# Fundamentals of Istio

# Logistics

- Class Hours:
- Start time is 8:30am
- End time is 3:30pm
- Class times may vary slightly for specific classes
- Breaks mid-morning and afternoon (10 minutes)

- Lunch:
- Lunch is 11:30am to 1pm
- Yes, 1 hour and 30 minutes
- Extra time for email, phone calls, or simply a walk.

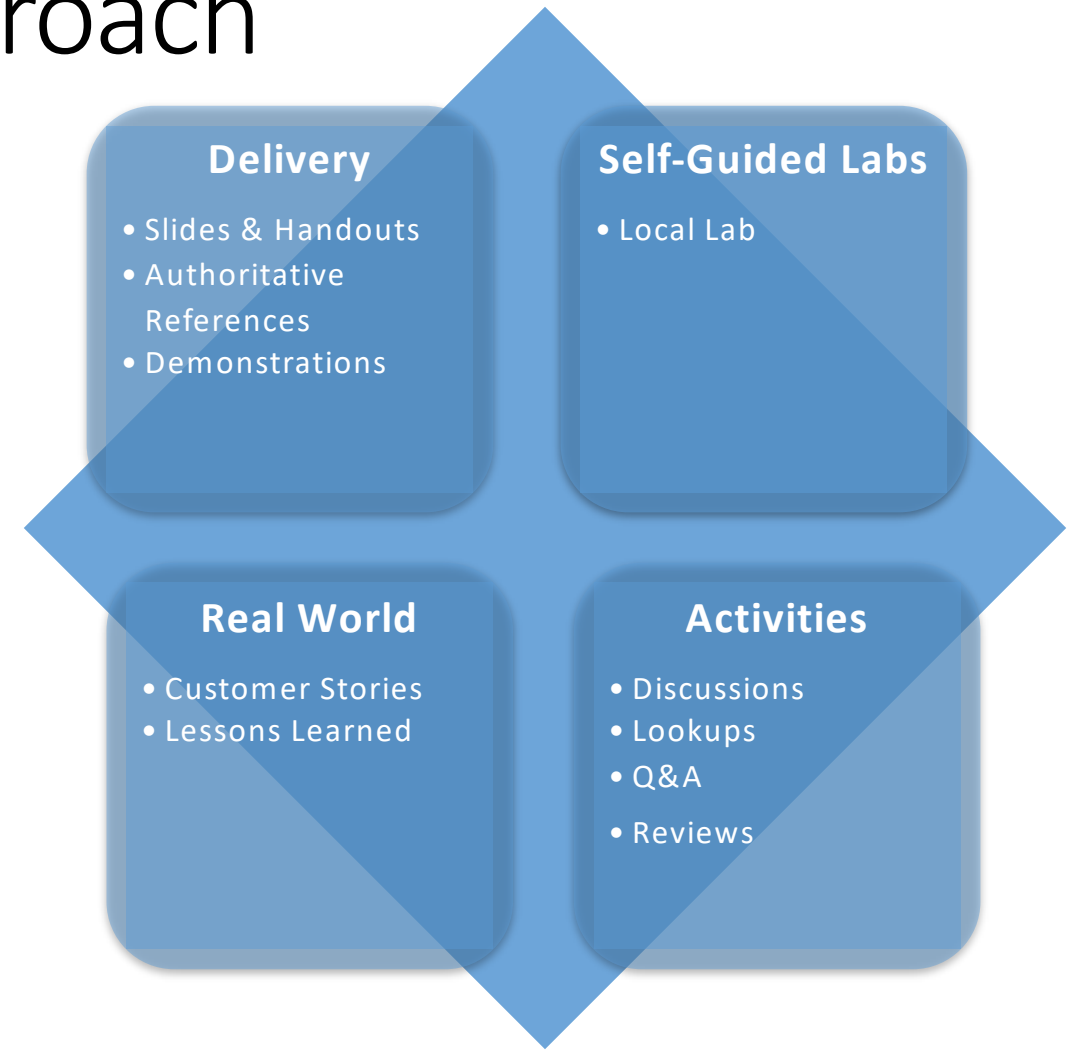
- Telecommunication:
- Turn off or set electronic devices to vibrate
- Reading or attending to devices can be distracting to other students

- Miscellaneous
- Courseware
- Bathroom

# Course Objectives

- Monolithic vs Microservices
- Challenges of microservices
- Overview of Kubernetes components
  - Pods
  - Services
  - Deployments
- Install and configure Kubernetes
- How Istio simplifies microservice communications
- Install and configure Istio
- Analyze microservice communications

# Course Approach




**End of Day Office Hours - 1:1 Assistance**

# The Training Dilemma

# Meet the Instructor

Jason Smith

Cloud Consultant with a Linux sysadmin background.
Focused on cloud-native technologies: automation, containers & orchestration

Twitter
@jruels

github
https://github.com/jruels

mail
jason@innovationinsoftware.com

Expertise

- Cloud
- Automation
- CICD
- Docker
- Kubernetes

# Introductions

- Name
- Job Role
- Which statement best describes your Istio/Kubernetes experience?
  a. I am ***currently working*** with Istio/Kubernetes on a project/initiative
  b. I ***expect to work*** with Istio/Kubernetes on a project/initiative in the future
  c. I am ***here to learn*** about Istio/Kubernetes outside of any specific work related project/initiative
- Expectations for course (please be specific)

# What are Monoliths?

# Monolithic

- Simple to develop
  - IDEs designed for building one application
- Simple to test
  - Selenium
- Easy to deploy
  - Copy packaged application onto server
- Easy to scale by running multiple copies behind Load Balancer
- Works well in the beginning

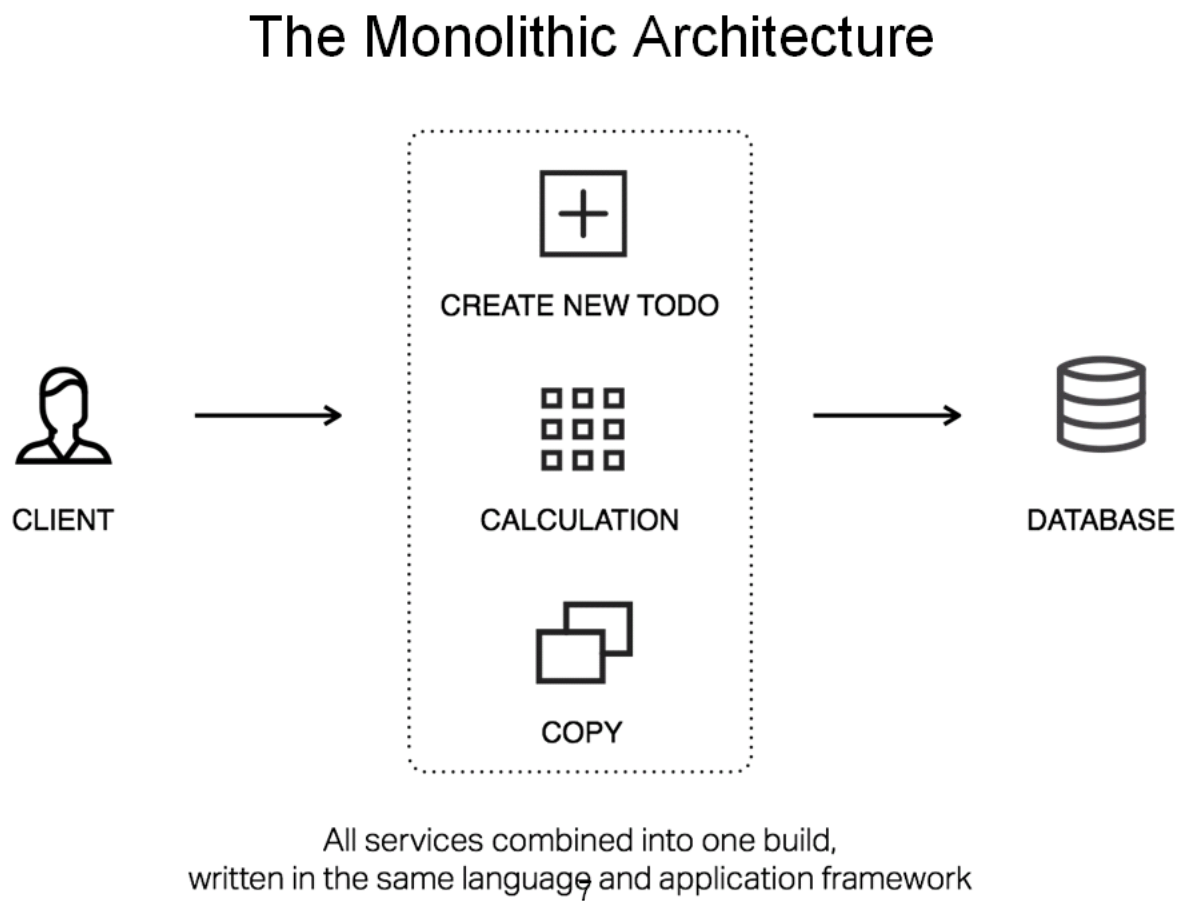
# Monolithic

- Designed to be self contained
- Inter-connected services
- Tightly coupled components
- Must test all components if anything is changed
- Slower deploys
- Slower scaling

# Monolithic

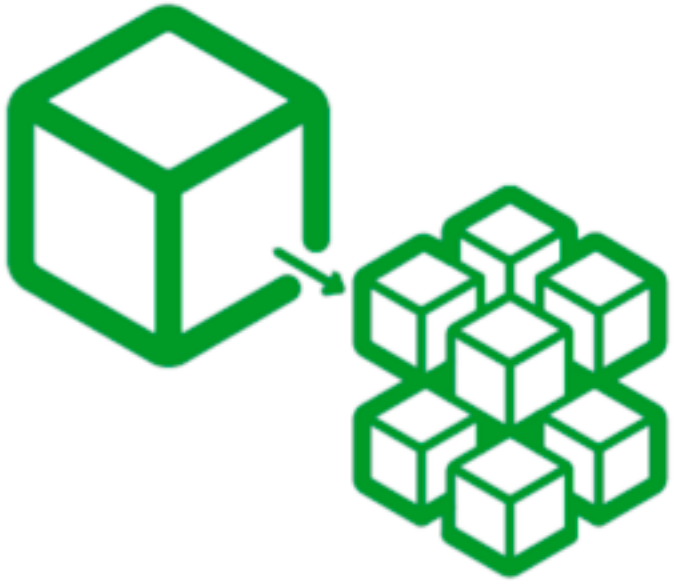
- Single development stack
- Changes are difficult
- Afraid to update

# Monolithic

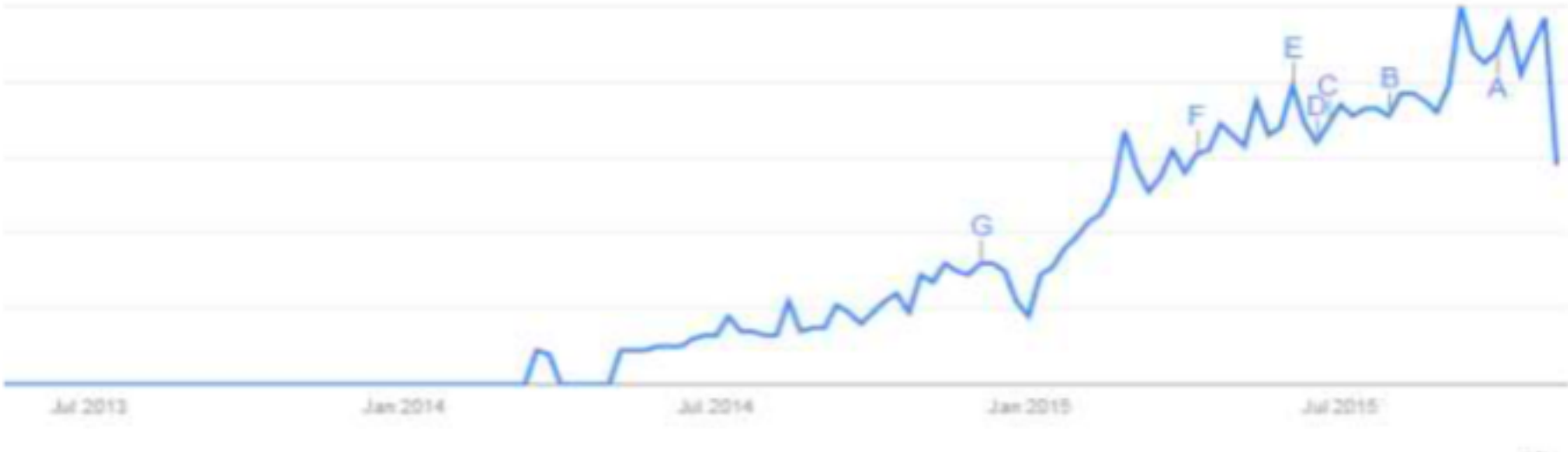

The Monolithic Architecture
CREATE NEW TODO
CLIENT
CALCULATION
DATABASE
COPY
All services combined into one build,
written in the same language and application framework

# What are Microservices?

# What are Microservices?



The term microservices was coined in 2012 by a group of software architects
It started gaining in popularity in 2014 when Martin Fowler wrote a blog post describing it.



# What are Microservices?

- Services communicate with each other over the network.
- Update services independently.
  - No need to change other services.
- Self-contained.
  - You can update the code without knowing anything about internals of other microservices.

# What are Microservices?


**Functional decomposition of systems into manageable and independently deployable components,**

Microservice Architectures by Dr. Andreas Schroeder

"Service-oriented architecture composed of loosely coupled elements that have bounded contexts"

- Adrian Cockcroft (VP Cloud Architecture at AWS)

# What are Microservices?

"In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API."

- Martin Fowler

# Guidelines for Microservices

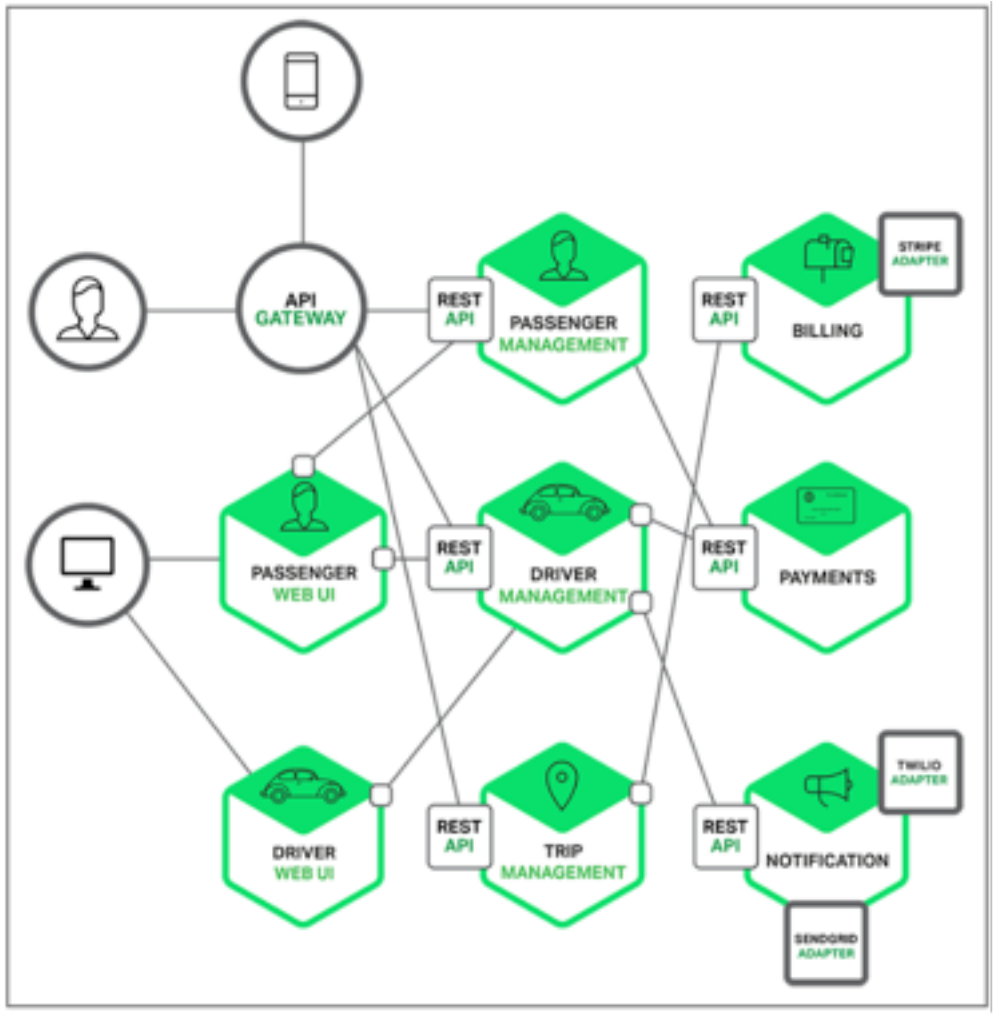
- Make each one focused
  - “2 pizza teams”
- Treat each service as one app
  - Have one SCM repo, own pipeline, etc…
- Each service should be independent of other services
- Service owners are responsible for entire lifecycle
- Use lightweight protocols
- Use the right tool for the job

# Guidelines for Microservices

- Each service should have it’s own datastore
- Failure will happen, design for it
- Automate everything!

