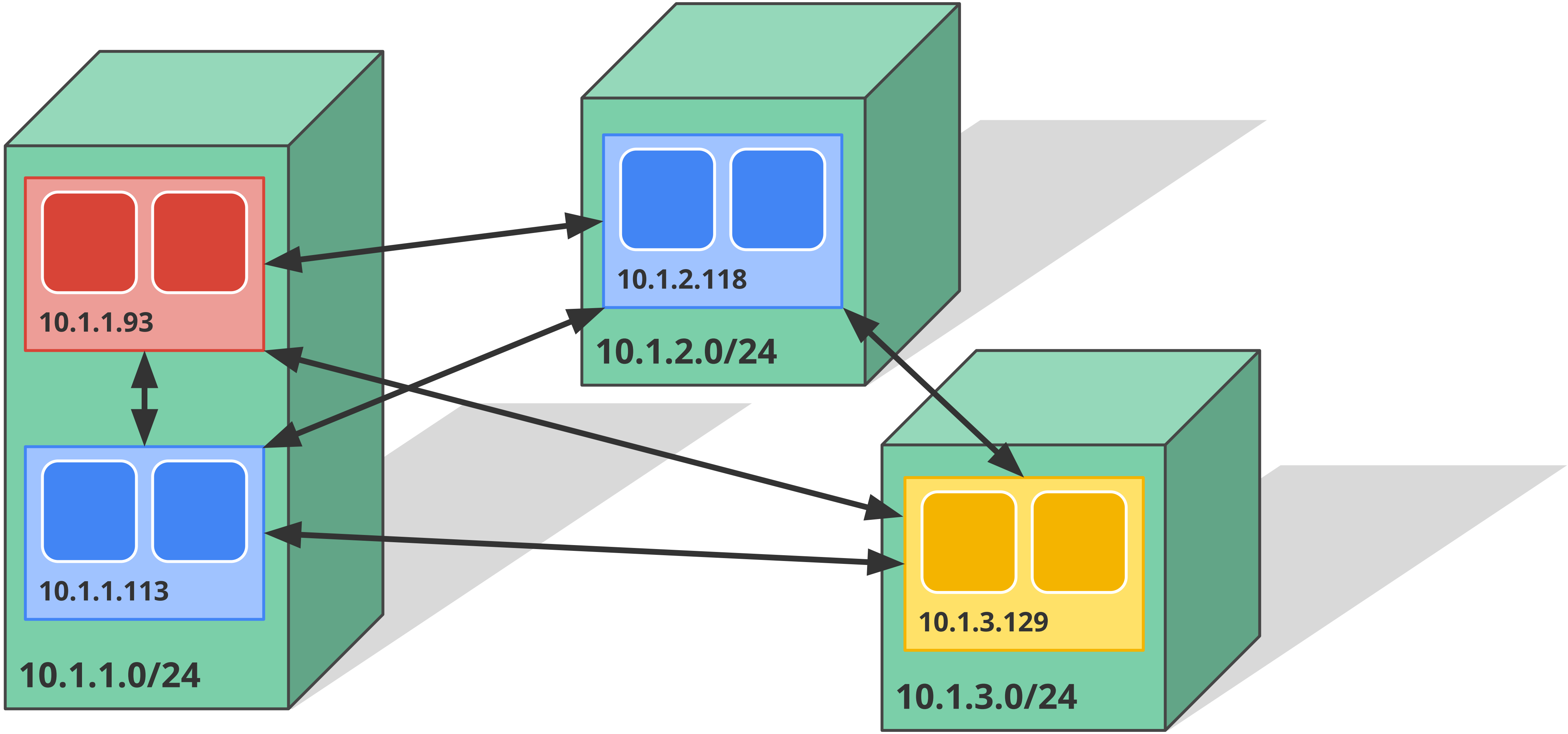
**No brokering** of port numbers

**This is a fundamental requirement**

- several SDN solutions



# Pod networking



# Labels

Arbitrary metadata

Attached to any API object

Generally represent **identity**

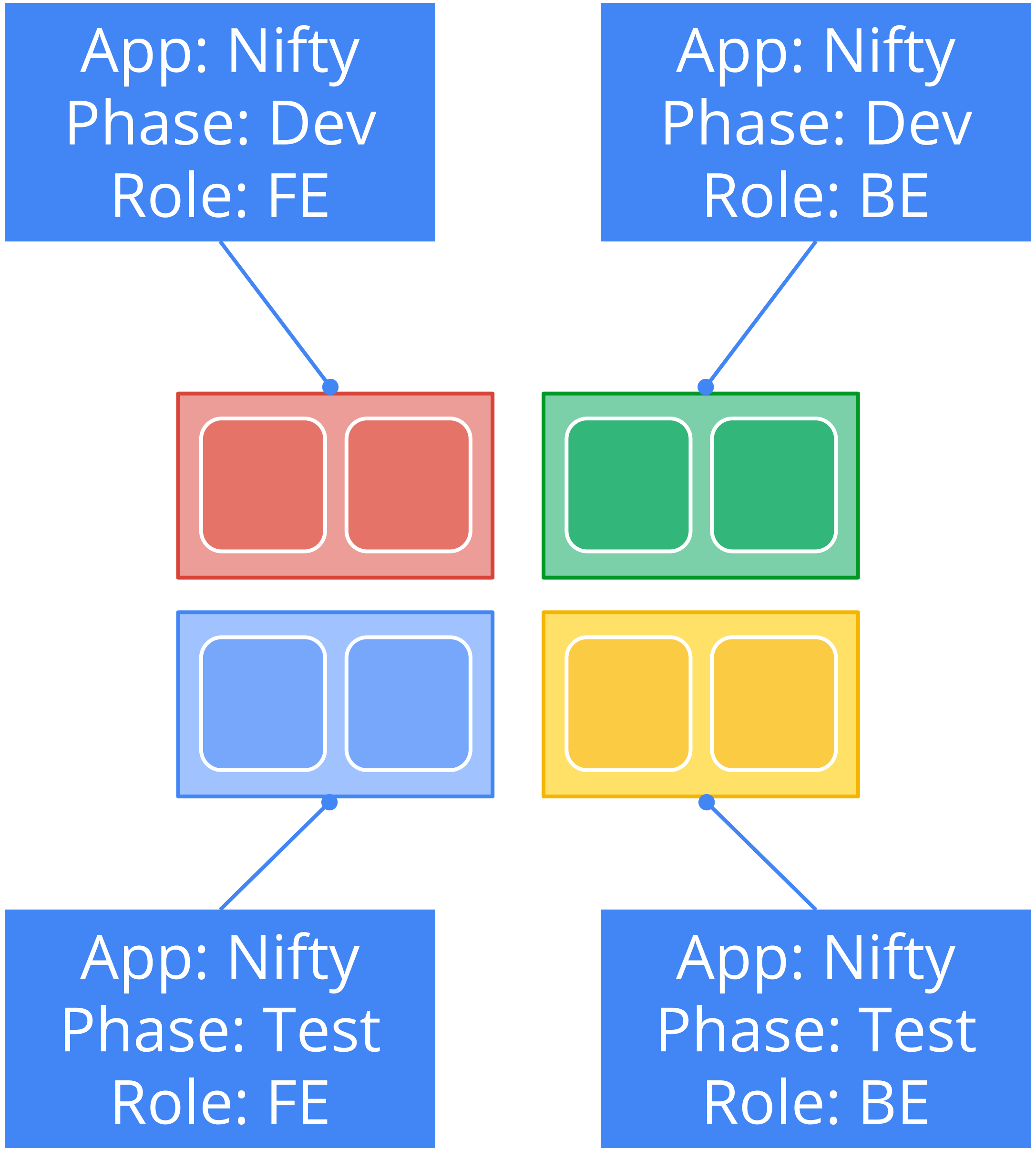
Queryable by **selectors**

- think SQL *'select ... where ...'*

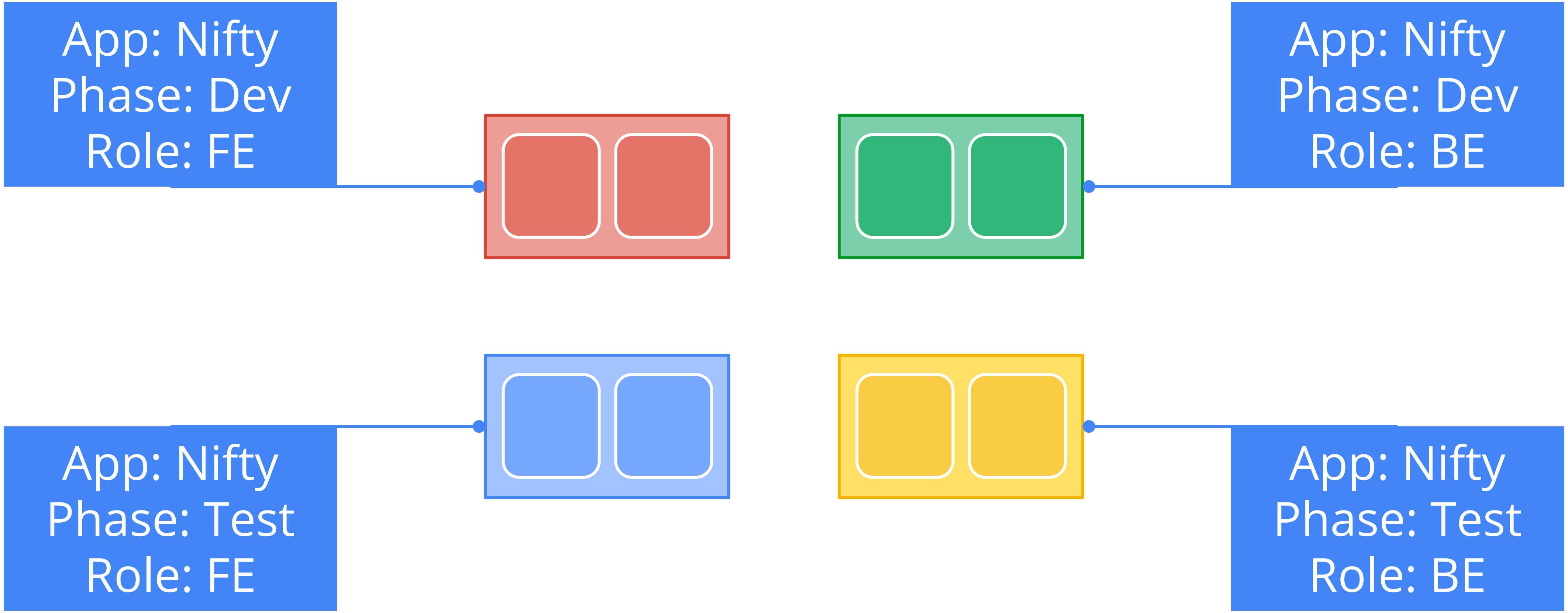
The **only** grouping mechanism

- pods under a ReplicationController
- pods in a Service
- capabilities of a node (constraints)