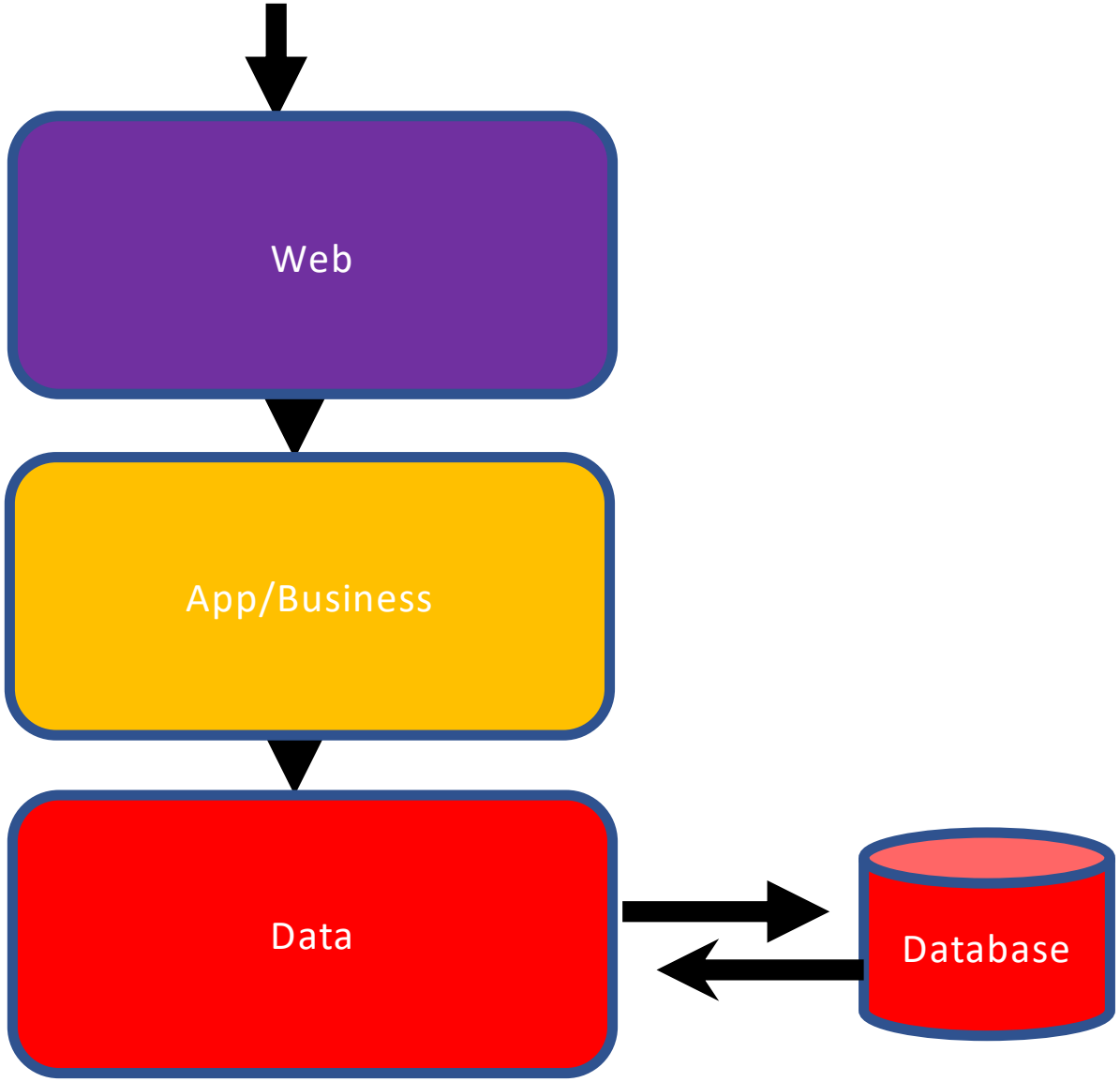
# Monolithic Architecture

# Microservice Architecture



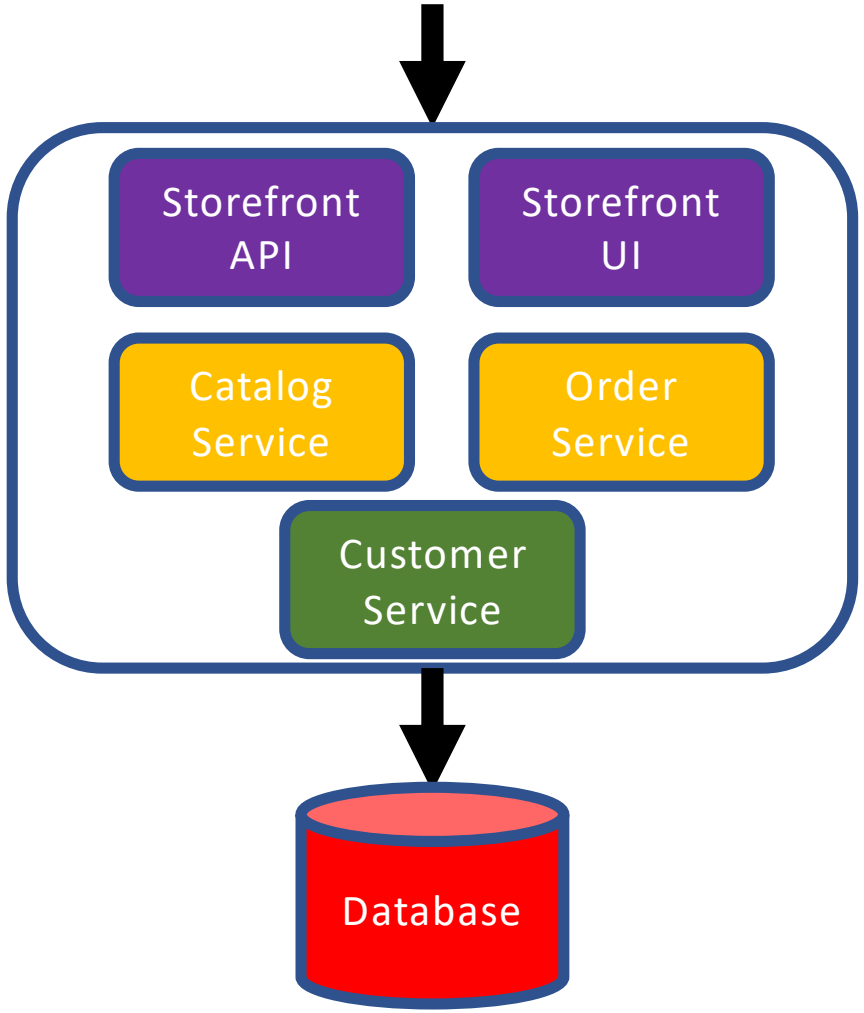
# Monolithic Architecture

- Classic “three-tier” architecture
- Deploy all three tiers as single entity
  - Java WAR/EAR file



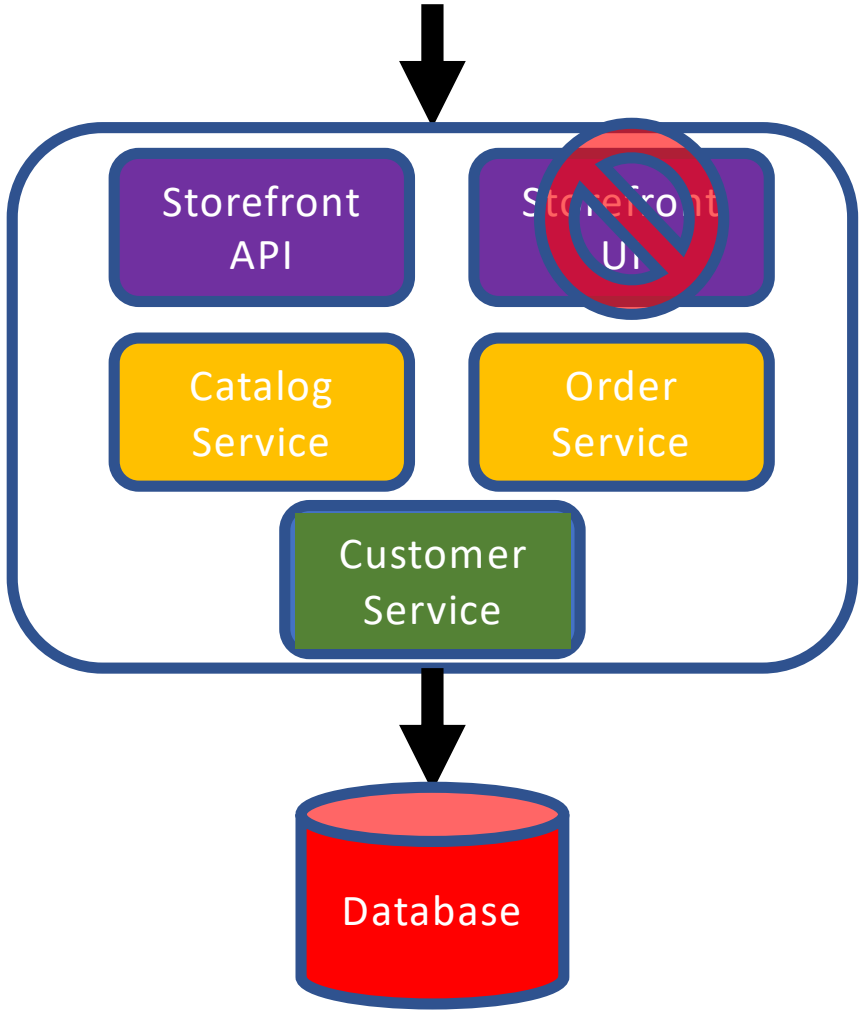
# Monolithic Architecture

- All services built into one application

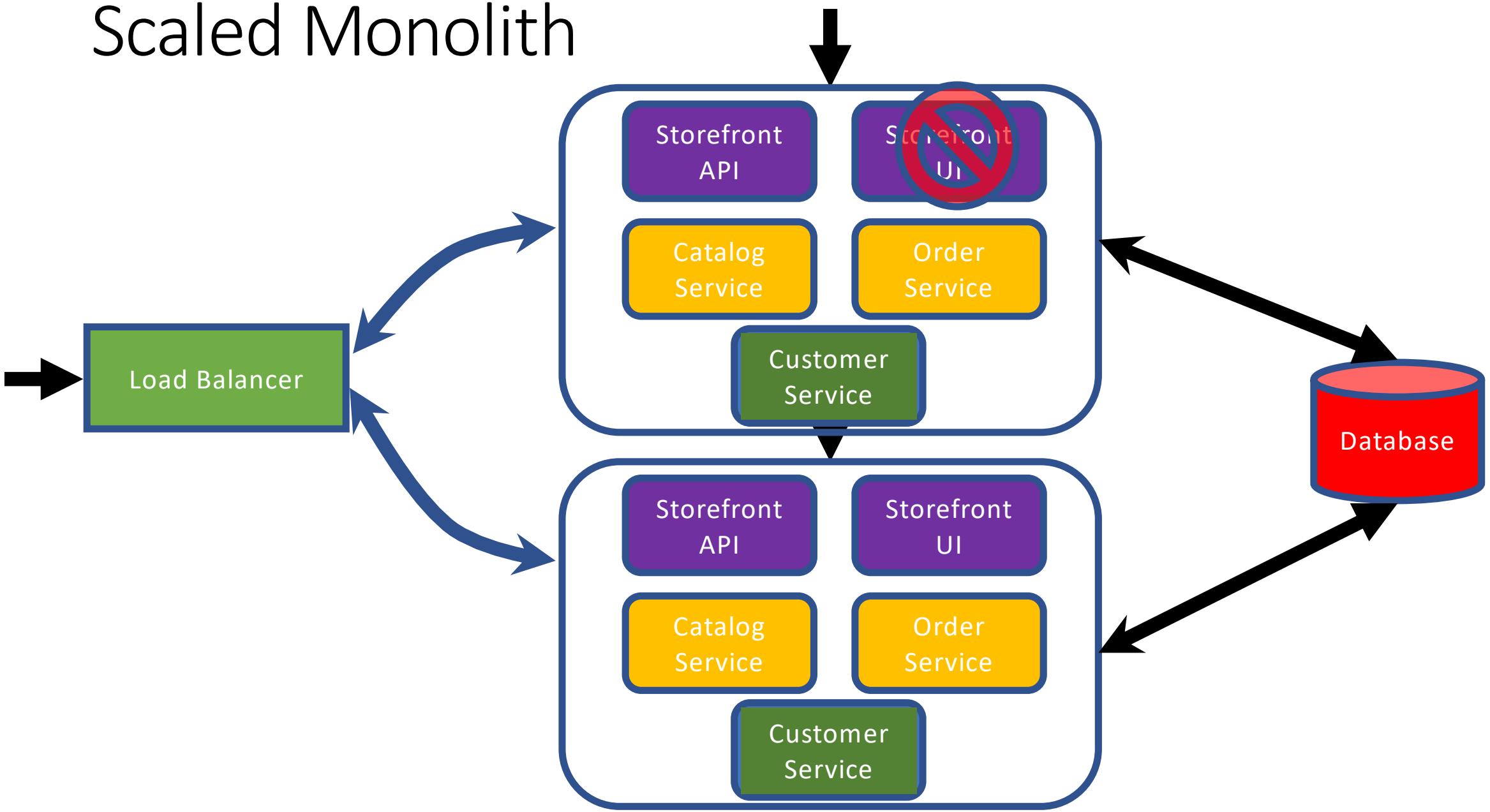


# Monolithic Architecture

- Failure in monolith

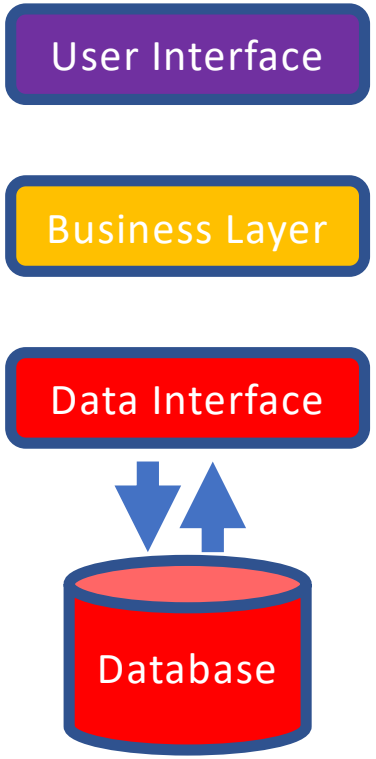


# Scaled Monolith

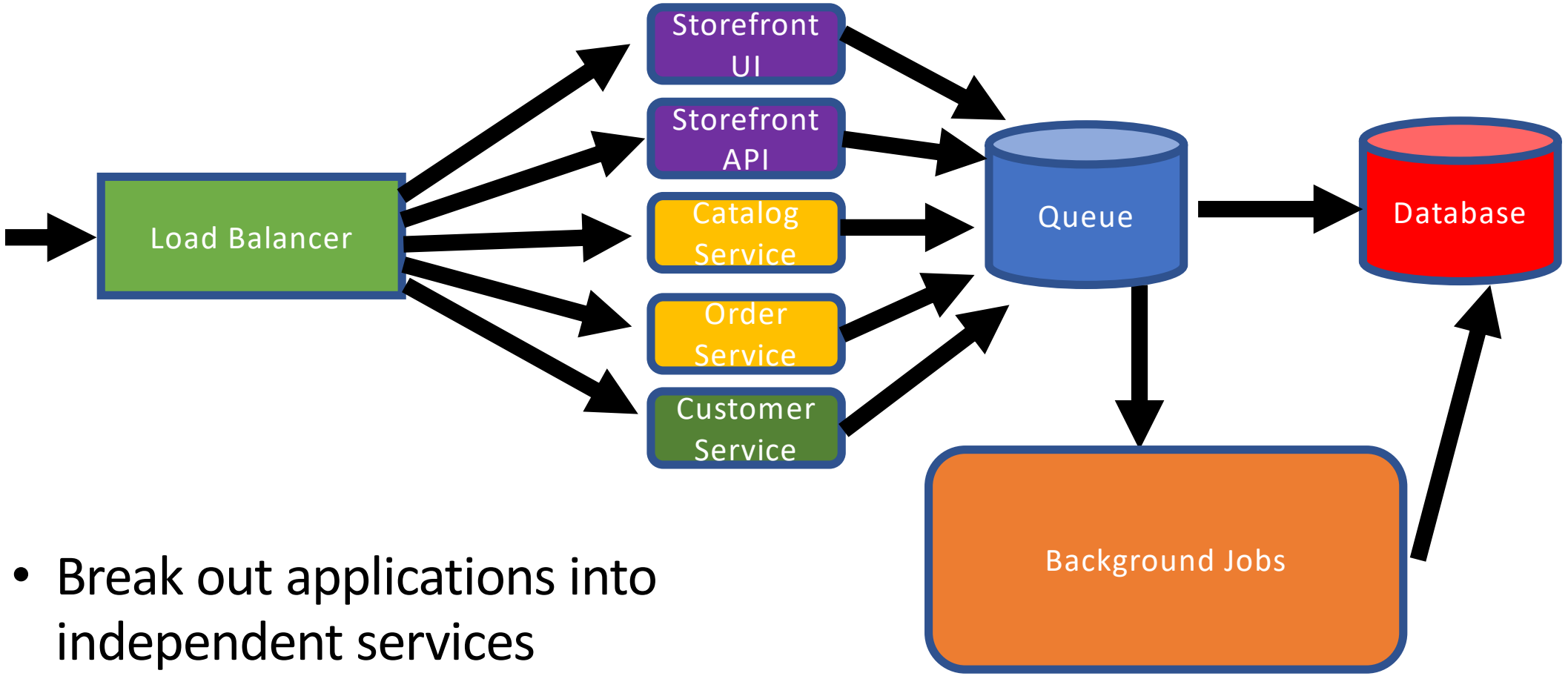


# Monoliths = Siloed Functional Teams

- Components are not aligned to business functionality, silos are created.
- Example: new feature requires three cross functional teams
  - UI, Middleware, DBAs
- Communication breakdown
- Anti-DevOps



# Microservices Architecture



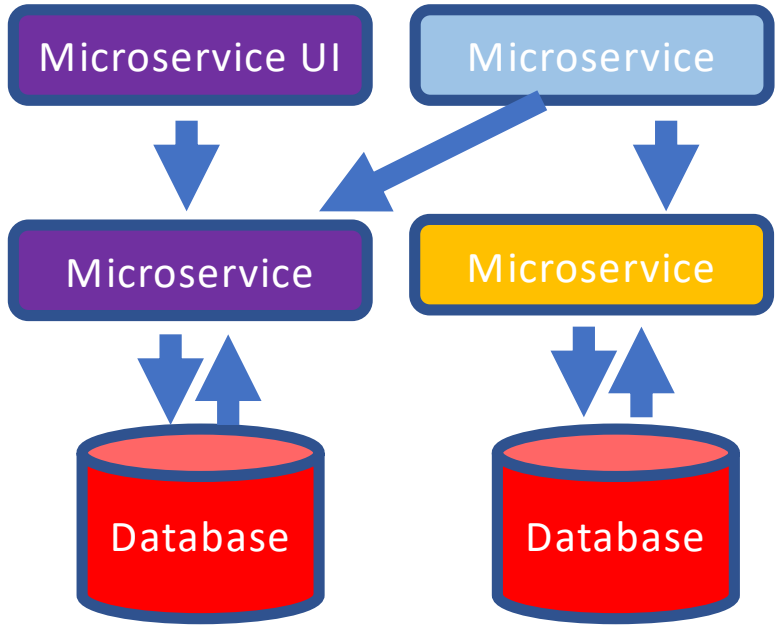

- Break out applications into independent services

# Microservices Application

- Components developed (as services)

AND deployed separately

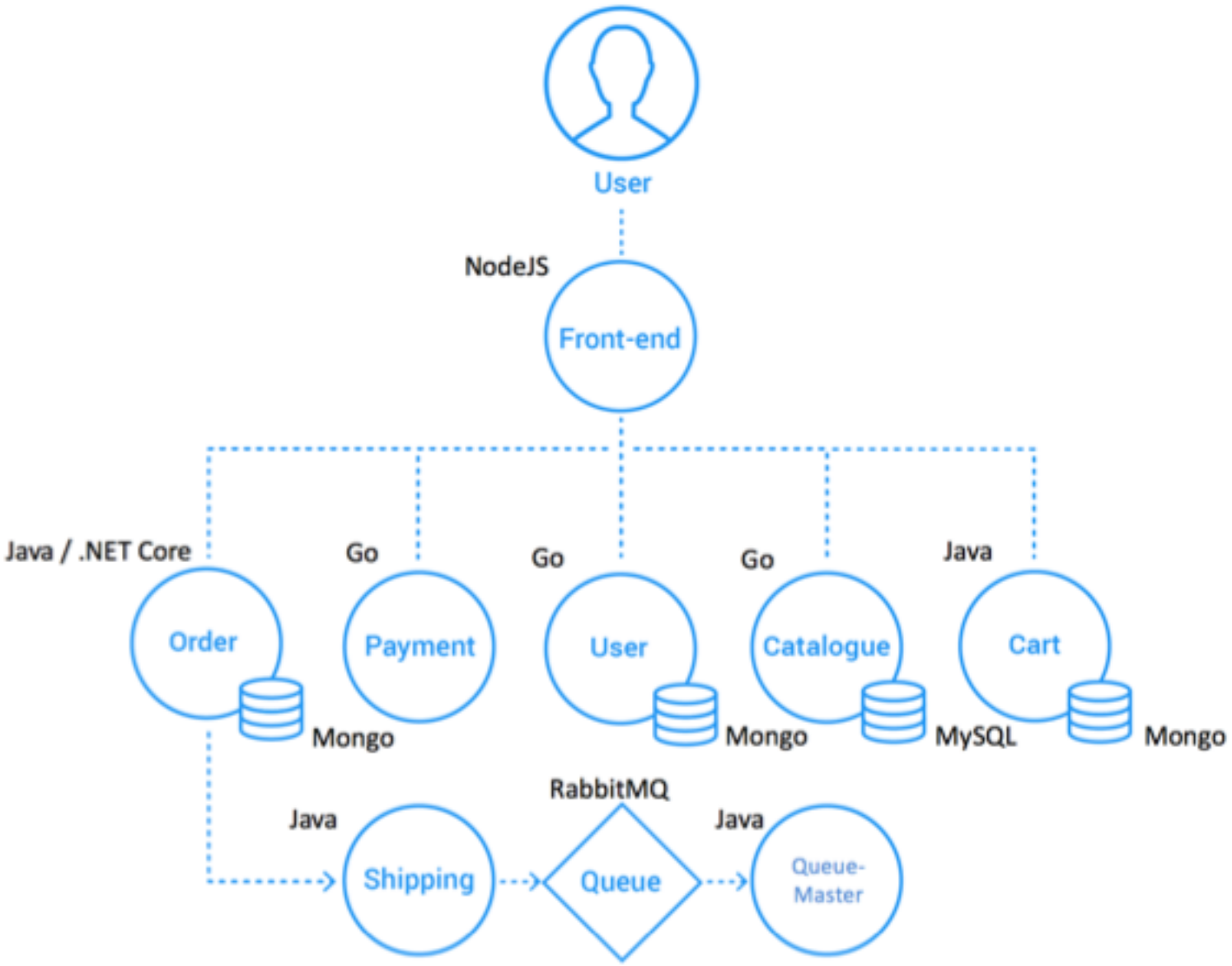
- Self-contained
- Adding new code much easier
- Less chance for inadvertent bugs



# Demo Microservices Applications

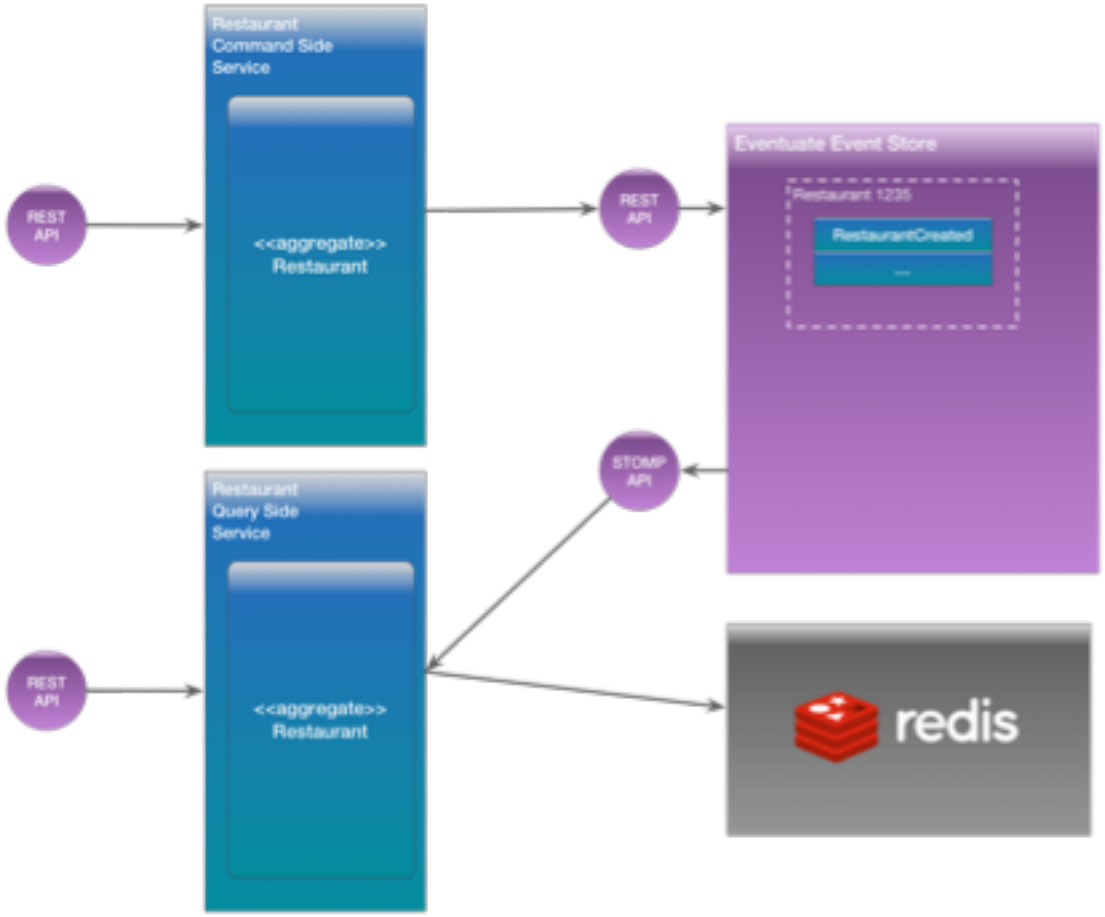
# Sock Shop

- Go
- Spring Boot
- Node.js
- RabbitMQ
- MySQL
- Mongo



# Restaurant Manager

- Spring Boot
- MongoDB
- SwaggerUI
- Gradle
- Zookeeper
- Kafka
- MySQL
- Redis



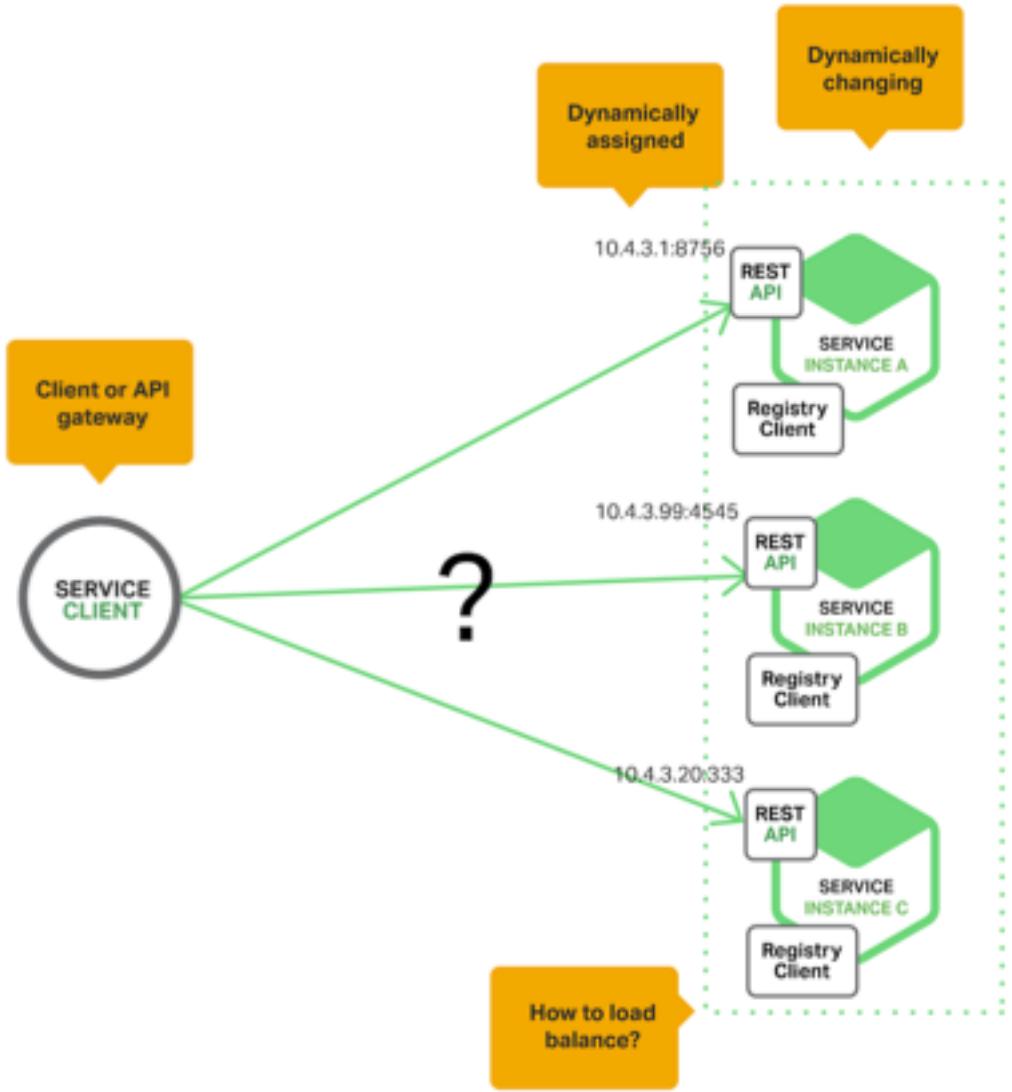
# Microservices Challenges

# Microservices Challenges

- Service discovery & request routing
- Security and access control
- Efficiently scale up/down
  - connectivity
  - service resiliency
- Logging/Monitoring
- Canary/AB testing/rollouts/rollbacks

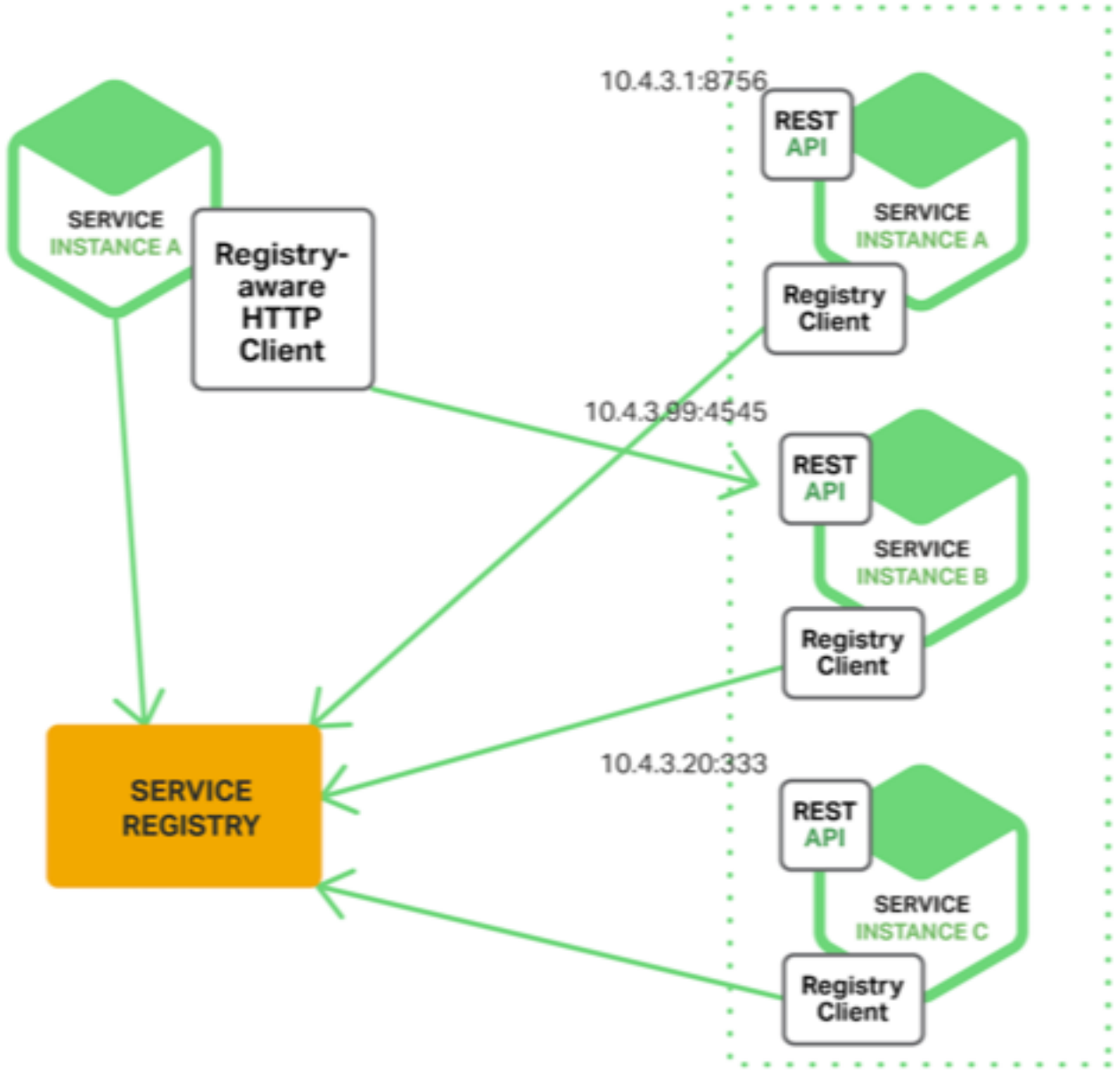
# Service discovery

- Service discovery
  - Client side?
  - Server side?