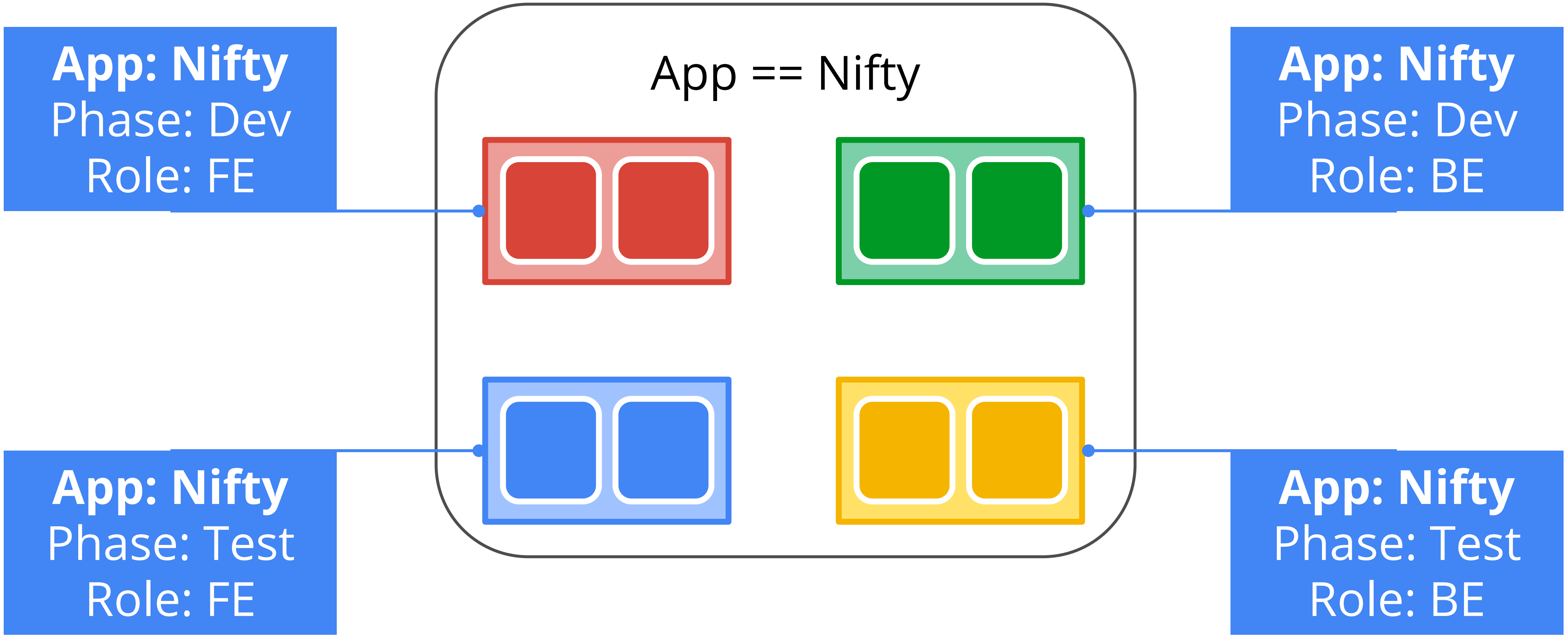
**Example: "phase: canary"**



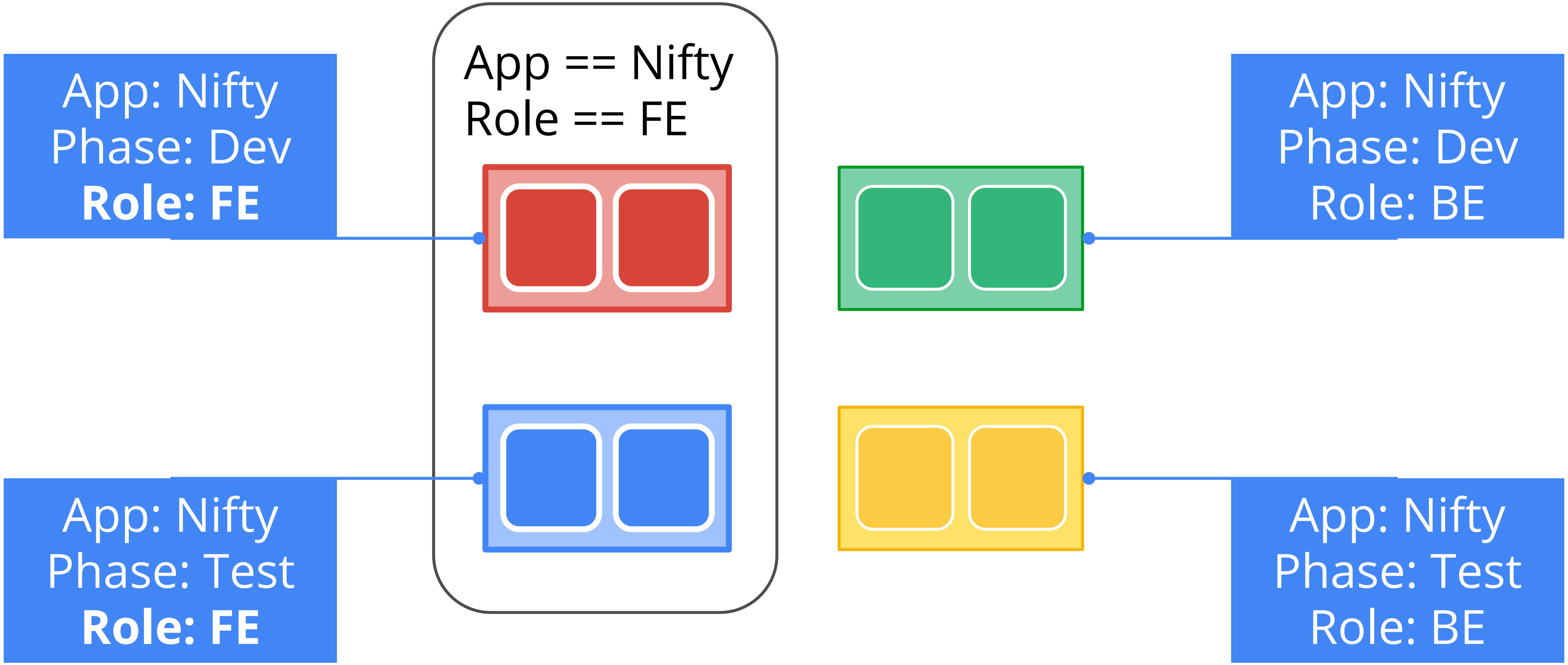
# Selectors


App: Nifty
Phase: Dev
Role: FE
App: Nifty
Phase: Dev
Role: BE
App: Nifty
Phase: Test
Role: FE
App: Nifty
Phase: Test
Role: BE

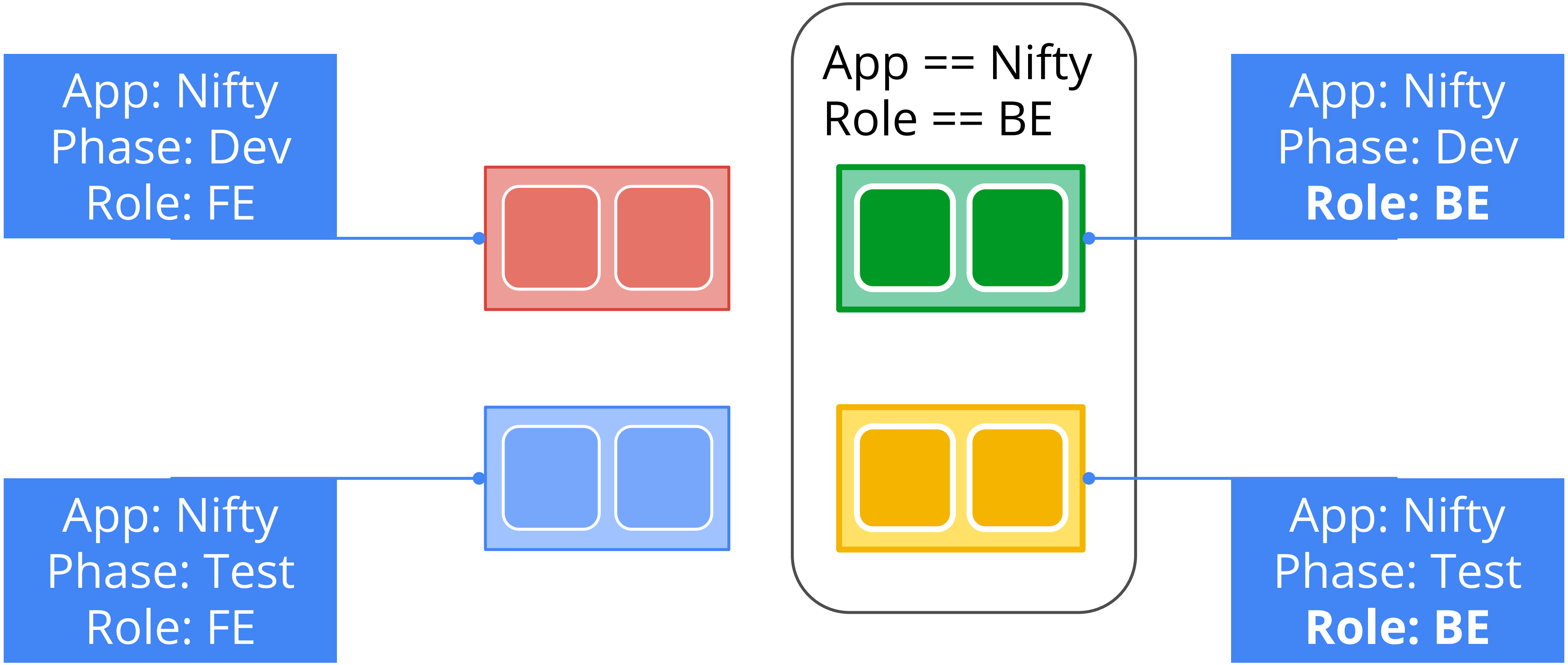
# Selectors


App: Nifty
Phase: Dev
Role: FE
App == Nifty
App: Nifty
Phase: Dev
Role: BE
App: Nifty
Phase: Test
Role: FE
App: Nifty
Phase: Test
Role: BE