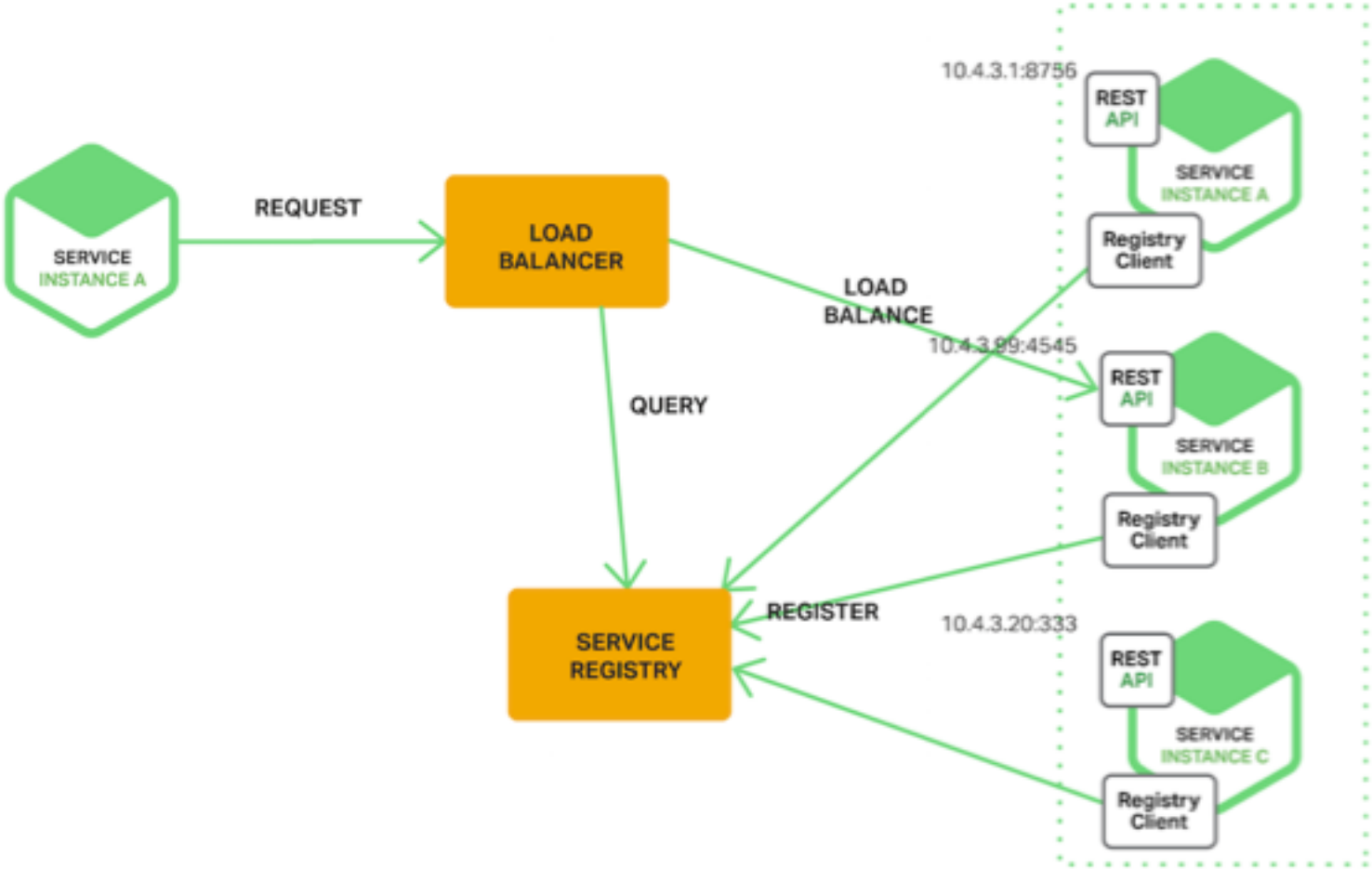
# Service discovery

- Client side
  - Client responsible for finding services
  - Couples client and services requiring code changes



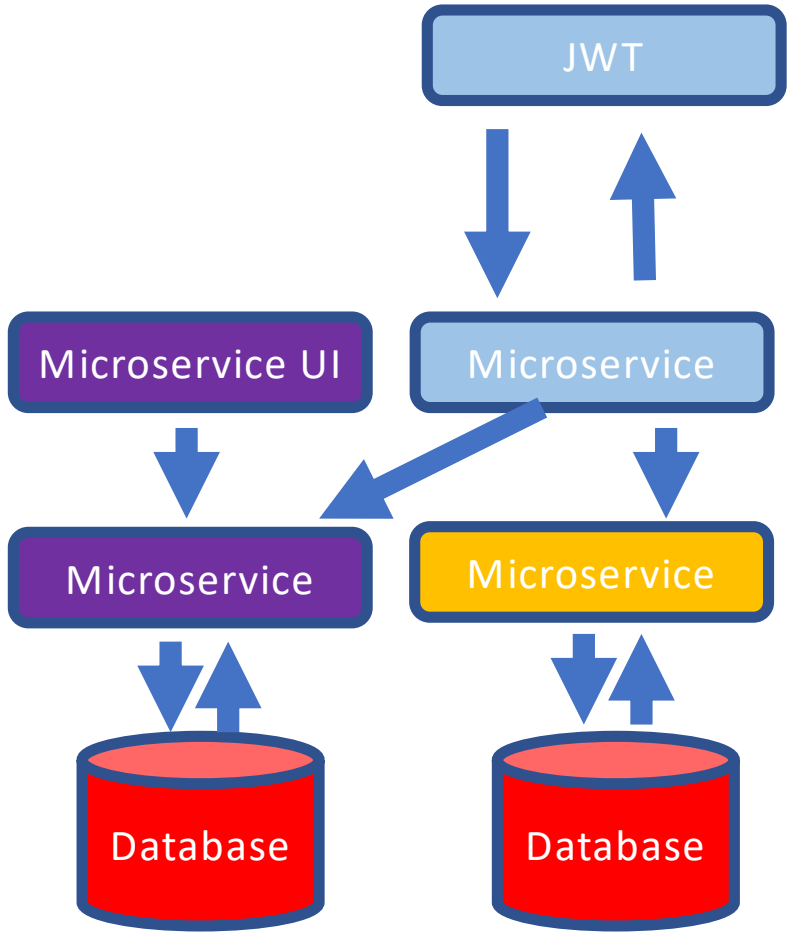
# Service discovery

- Server side
  - ELB
  - Nginx & Consul
  - included in Kubernetes
  - Discovery abstractected away from client

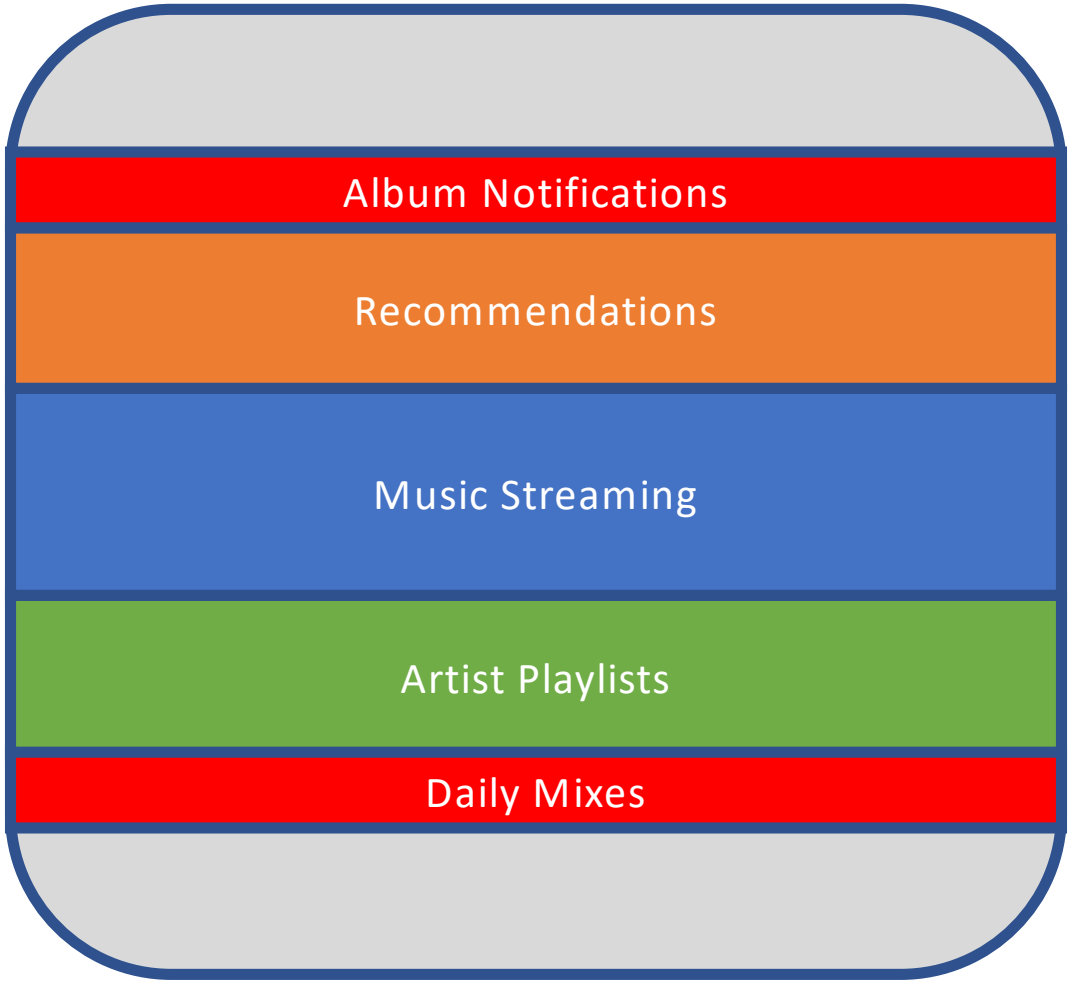


# Security & Access Control

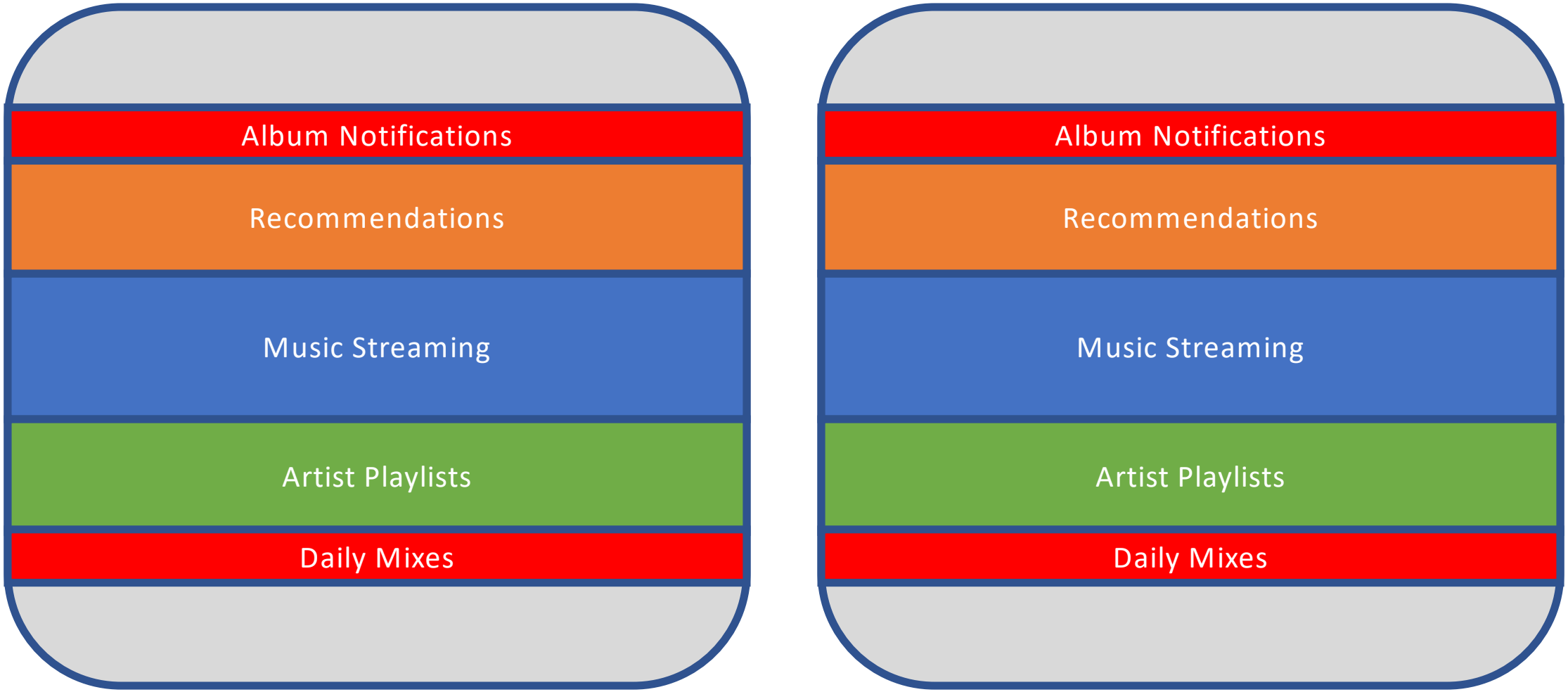
- Stateless
- JSON Web Token (JWT)
- API Gateway usually handles auth



# Scaling - Monolithic

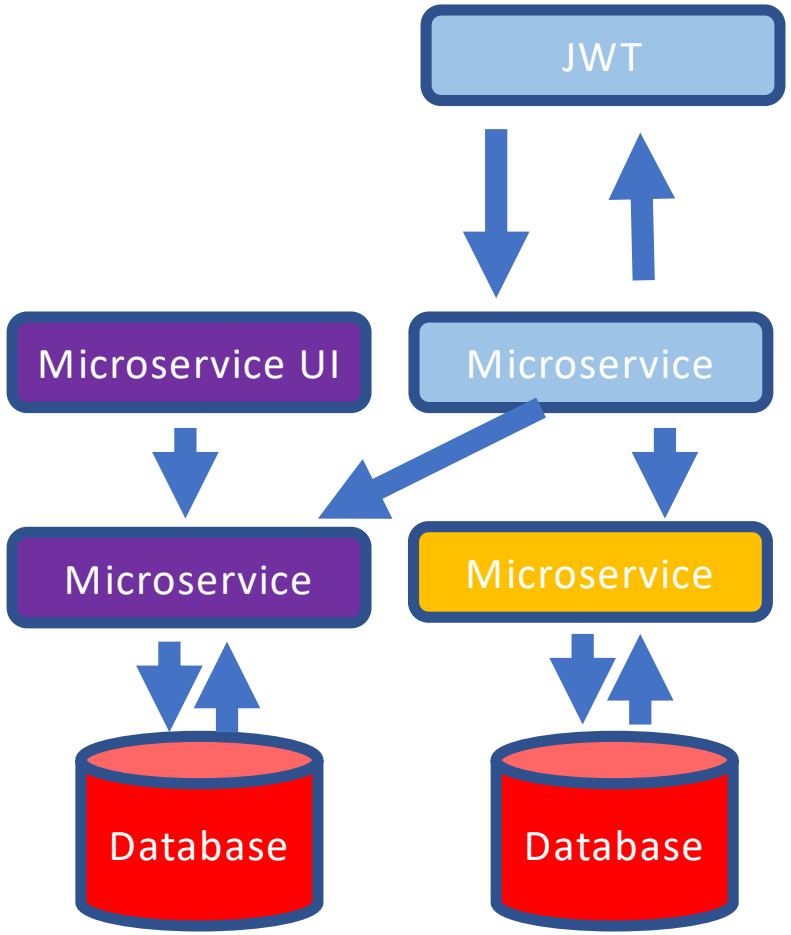


# Scaling - Monolithic



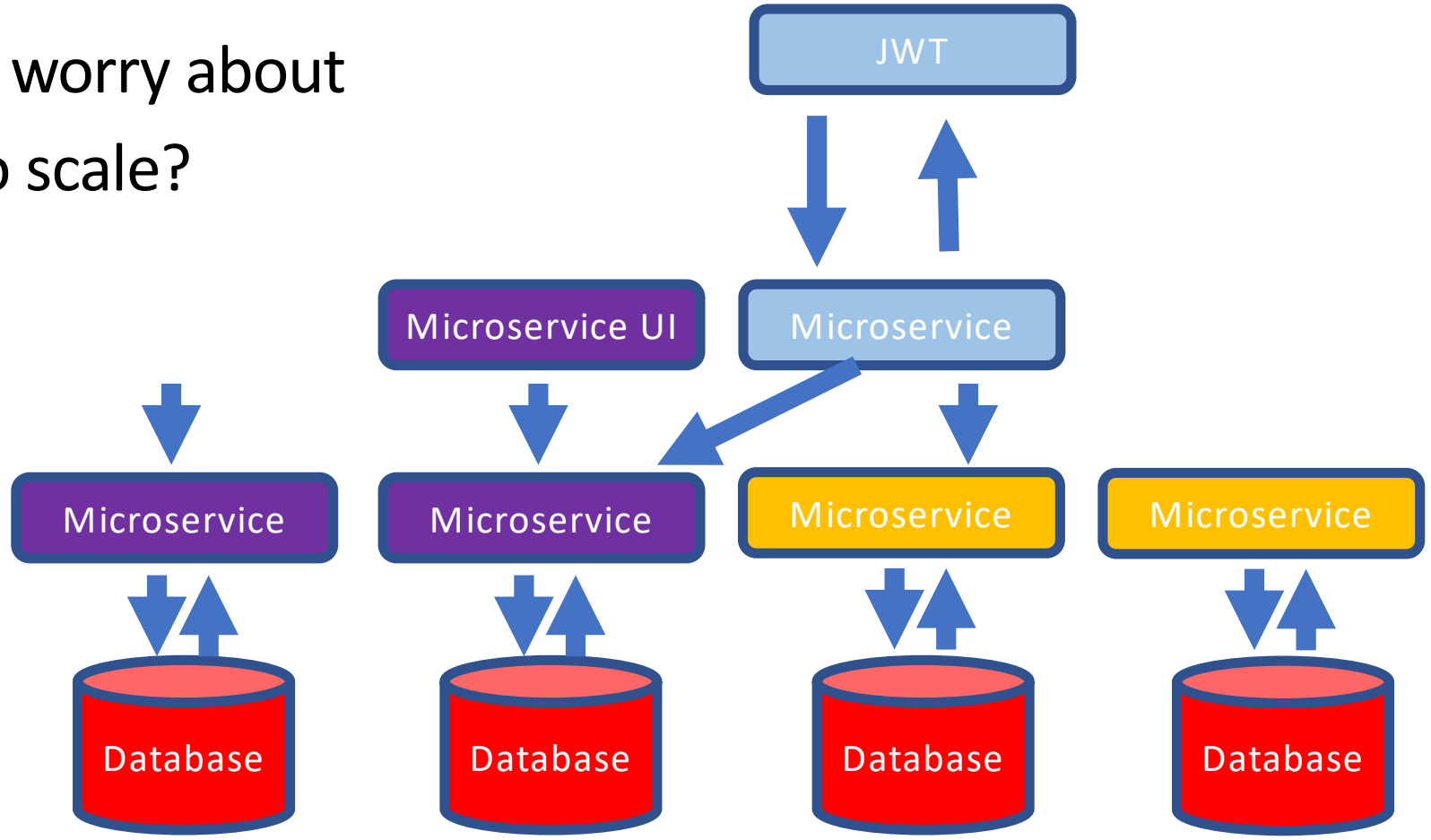
# Scaling - Microservices

- Multiple components to worry about
- Which service(s) need to scale?



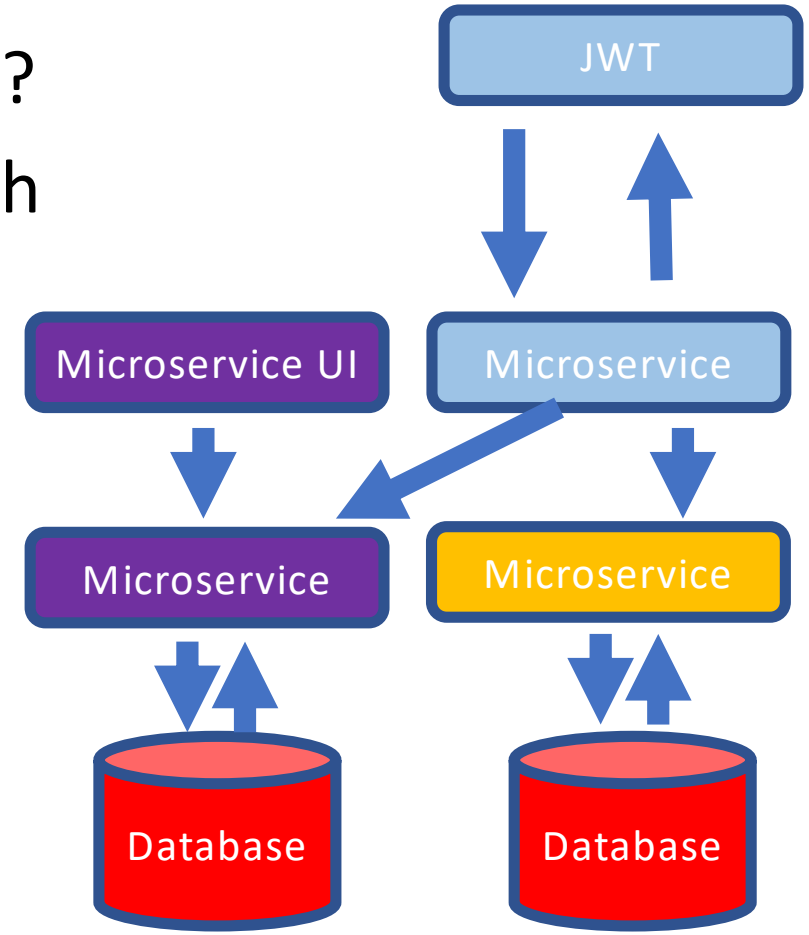
# Scaling - Microservices

- Multiple components to worry about
- Which service(s) need to scale?



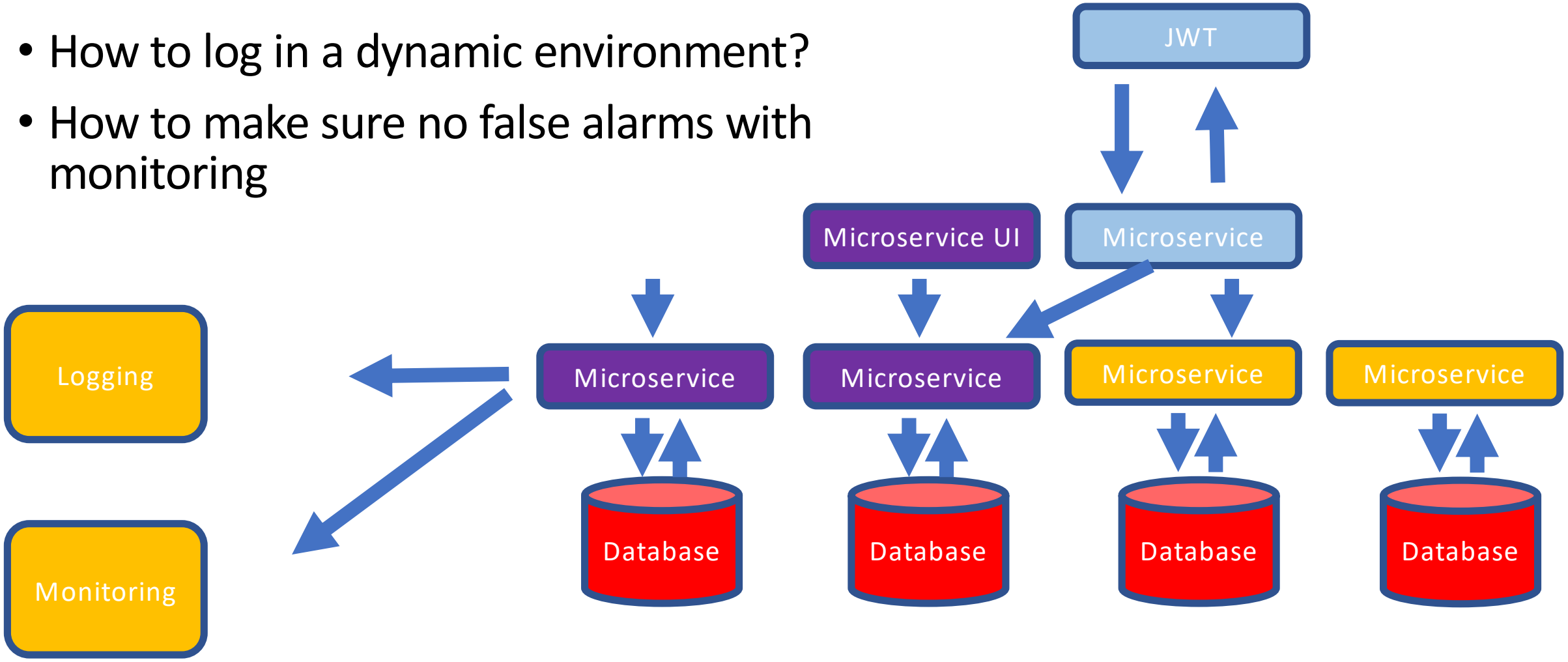
# Logging and Monitoring

- How to log in a dynamic environment?
- How to make sure no false alarms with monitoring



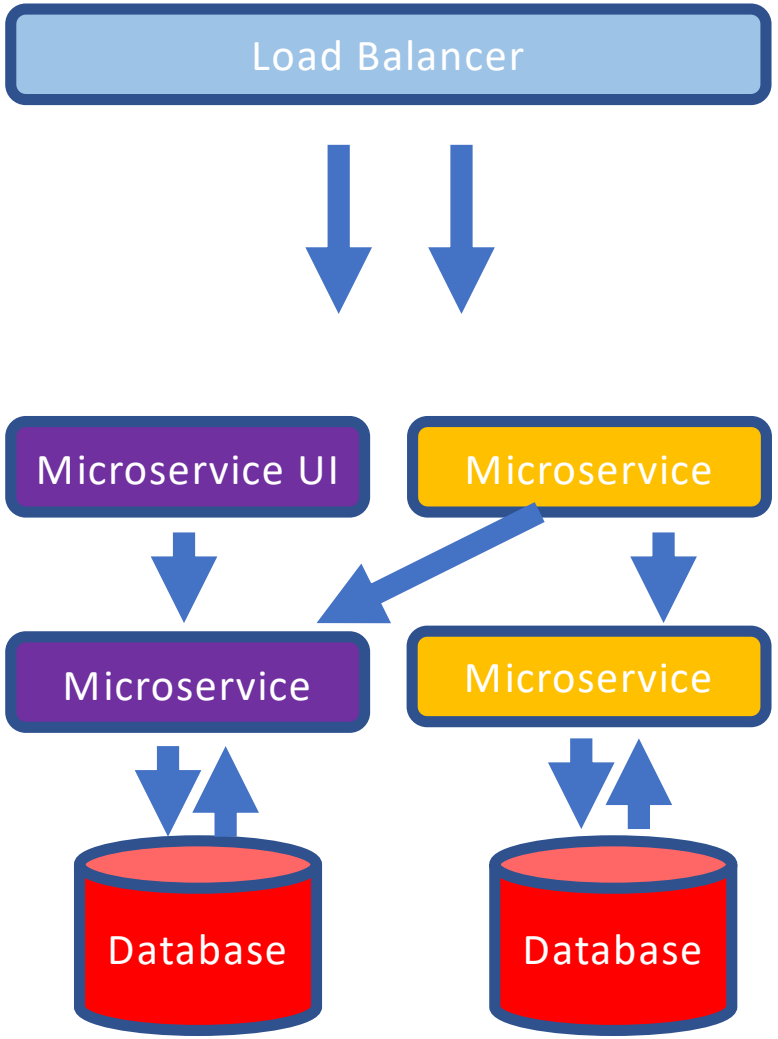
# Logging and Monitoring

- How to log in a dynamic environment?
- How to make sure no false alarms with monitoring