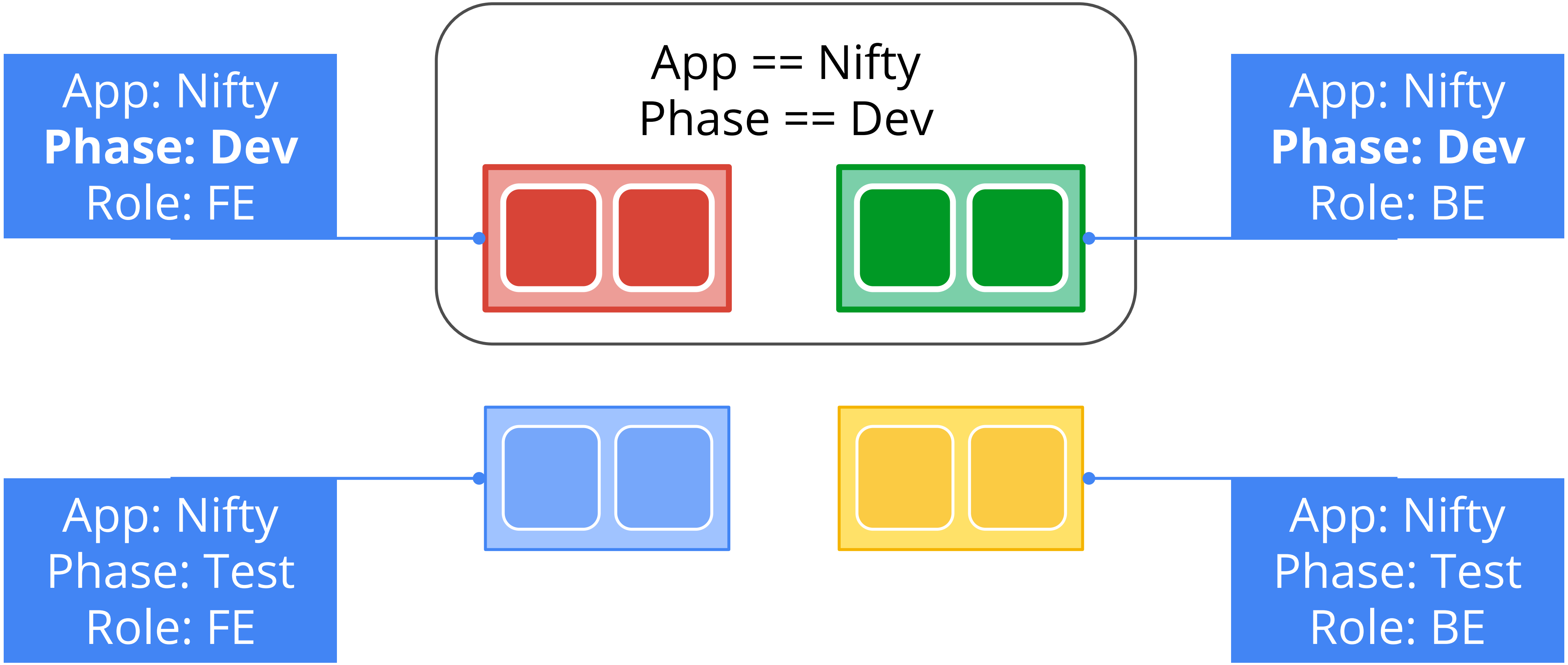
# Selectors


App: Nifty
Phase: Dev
**Role: FE**
App == Nifty
Role == FE
App: Nifty
Phase: Dev
Role: BE
App: Nifty
Phase: Test
**Role: FE**
App: Nifty
Phase: Test
Role: BE

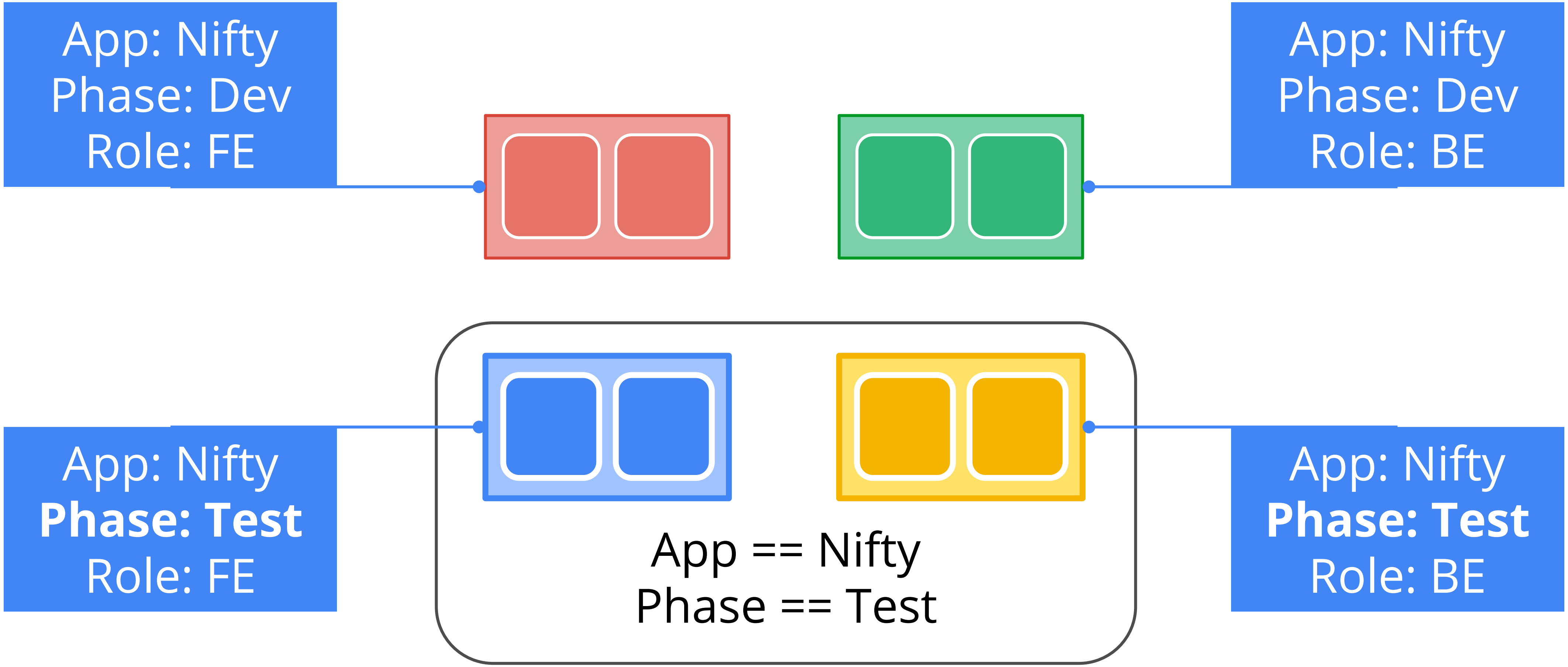
# Selectors


App: Nifty
Phase: Dev
Role: FE
App == Nifty
Role == BE
App: Nifty
Phase: Dev
Role: BE
App: Nifty
Phase: Test
Role: FE
App: Nifty
Phase: Test
Role: BE

# Selectors


App == Nifty
Phase == Dev
App: Nifty
Phase: Dev
Role: FE
App: Nifty
Phase: Dev
Role: BE
App: Nifty
Phase: Test
Role: FE
App: Nifty
Phase: Test
Role: BE

# Selectors


App: Nifty
Phase: Dev
Role: FE
App: Nifty
Phase: Dev
Role: BE
App: Nifty
Phase: Test
Role: FE
App == Nifty
Phase == Test
App: Nifty
Phase: Test
Role: BE

# Replication Controllers

Canonical example of control loops

Runs out-of-process wrt API server

Have 1 job: ensure N copies of a pod

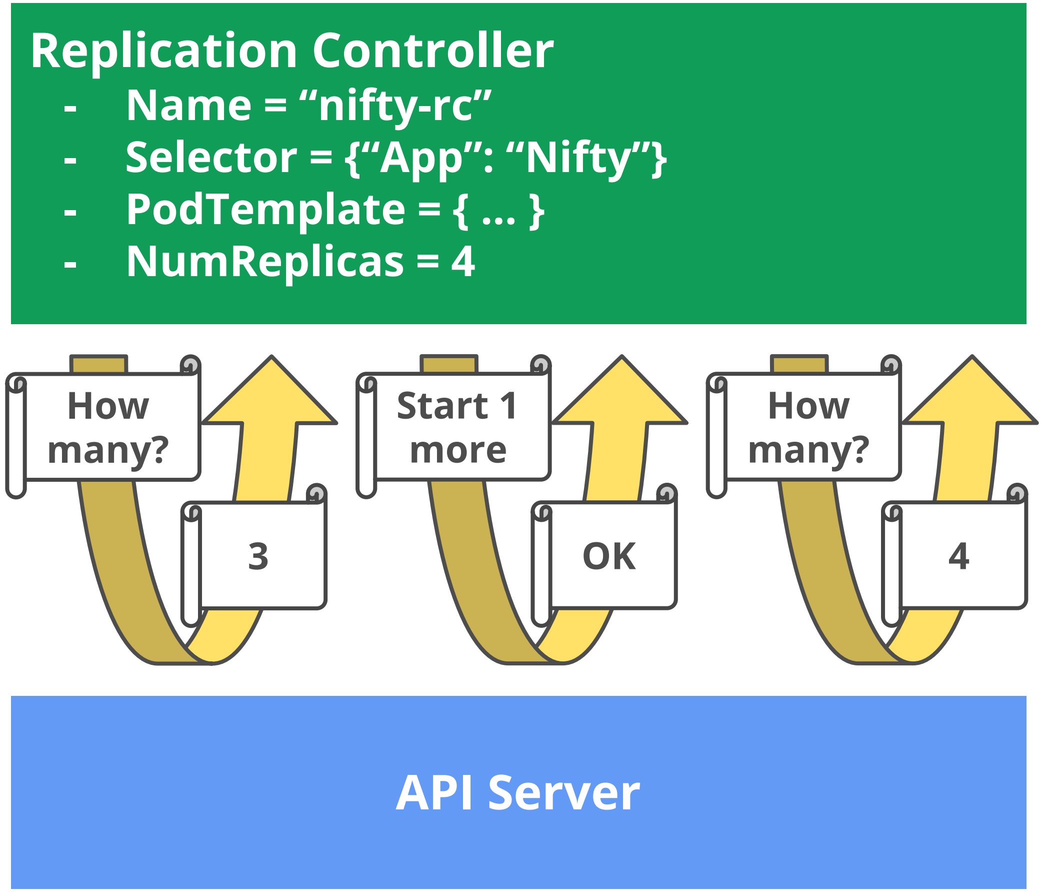
- if too few, start new ones
- if too many, kill some
- group == selector