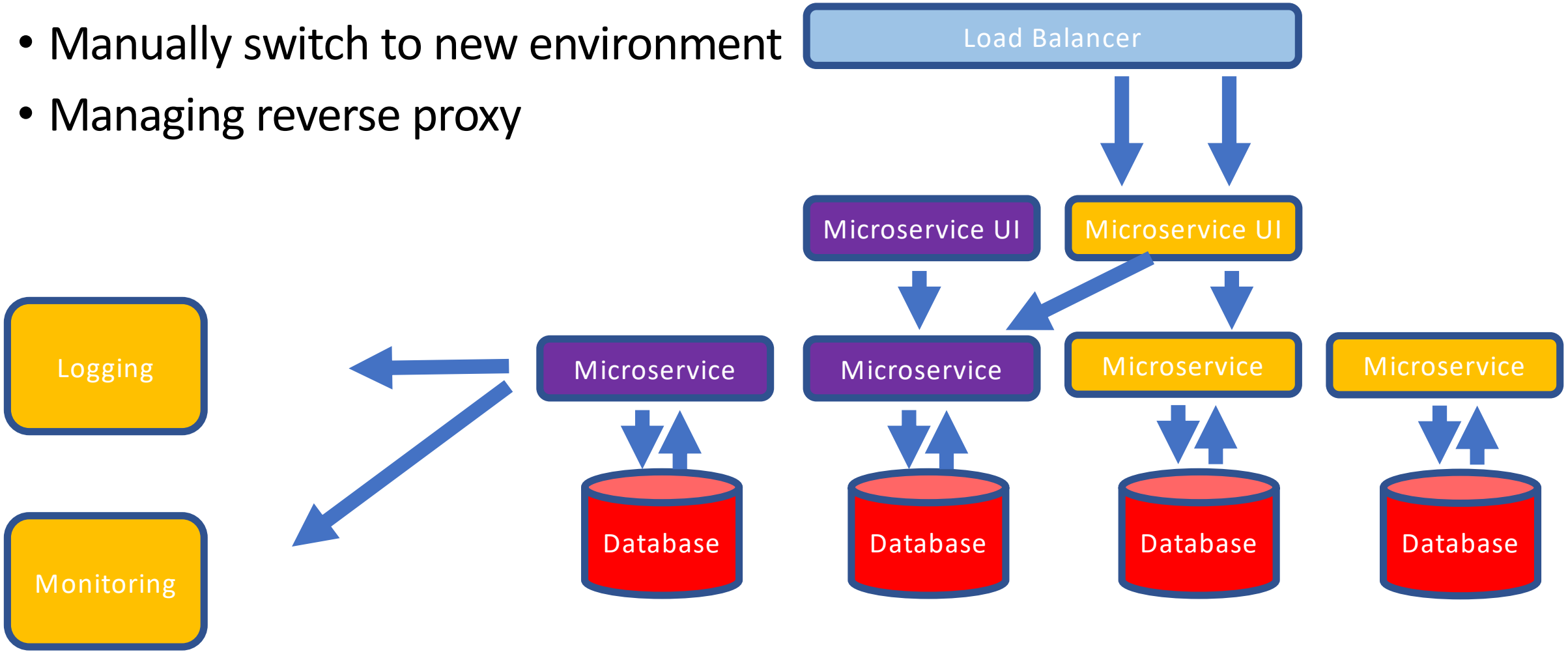
# Canary and A/B Testing

- Manually switch to new environment
- Managing reverse proxy



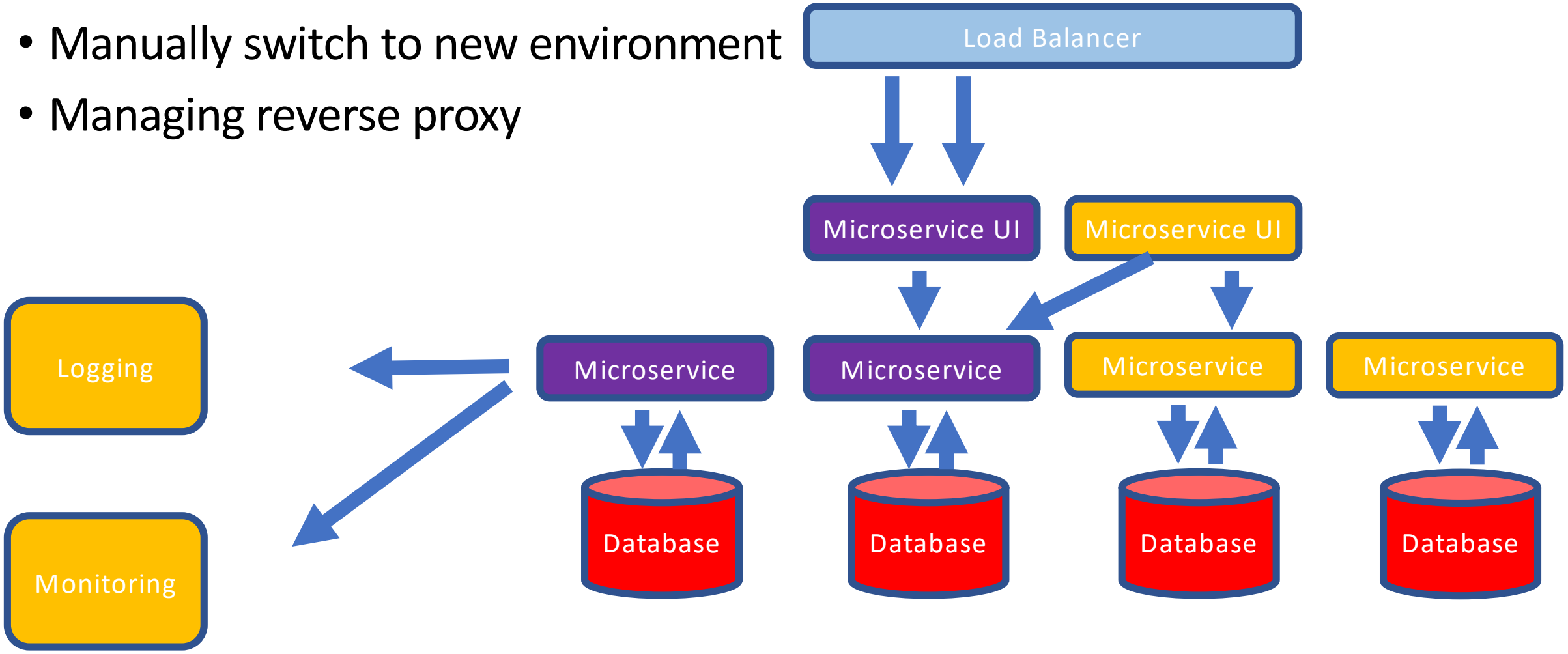
# Canary and A/B Testing

- Manually switch to new environment
- Managing reverse proxy



# Canary and A/B Testing

- Manually switch to new environment
- Managing reverse proxy



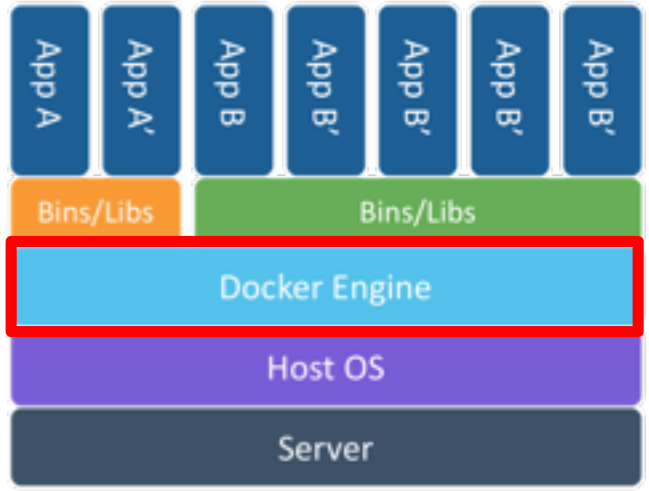
# The Docker Platform Components

# Docker Engine

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

*Lightweight runtime program to **build, ship, and run Docker containers***

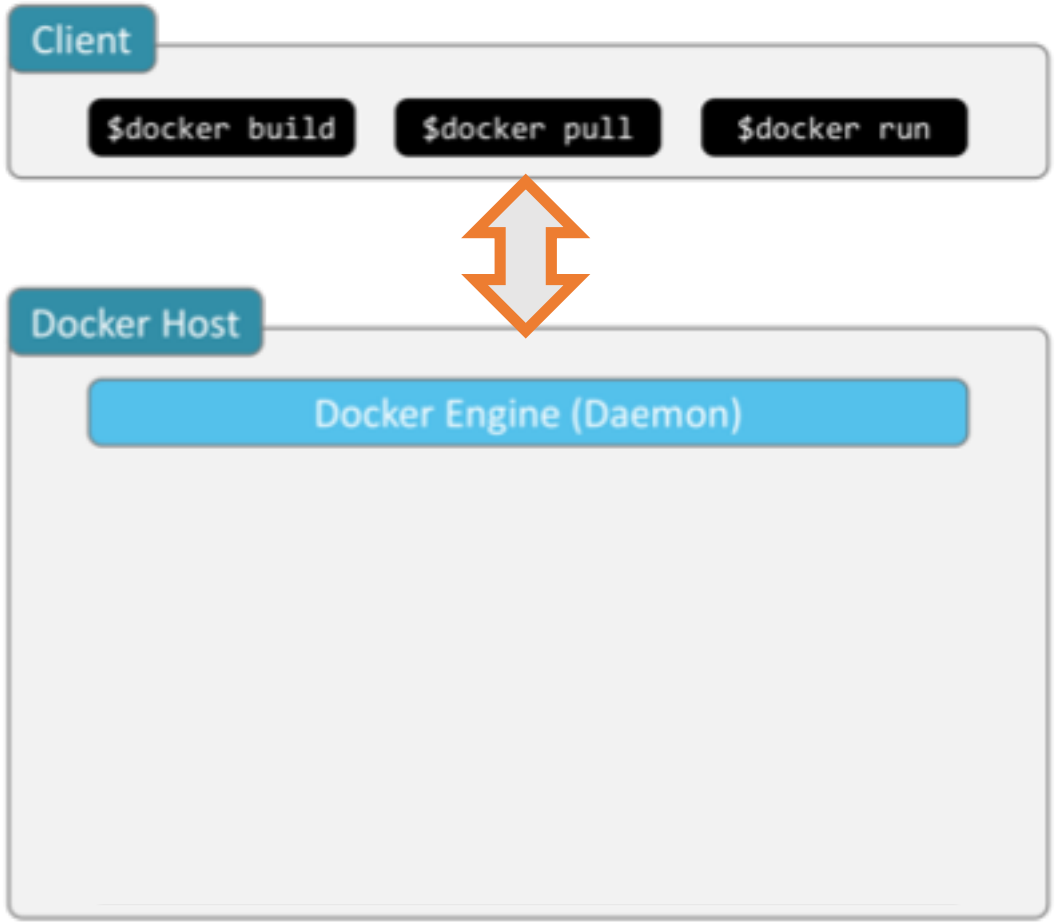
- Also known as **Docker Daemon**
- Uses Linux Kernel namespaces and control groups
  - **Linux Kernel (>= 3.10)**
- Namespaces provide an isolated workspace



# Docker Engine

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

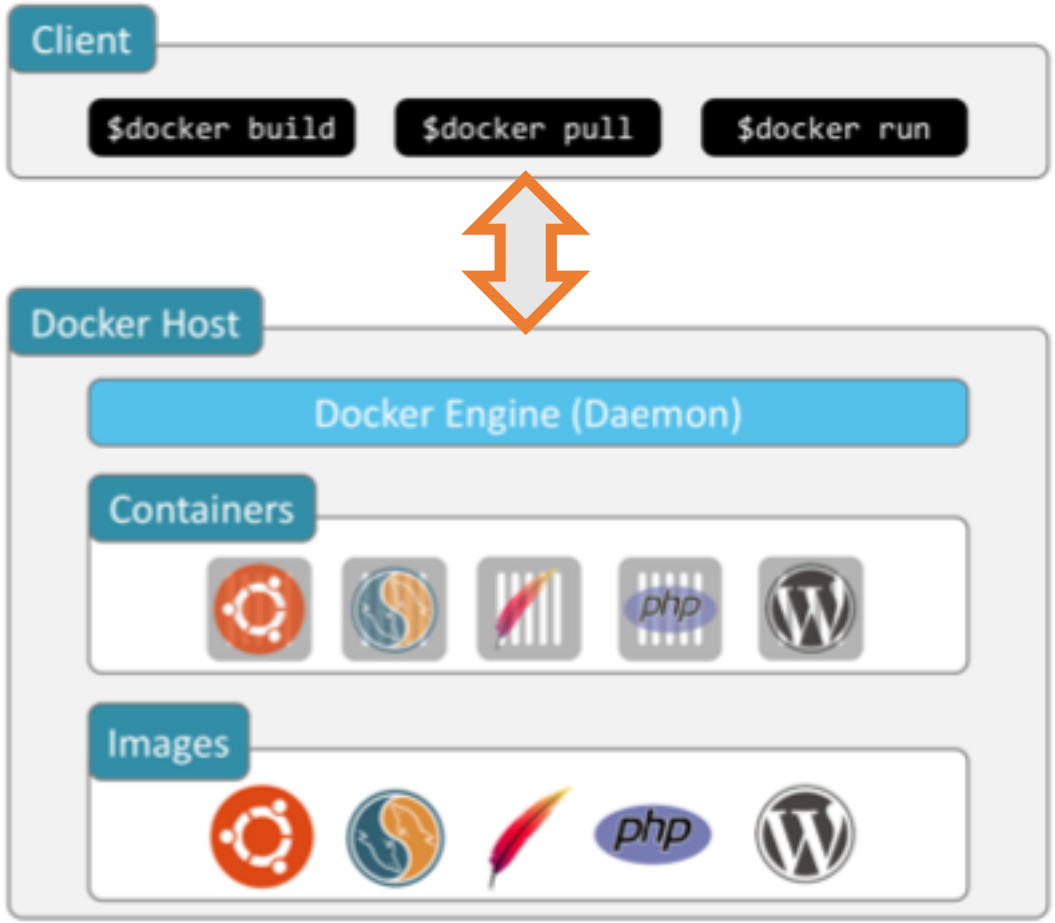
- The Docker Client is the `docker` binary
  - Primary interface to the Docker Host
- Accepts commands and communicates with the Docker Engine (Daemon)



# Docker Engine

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

- Lives on a Docker host
- Creates and manages containers on the host



# Docker Registry

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

*Image Storage & Retrieval System*

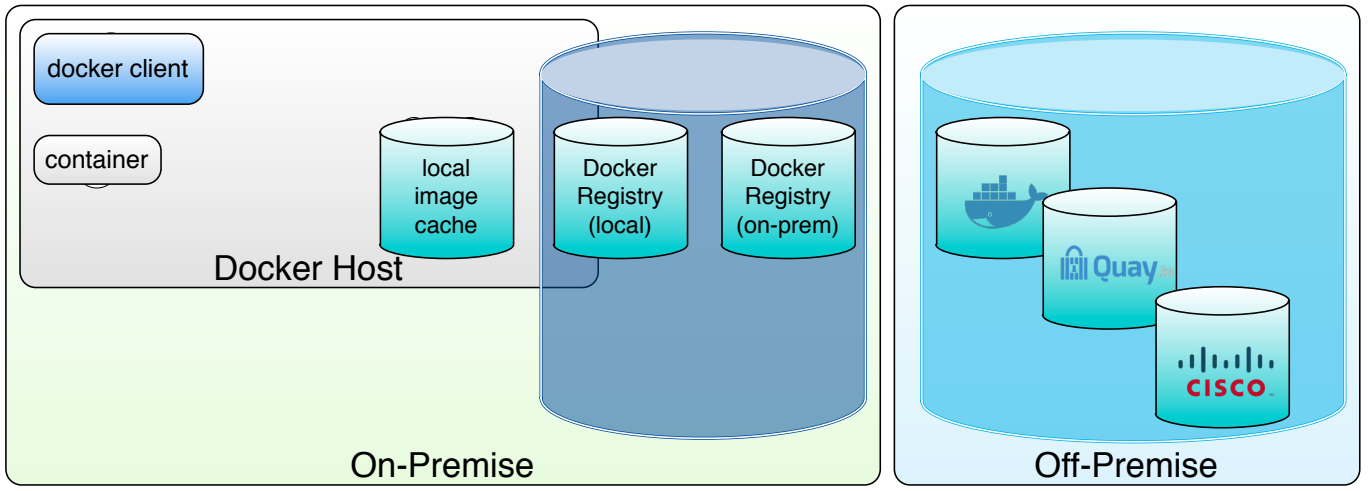
- Docker Engine **Pushes** Images to a Registry
- Version Control
- Docker Engine **Pulls** Images to Run




# Docker Registry

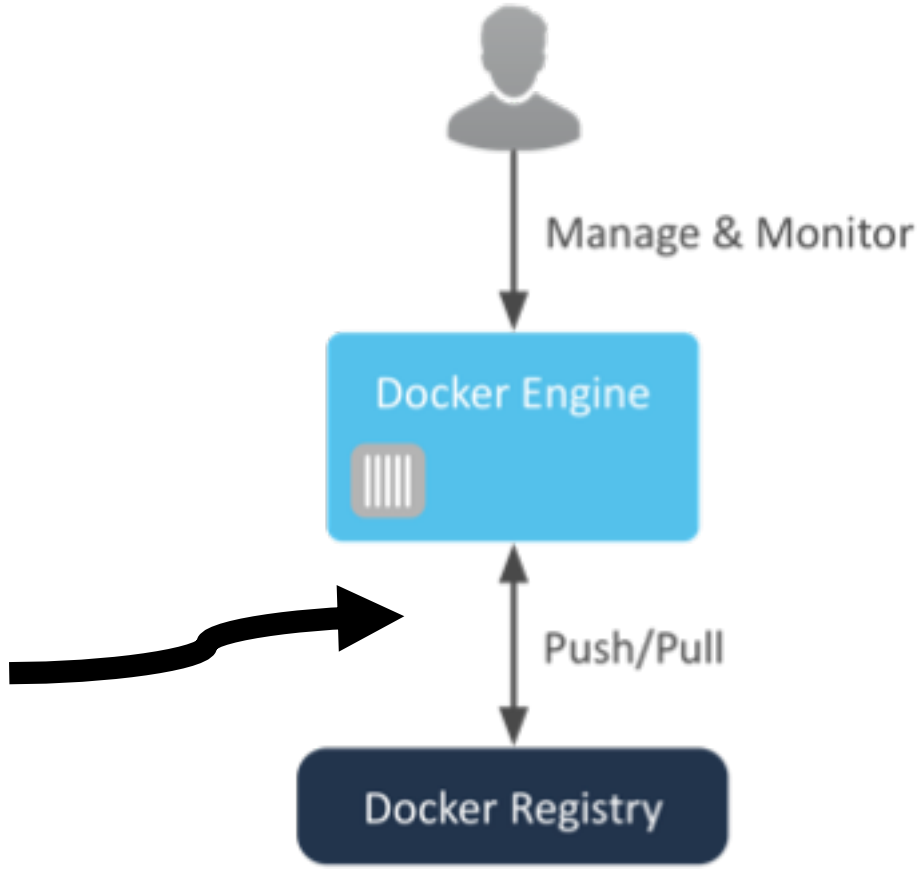
Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

- **Types of Docker Registries**
- Local Docker Registry (On Docker Host)
- Remote Docker Registry (On-Premise/Off-Premise)
- Docker Hub (Off-Premise)



# Docker Engine and Registry

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm




## Docker Engine:

- **Pushes** Images to a Registry
- **Pulls** Images to Run

# Docker Registry

Docker Engine | *Docker Registry* | Docker Compose | Docker Swarm

**The registry and engine both present APIs**

- All of Docker's functionality will utilize these APIs

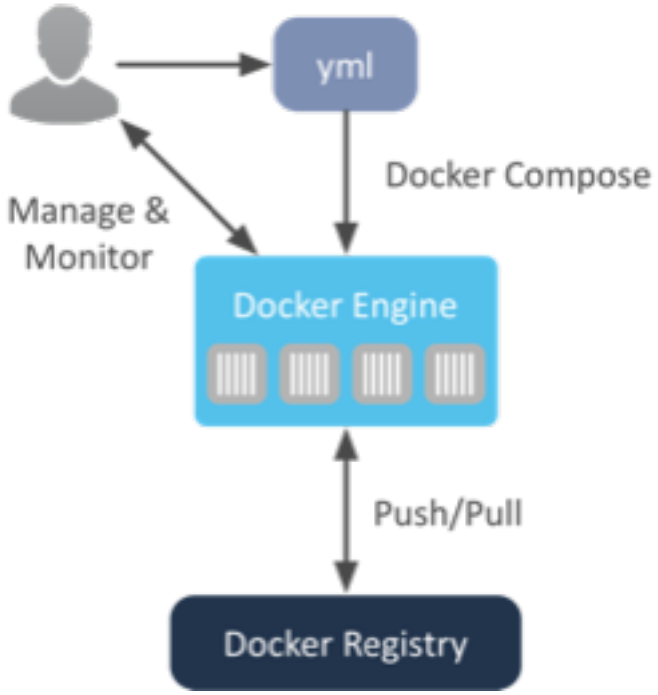
- RESTFUL API
- Commands presented with Docker's CLI tools can also be used with curl and other tools

# Docker Compose

Docker Engine | *Docker Registry* | **Docker Compose** | Docker Swarm

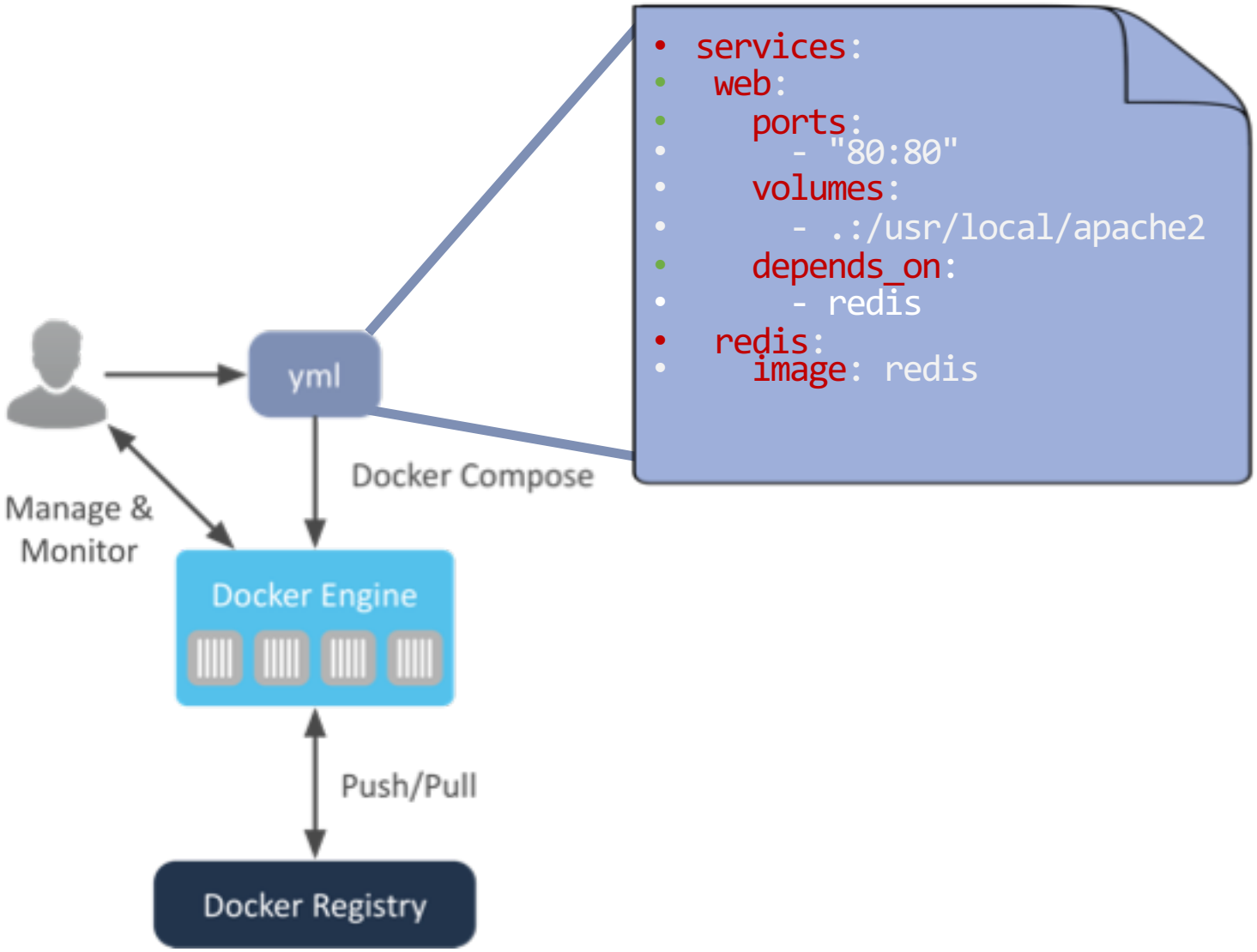
*Tool to create and manage multi-container applications*