Cleanly layered on top of the core

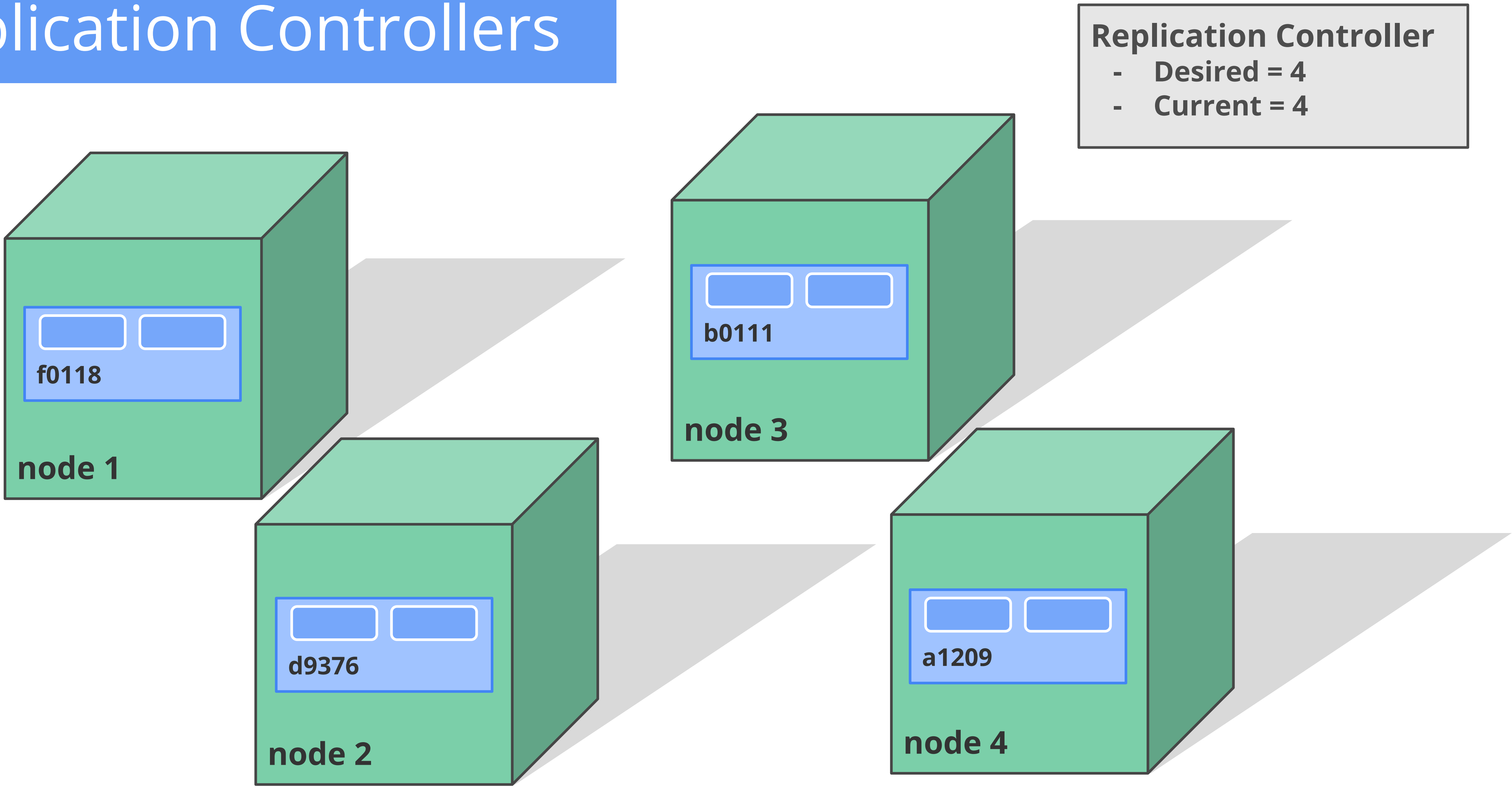
- all access is by public APIs

Replicated pods are fungible

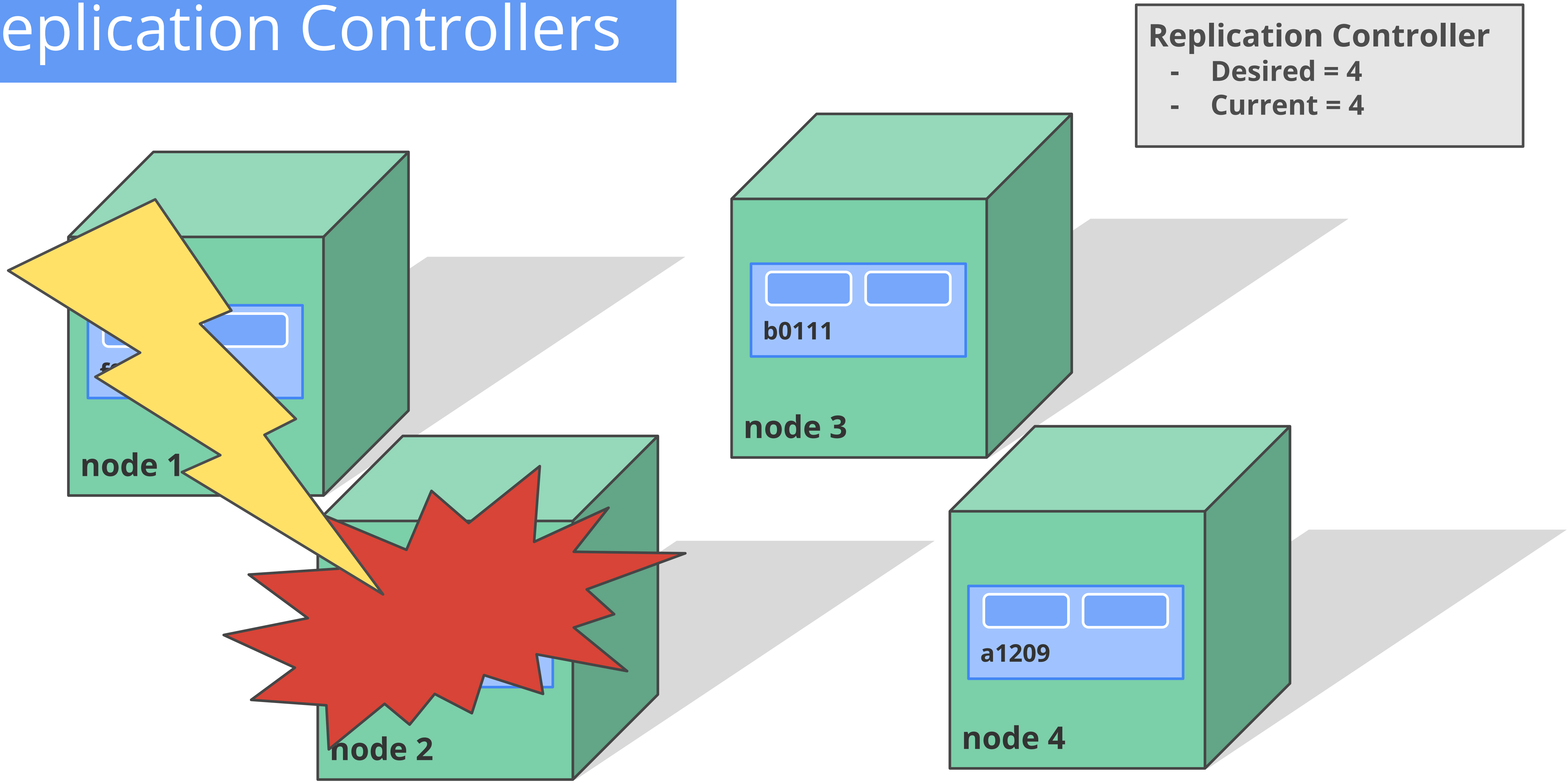
- No implied ordinality or identity



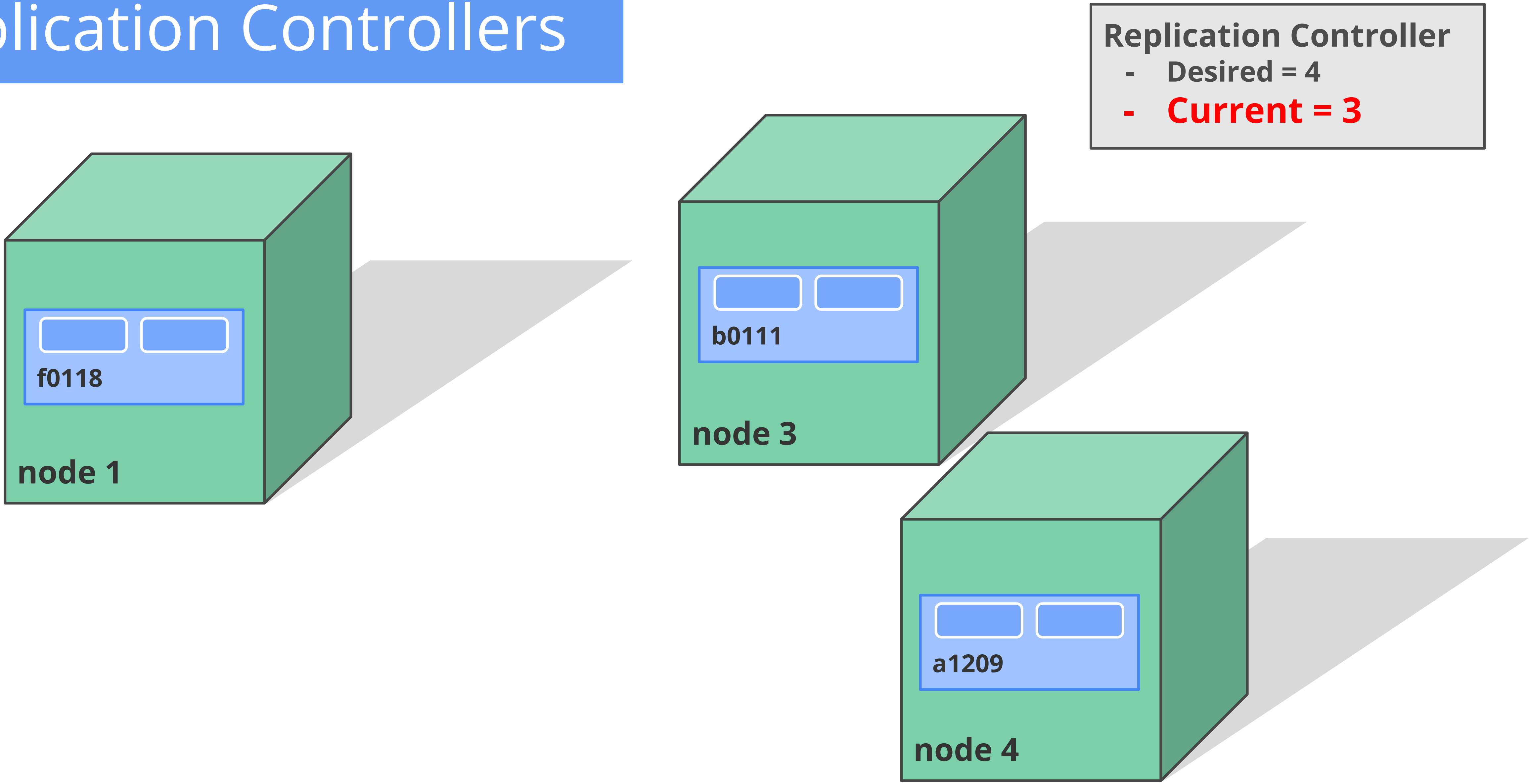
# Replication Controllers



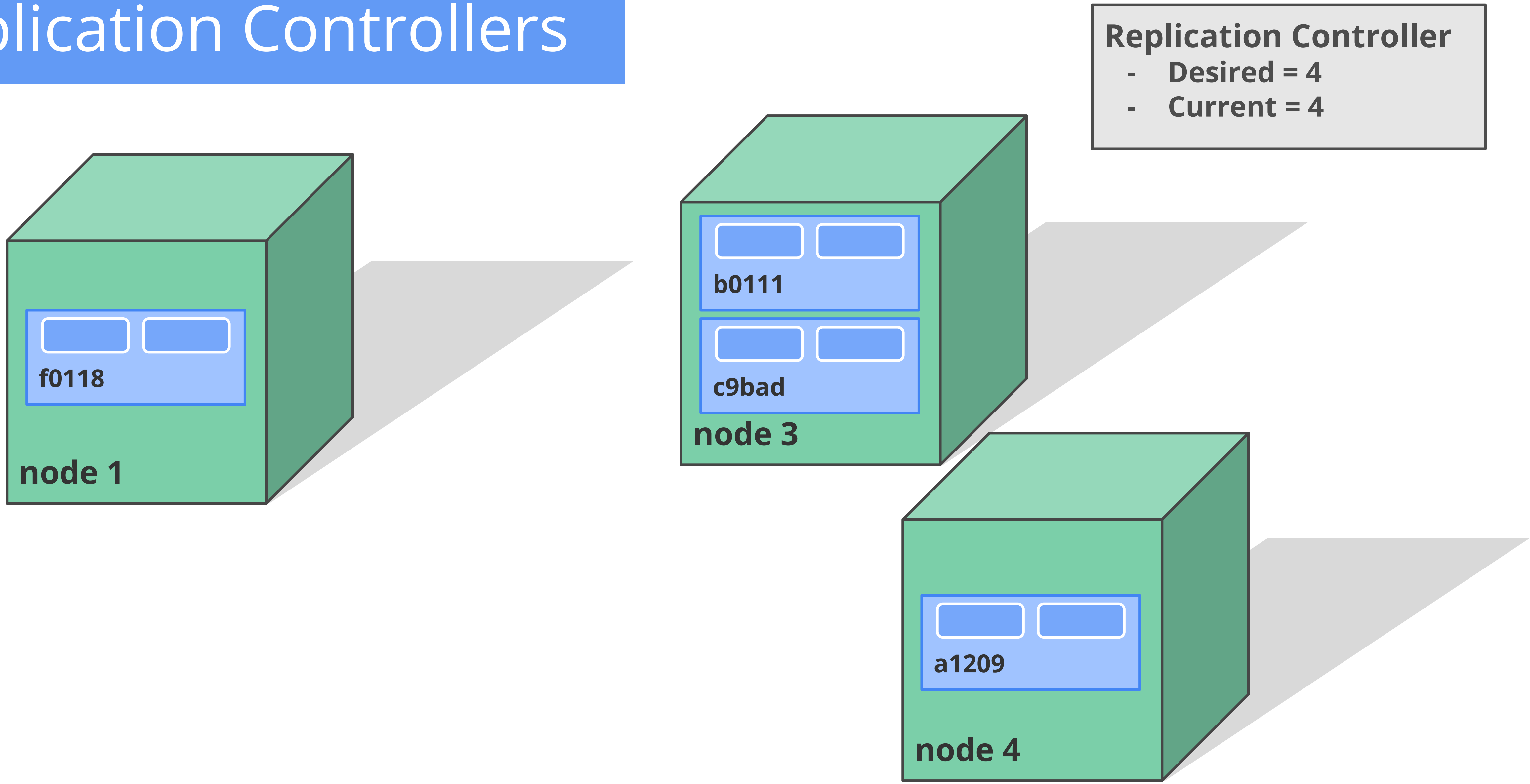
# Replication Controllers



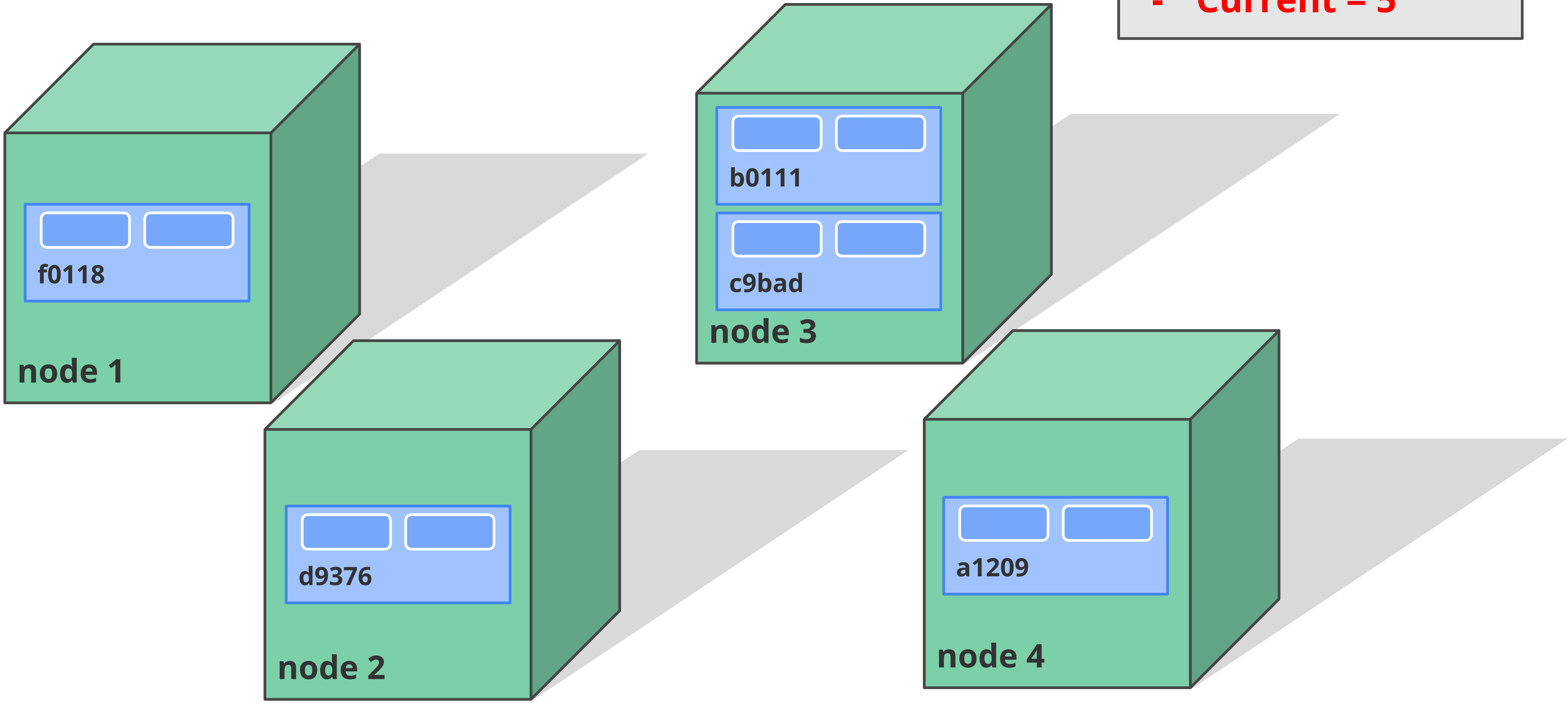
# Replication Controllers



# Replication Controllers

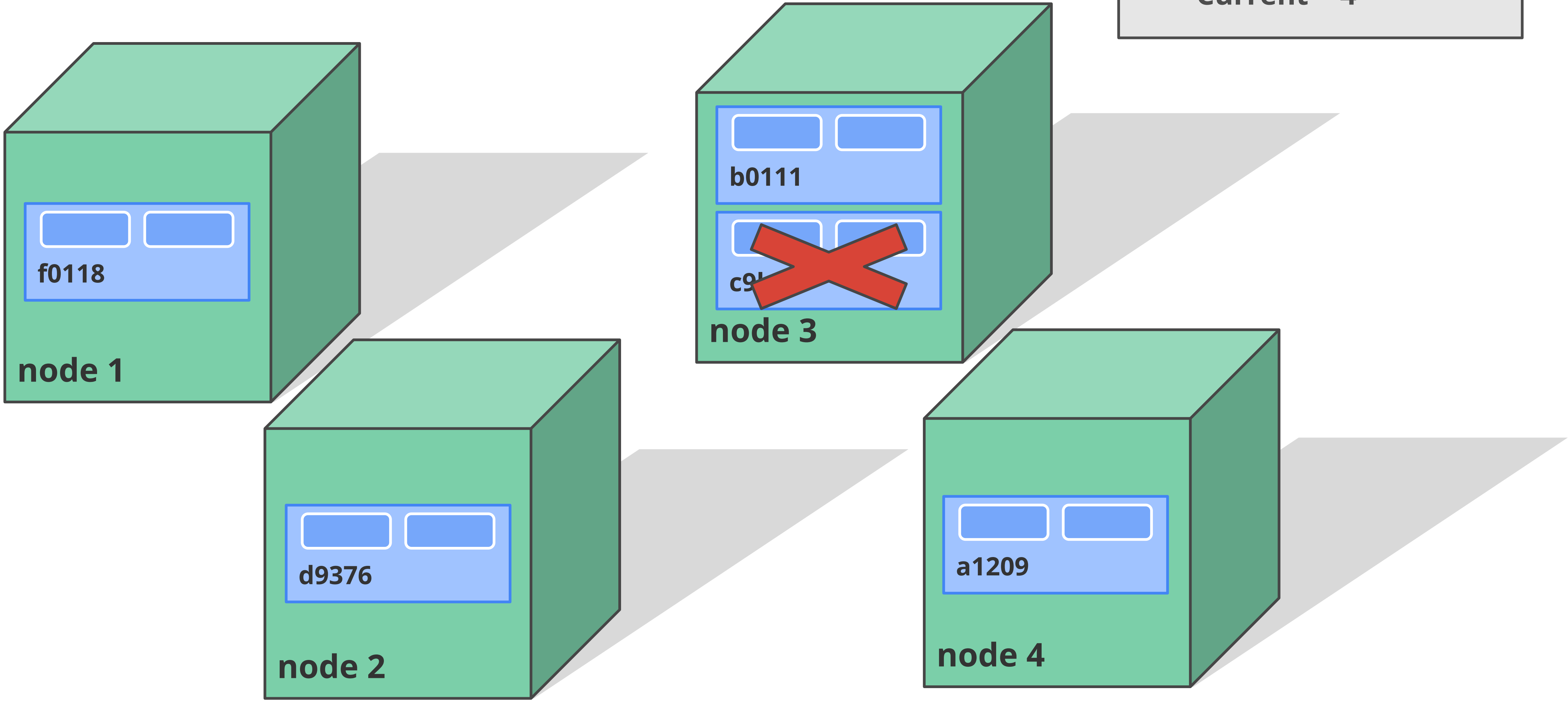


# Replication Controllers




Replication Controller
- Desired = 4
- Current = 5

# Replication Controllers




Replication Controller
- Desired = 4
- Current = 4

# Services

A group of pods that **act as one** == Service

- group == selector

Defines access policy

- only “load balanced” for now

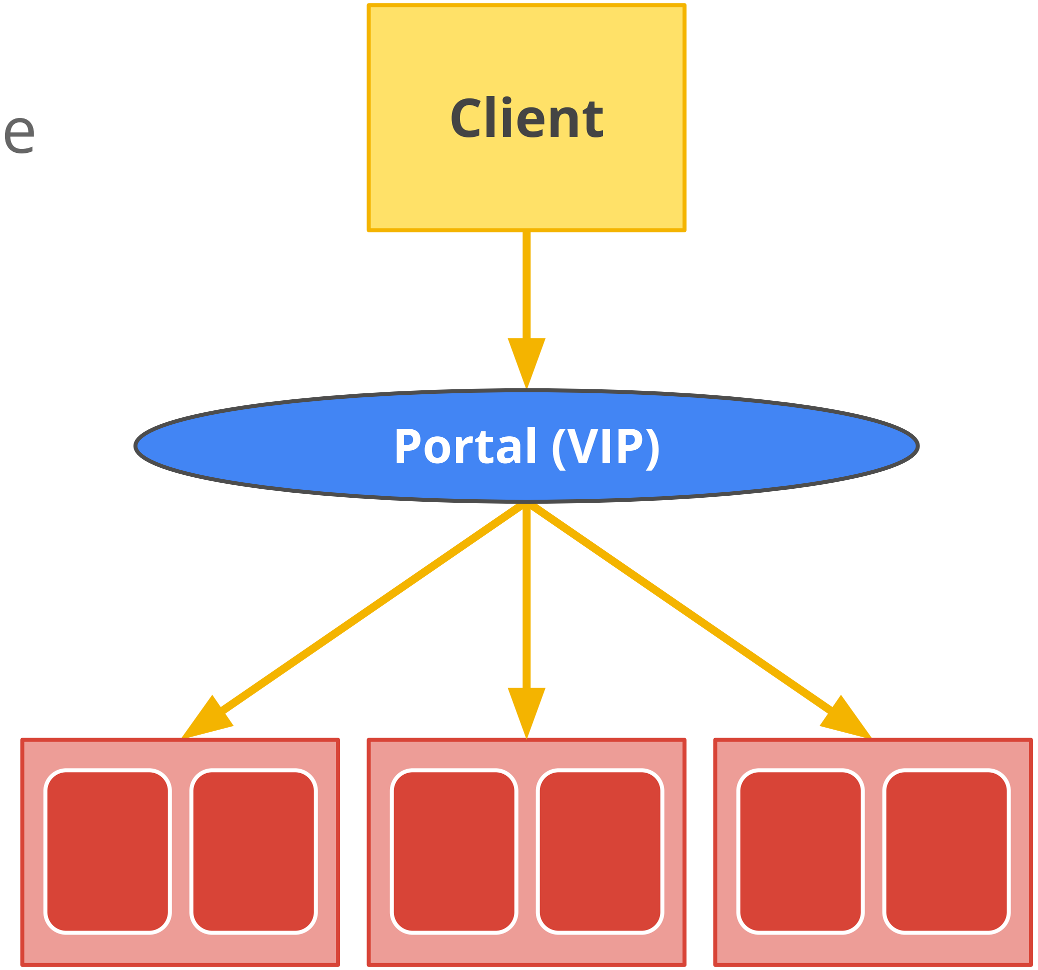