- Applications defined in a single file:
- **docker-compose.yml**
- Transforms applications into individual containers that are linked together
- Compose will start all containers in a single command



# Docker Compose

Docker Engine | *Docker Registry* | **Docker Compose** | Docker Swarm

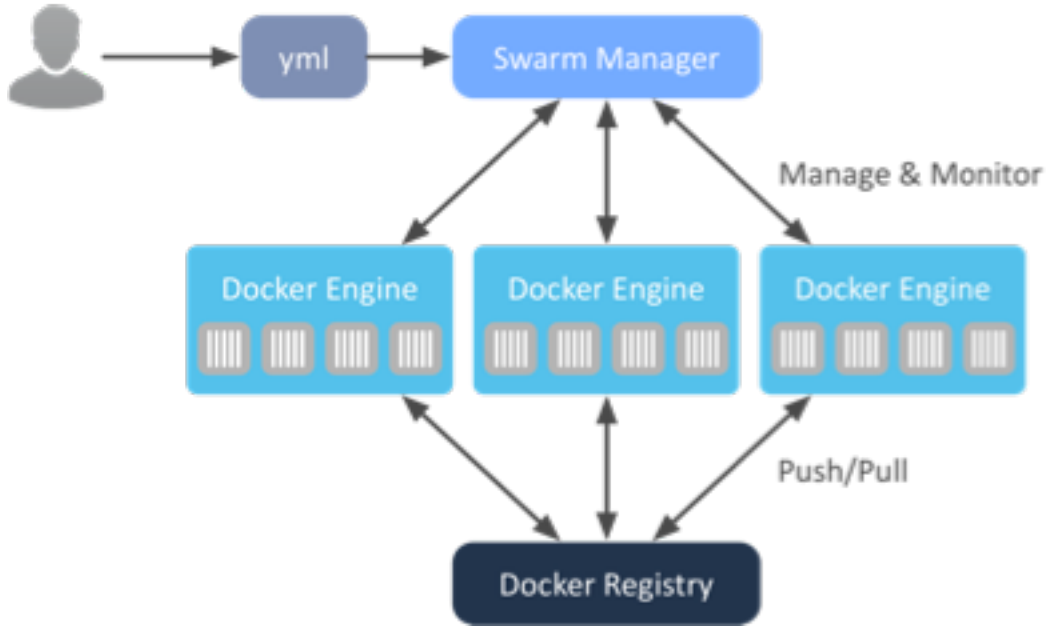


# Docker Swarm

Docker Engine | *Docker Registry* | Docker Compose | **Docker Swarm**

*Clusters Docker hosts and schedules containers*

- Native Clustering for Docker
  - Turn a pool of Docker hosts into a single, virtual host
  - Serves the standard Docker API



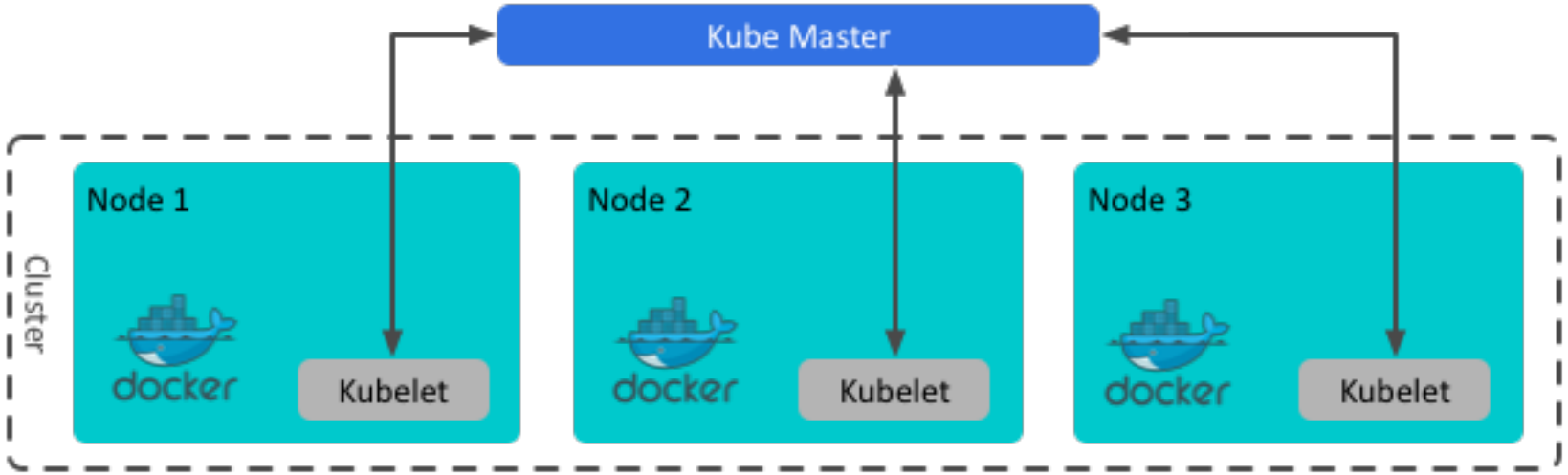
# Labs:

1. Manually build Docker images
2. Package Microservices in containers

# Overview of Kubernetes Clusters

# Kubernetes Clusters

- Kubernetes deployed on a set of physical or virtual hosts – K8s nodes
- Hosts run host OS that supports Linux containers, e.g. Docker or rkt hosts
- Kubernetes runs well in both private and public IaaS environments
- Users and admins control Kubernetes resources via REST API on K8s master



# Kubernetes Components: Master

**Main Components:**

**Master Node**:

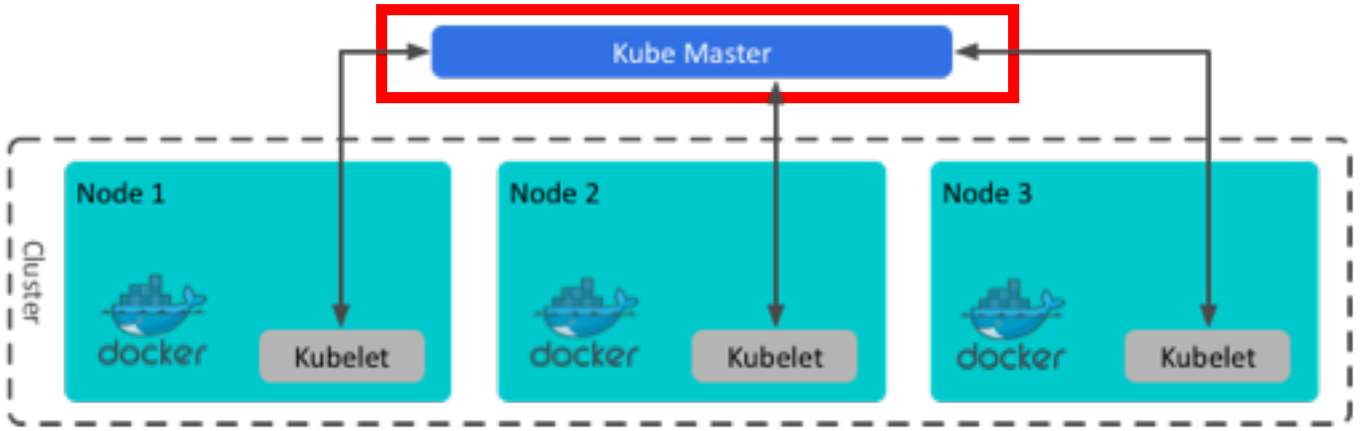
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

**Worker Nodes**:

- Run user workloads as directed by the K8s master
- The master may also serve as a worker node

**Clusters**:

- A collection of nodes bound to a Master and managed as a single logical unit of capacity



# Kubernetes Components: Nodes

**Main Components:**

**Master Node:**

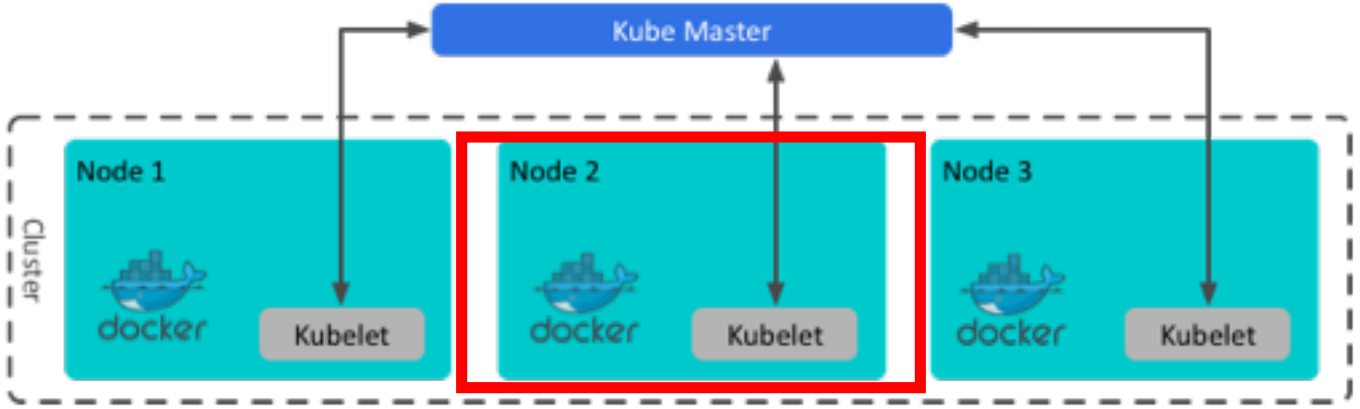
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

**Worker Nodes:**

- Run user workloads as directed by the K8s master
- The master may also serve as a worker node

**Clusters:**

- A collection of nodes bound to a Master and managed as a single logical unit of capacity



# Kubernetes Cluster Components

**Main Components:**

**Master Node**:

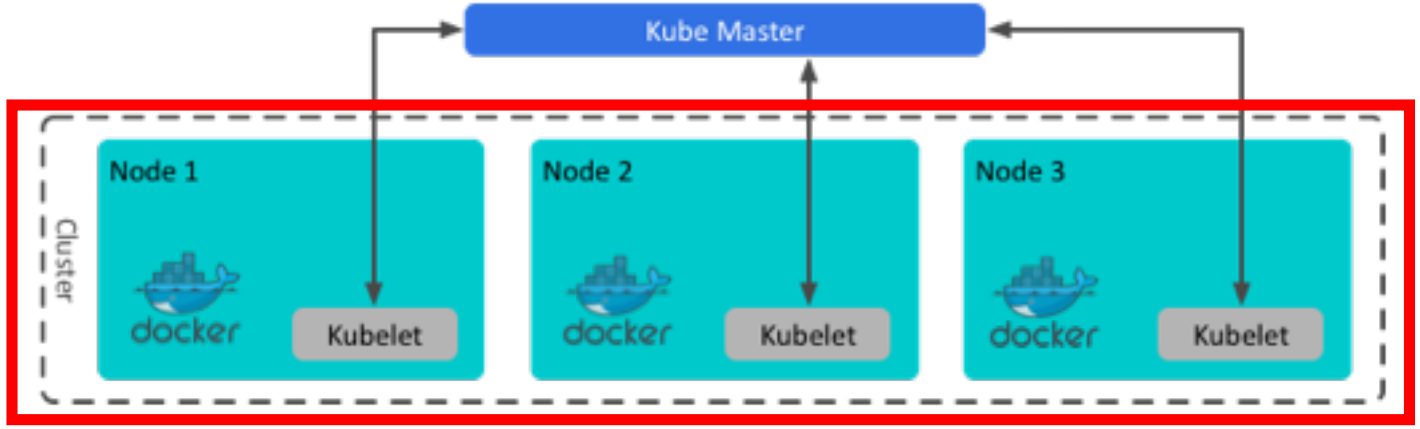
- This node manages and oversees other nodes and hosts the K8s API, scheduler, and controllers

**Worker Nodes**:

- Run user workloads as directed by the K8s master
- The master may also serve as a worker node

**Clusters**:

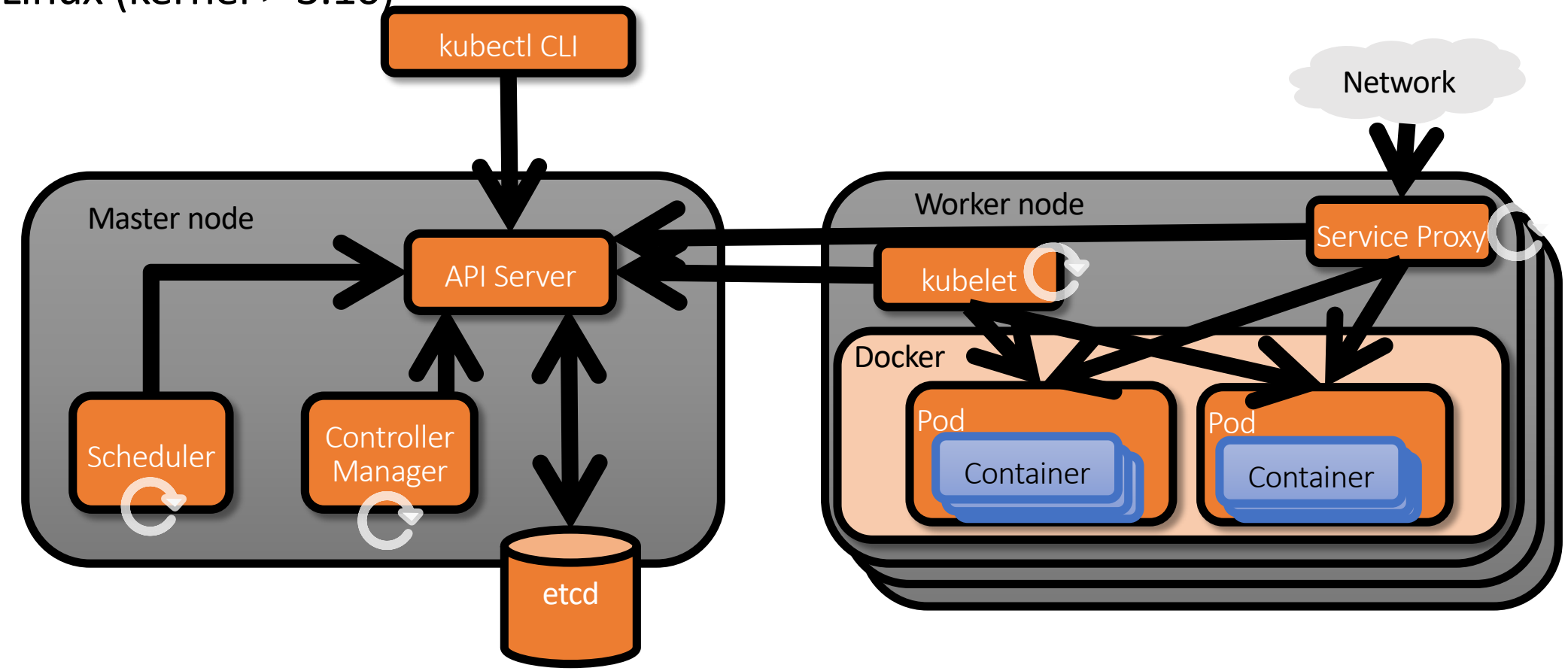
- A collection of nodes bound to a Master and managed as a single logical unit of capacity