Gets a **stable** virtual IP and port

- called the service *portal*
- also a DNS name

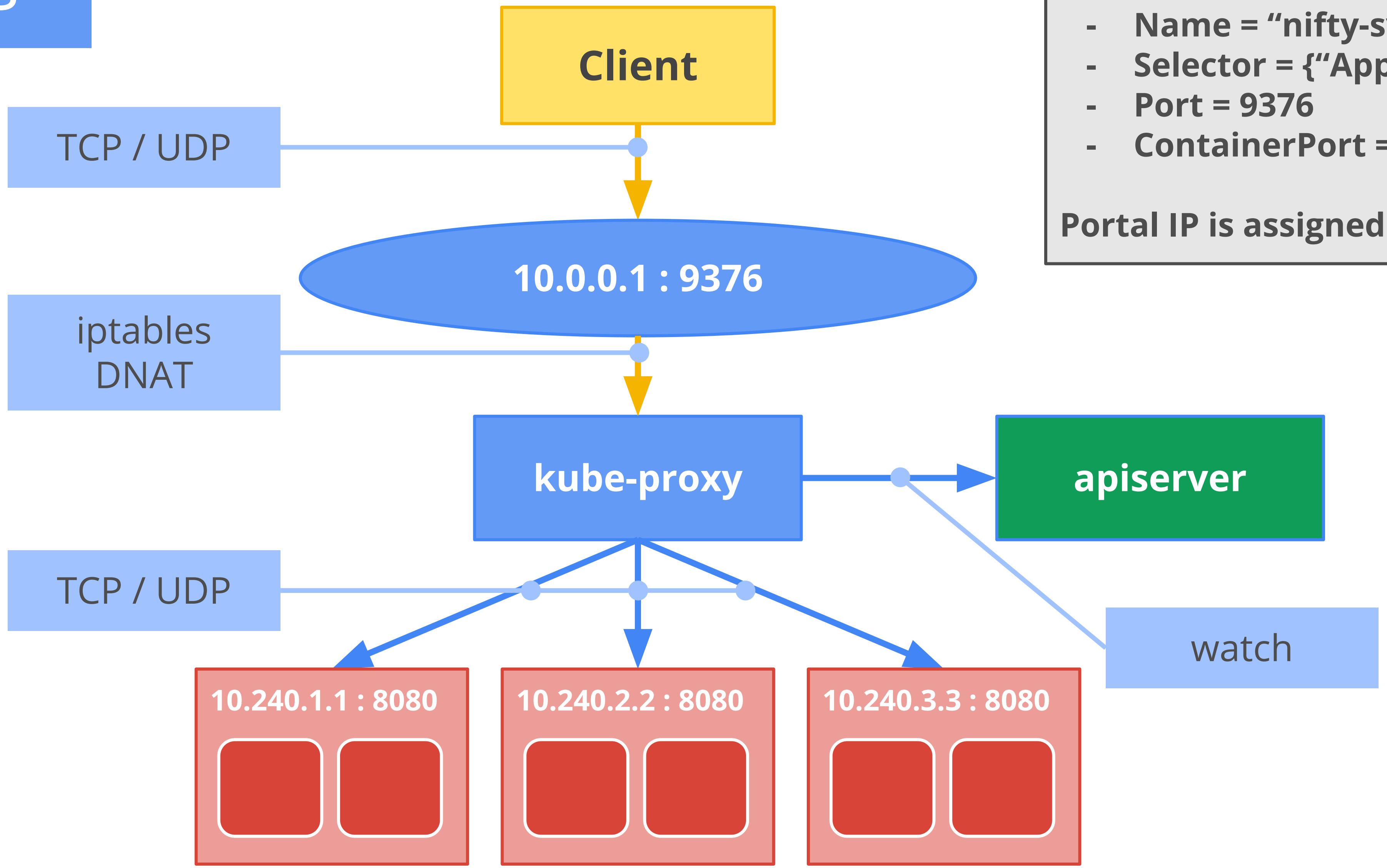
VIP is captured by *kube-proxy*

- watches the service **constituency**
- updates when backends change

Hide complexity - ideal for non-native apps



# Services




**Service**

- Name = “nifty-svc”
- Selector = {“App”: “Nifty”}
- Port = 9376
- ContainerPort = 8080

**Portal IP is assigned**

# Kubernetes Status & plans

Open sourced in June, 2014

- won the BlackDuck “rookie of the year” award
- so did cAdvisor :)

Google launched **Google Container Engine** (GKE)

- hosted Kubernetes
- https://cloud.google.com/container-engine/

Roadmap:

- https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/roadmap.md

Driving towards a 1.0 release in O(months)

- O(100) nodes, O(50) pods per node
- focus on web-like app serving use-cases

# Monitoring

Optional add-on to Kubernetes clusters

Run cAdvisor as a pod on each node

- gather stats from all containers
- export via REST

Run Heapster as a pod in the cluster

- just another pod, no special access
- aggregate stats

Run Influx and Grafana in the cluster

- more pods
- alternately: store in Google Cloud Monitoring

# Logging

## Optional add-on to Kubernetes clusters

## Run fluentd as a pod on each node

- gather logs from all containers
- export to elasticsearch

## Run Elasticsearch as a pod in the cluster

- just another pod, no special access
- aggregate logs

## Run Kibana in the cluster

- yet another pod
- alternately: store in Google Cloud Logging

# Kubernetes and isolation

We support isolation…

- …inasmuch as Docker does

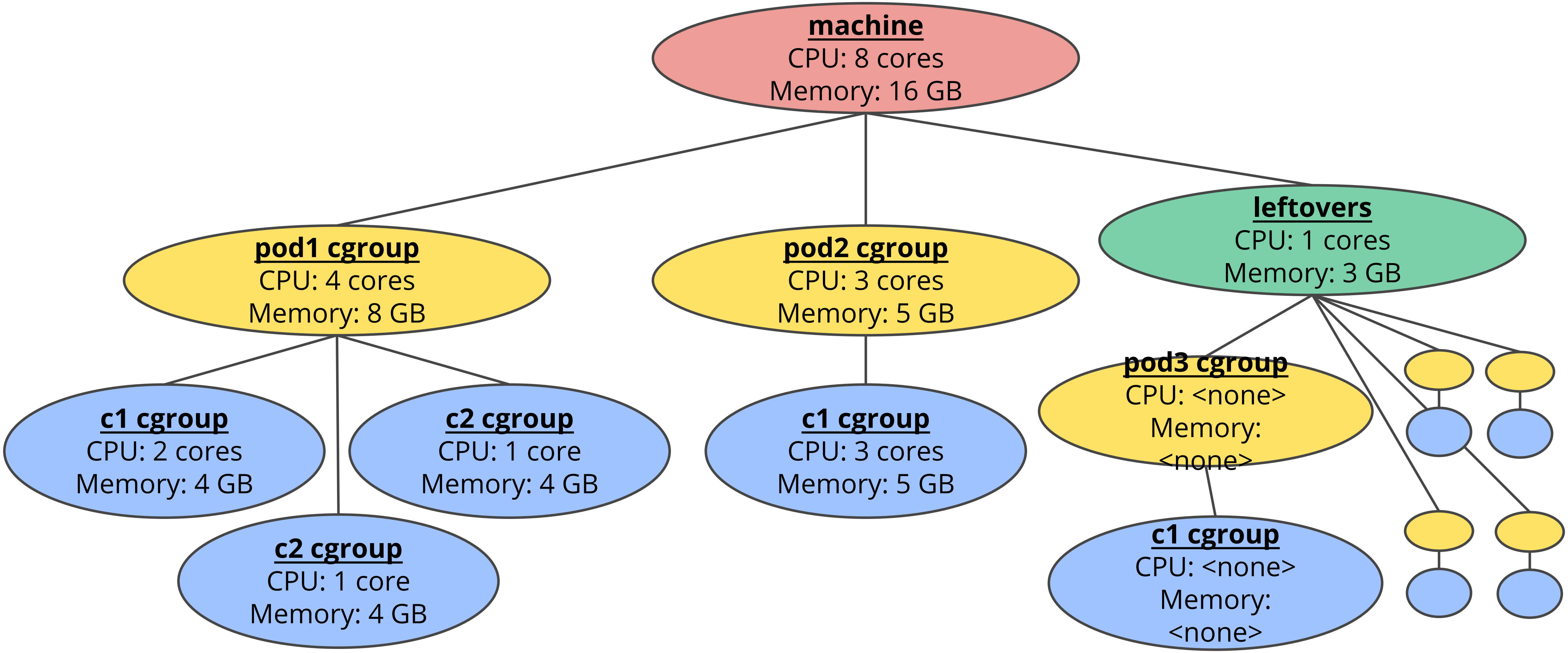
We want better isolation

- issues are open with Docker
  - parent cgroups, GIDs, in-place updates,
- will also need kernel work
- we have lots of tricks we want to share!

We have to **meet users where they are**

- strong isolation is new to most people
- we'll all have to grow into it

# Example: nested cgroups



# The Goal: Shake things up

Containers is a **new way of working**

Requires new concepts and new tools

Google has a **lot** of experience...

...but we are **listening to the users**

**Workload portability is important!**

# Kubernetes is **Open Source**

## We want your help!

http://kubernetes.io
https://github.com/GoogleCloudPlatform/kubernetes
irc.freenode.net *#google-containers*
@kubernetesio

# Questions?

http://kubernetes.io

Images by Connie Zhou

# Backup Slides

# Control loops

Drive **current state** -> **desired state**

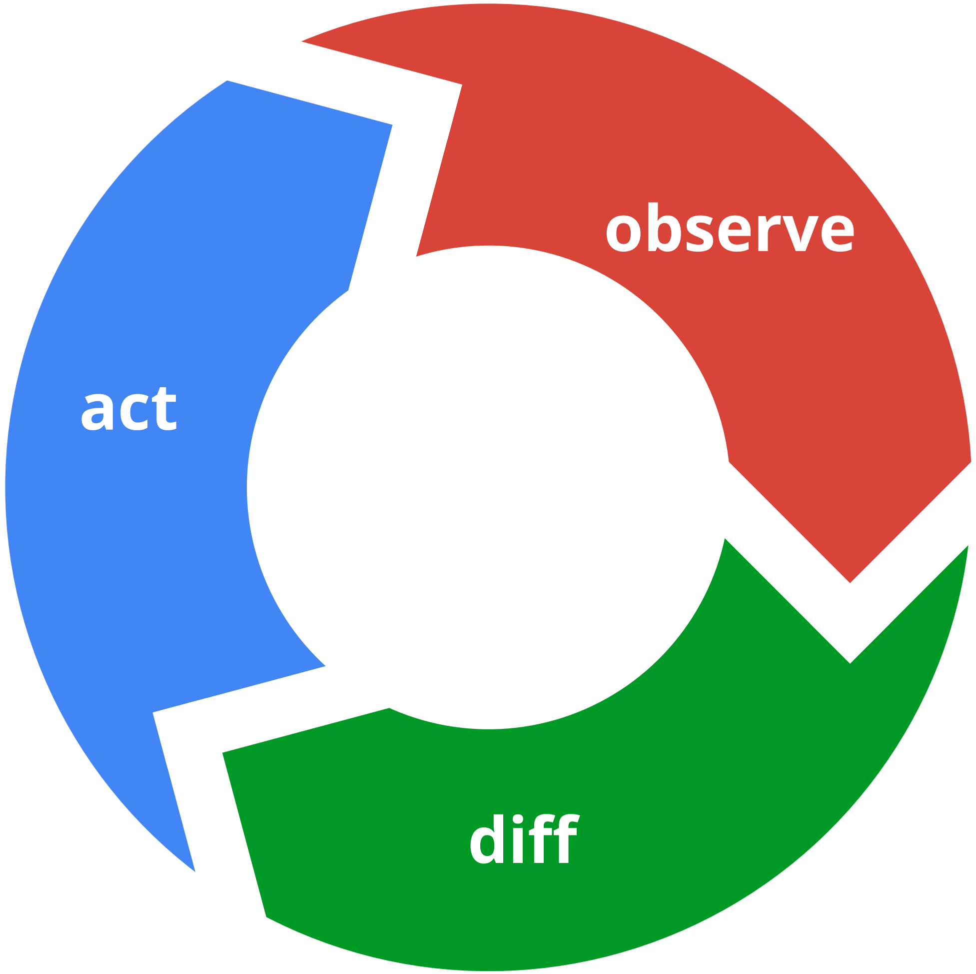
Act independently

APIs - **no shortcuts** or back doors

Observed state is truth

Recurring pattern in the system

**Example: ReplicationController**



# Modularity

Loose coupling is a goal **everywhere**

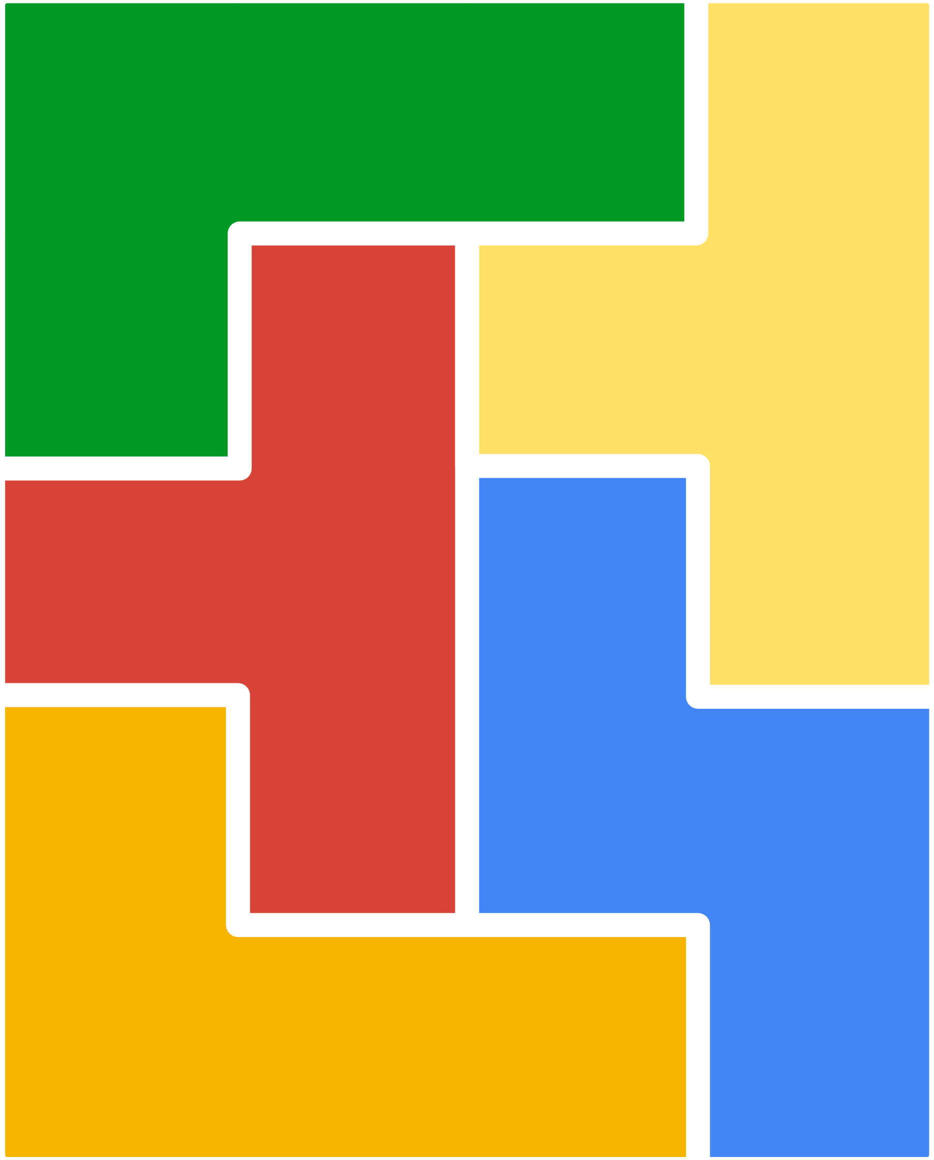
- simpler
- composable
- extensible

Code-level plugins where possible

Multi-process where possible

Isolate risk by interchangeable parts

**Example: ReplicationController**
**Example: Scheduler**



# Atomic storage

Backing store for all master state

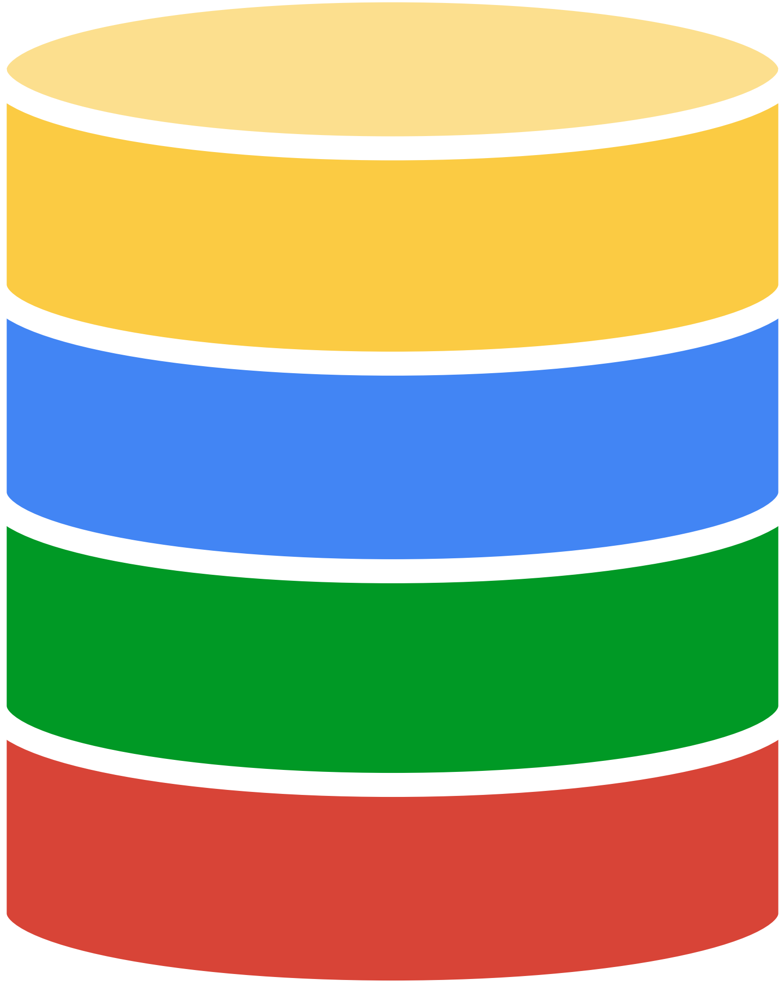
Hidden behind an abstract interface

Stateless means **scalable**

**Watch**able

- this is a fundamental primitive
- don't poll, watch

Using **CoreOS etcd**



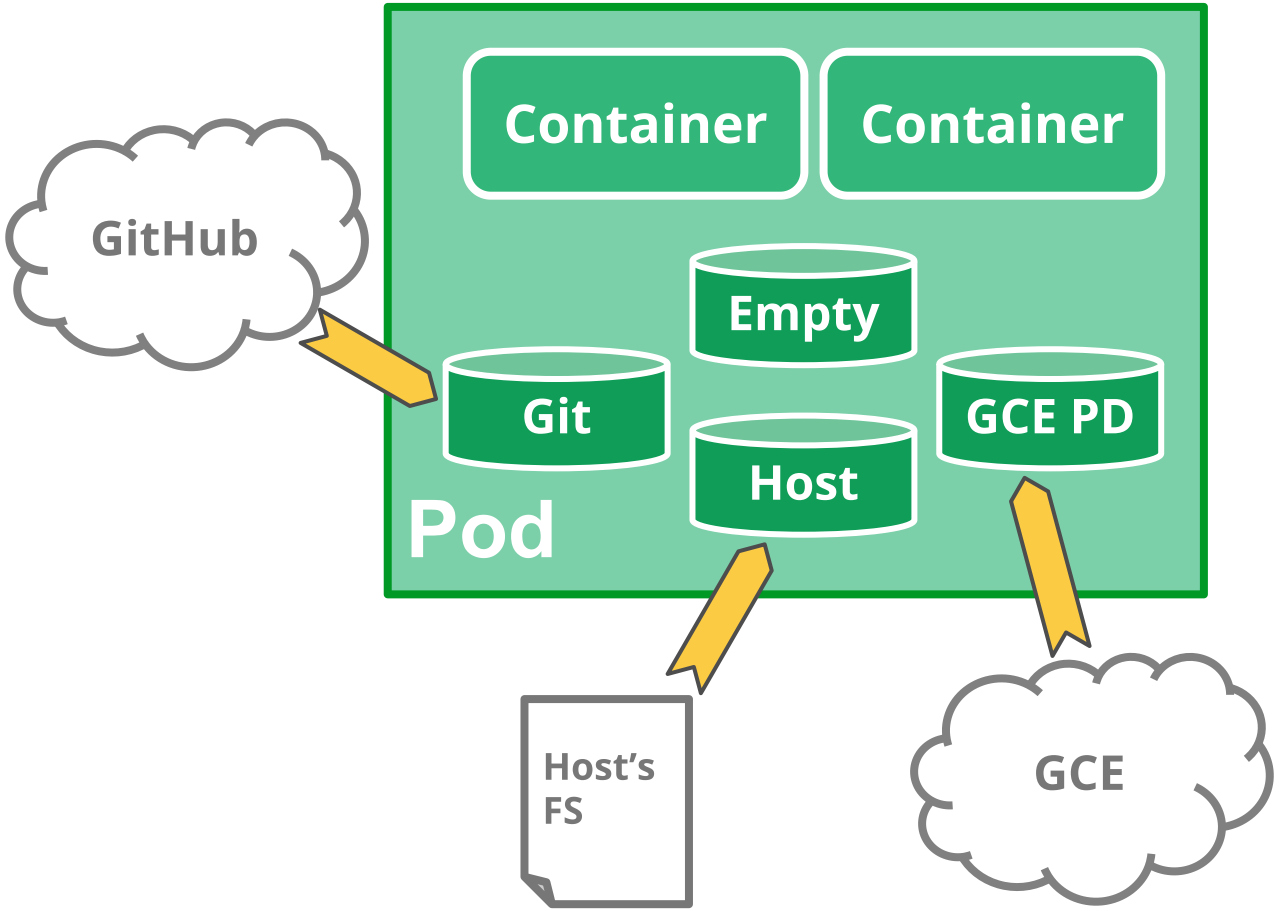
# Volumes

Pod scoped

Share pod’s lifetime & fate

Support various types of volumes

- Empty directory (default)
- Host file/directory
- Git repository
- GCE Persistent Disk
- …more to come, suggestions welcome



# Pod lifecycle

Once scheduled to a node, pods do not move

- restart policy means restart **in-place**

Pods can be observed *pending*, *running*, *succeeded*, or *failed*

- *failed* is **really** the end - no more restarts
- no complex state machine logic

Pods are **not rescheduled** by the scheduler or apiserver

- even if a node dies
- controllers are responsible for this
- keeps the scheduler **simple**

Apps should consider these rules

- Services hide this
- Makes pod-to-pod communication more formal

# Cluster services

Logging, Monitoring, DNS, etc.

All run as pods in the cluster - no special treatment, no back doors

Open-source solutions for everything

- cadvisor + influxdb + heapster == cluster monitoring
- fluentd + elasticsearch + kibana == cluster logging
- skydns + kube2sky == cluster DNS

Can be easily replaced by custom solutions

- Modular clusters to fit your needs