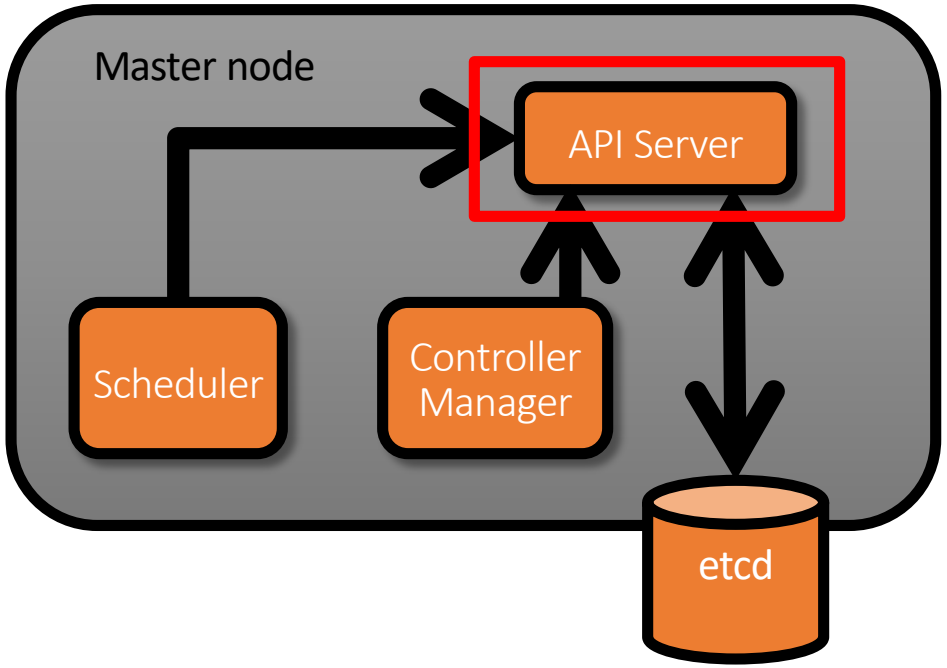
# Kubernetes Architecture

# Kubernetes Cluster Architecture

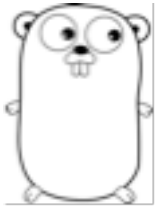
- Kubernetes nodes can be physical hosts or VM's running a container-friendly Linux (kernel > 3.10)



# Kubernetes Master Node Components

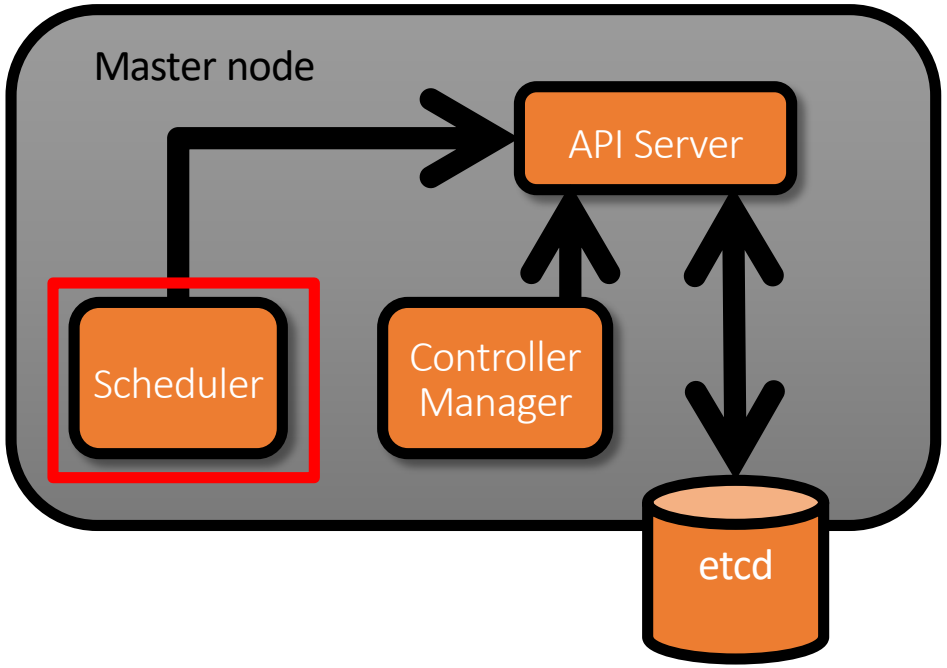



- **API Server (kube-apiserver)**: exposes the Kubernetes REST API, and can be scaled horizontally
- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on
- **Controller manager (kube-controller-manager)**: runs background controller processes for the system to enforce declared object states, e.g. Node Controller, Replication Controller, …
- **Persistent data store (etcd)**: all K8s system data is stored in a distributed, reliable key-value store. etcd may run on separate nodes from the master

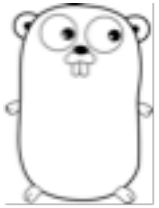


K8s components written in Go (golang.org)

# Kubernetes Master Node Components

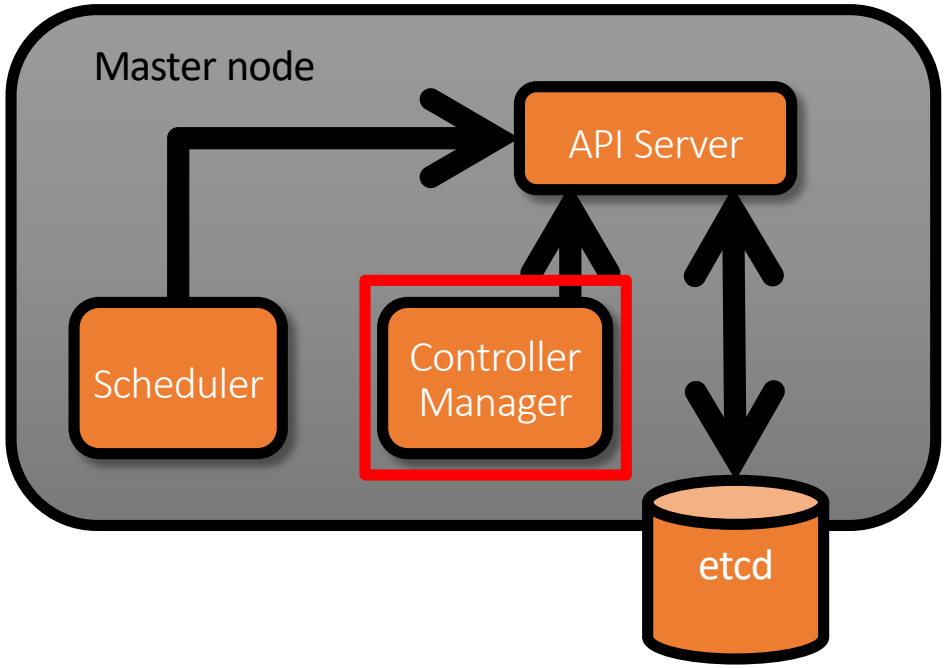


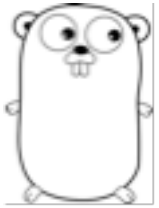

- **API Server (kube-apiserver)**: exposes the Kubernetes REST API, and can be scaled horizontally
- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on
- **Controller manager (kube-controller-manager)**: runs background controller processes for the system to enforce declared object states, e.g. Node Controller, Replication Controller, …
- **Persistent data store (etcd)**: all K8s system data is stored in a distributed, reliable key-value store. etcd may run on separate nodes from the master



K8s components written in Go (golang.org)

# Kubernetes Master Node Components

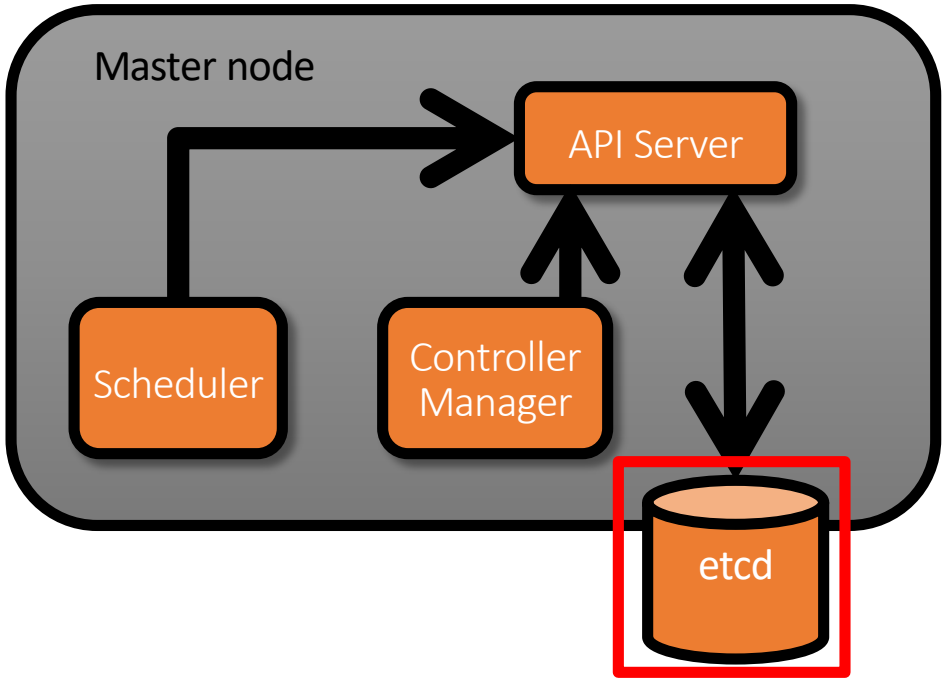


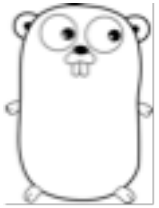
K8s components written in Go (golang.org)

- **API Server (kube-apiserver)**: exposes the Kubernetes REST API, and can be scaled horizontally
- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on
- **Controller manager (kube-controller-manager)**: runs background controller processes for the system to enforce declared object states, e.g. Node Controller, Replication Controller, …
- **Persistent data store (etcd)**: all K8s system data is stored in a distributed, reliable key-value store. etcd may run on separate nodes from the master

# Kubernetes Master Node Components

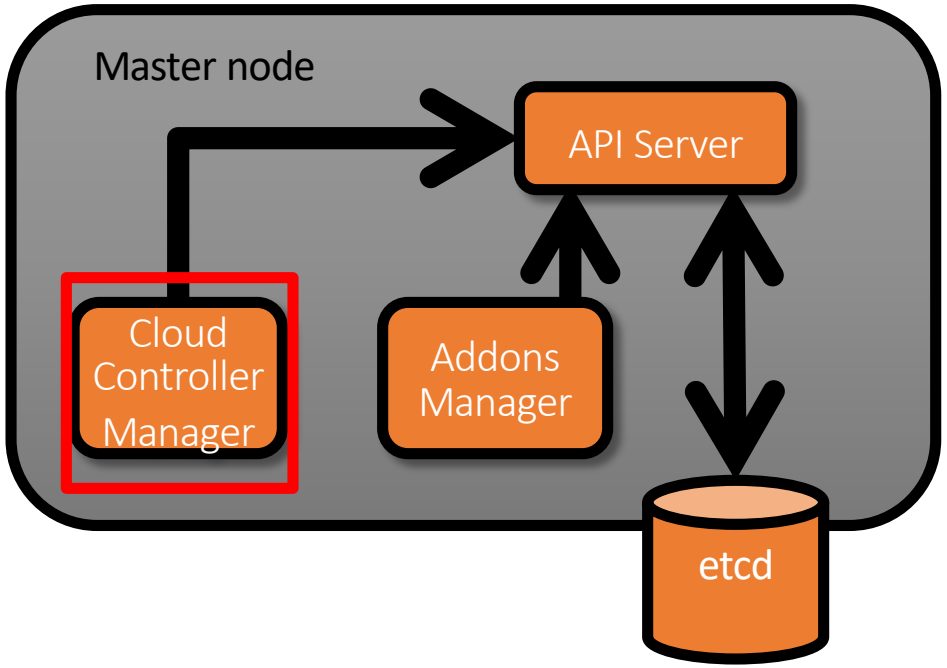


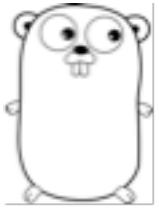

- **API Server (kube-apiserver)**: exposes the Kubernetes REST API, and can be scaled horizontally
- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on
- **Controller manager (kube-controller-manager)**: runs background controller processes for the system to enforce declared object states, e.g. Node Controller, Replication Controller, …
- **Persistent data store (etcd)**: all K8s system data is stored in a distributed, reliable key-value store. etcd may run on separate nodes from the master



K8s components written in Go (golang.org)

# Master Node Additional Components



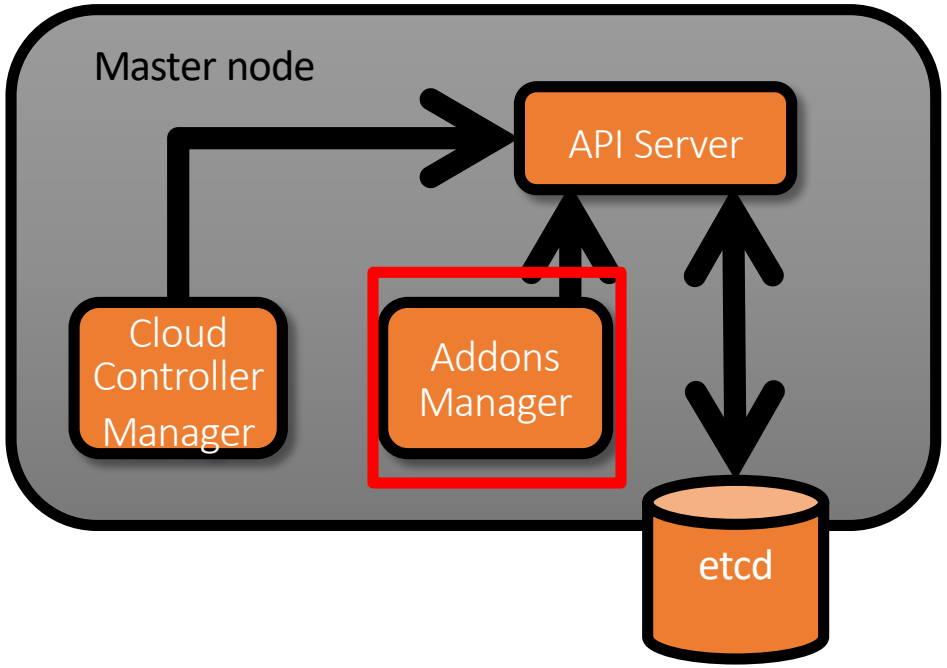
K8s components written in Go (golang.org)

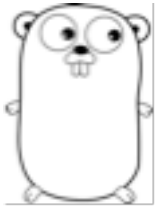
- **cloud-controller-manager**: runs controllers interacting with underlying IaaS providers – alpha feature

  Allows cloud vendor-specific code to be separate from main K8s system components

  - **addons-manager**: creates and maintains cluster addon resources in 'kube-system' namespace, e.g.
  - **Kubernetes Dashboard**: general web UI for application and cluster management
  - **kube-dns**: serves DNS records for K8s services and resources
  - Container resource monitoring and cluster-level logging

# Master Node Additional Components



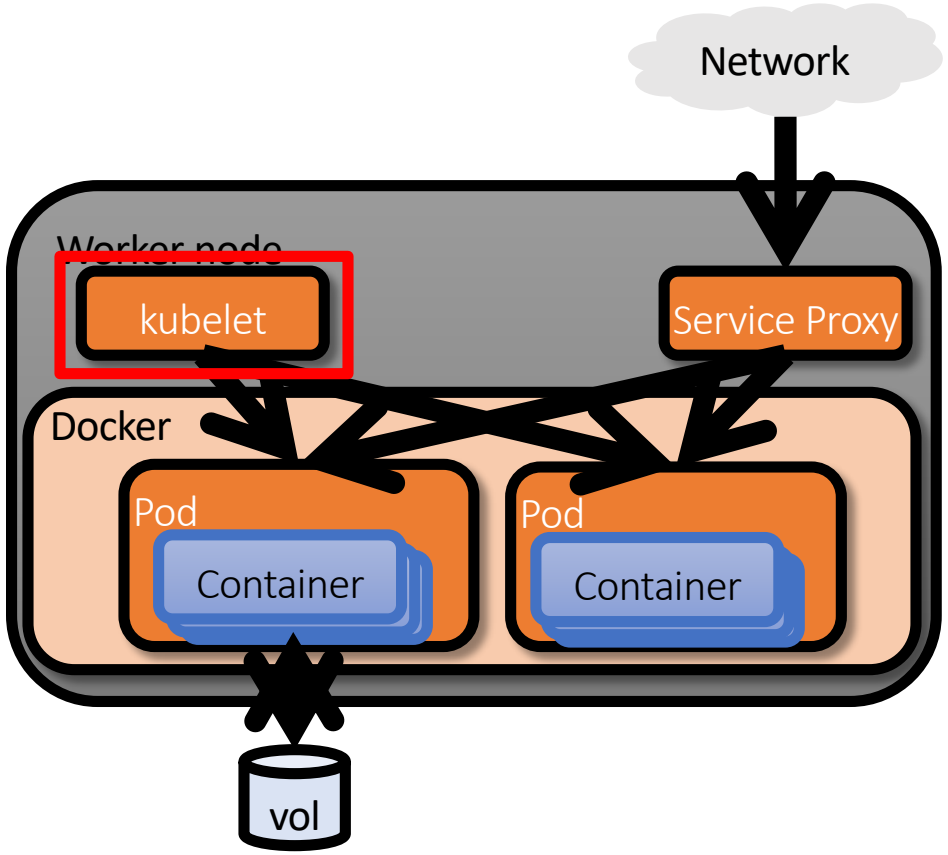



K8s components written in Go (golang.org)

- **cloud-controller-manager**: runs controllers interacting with underlying IaaS providers – alpha feature
  - Allows cloud vendor-specific code to be separate from main K8s system components
- **addons-manager**: creates and maintains cluster addon resources in 'kube-system' namespace, e.g.
  - **Kubernetes Dashboard**: general web UI for application and cluster management
  - **kube-dns**: serves DNS records for K8s services and resources
  - Container resource monitoring and cluster-level logging

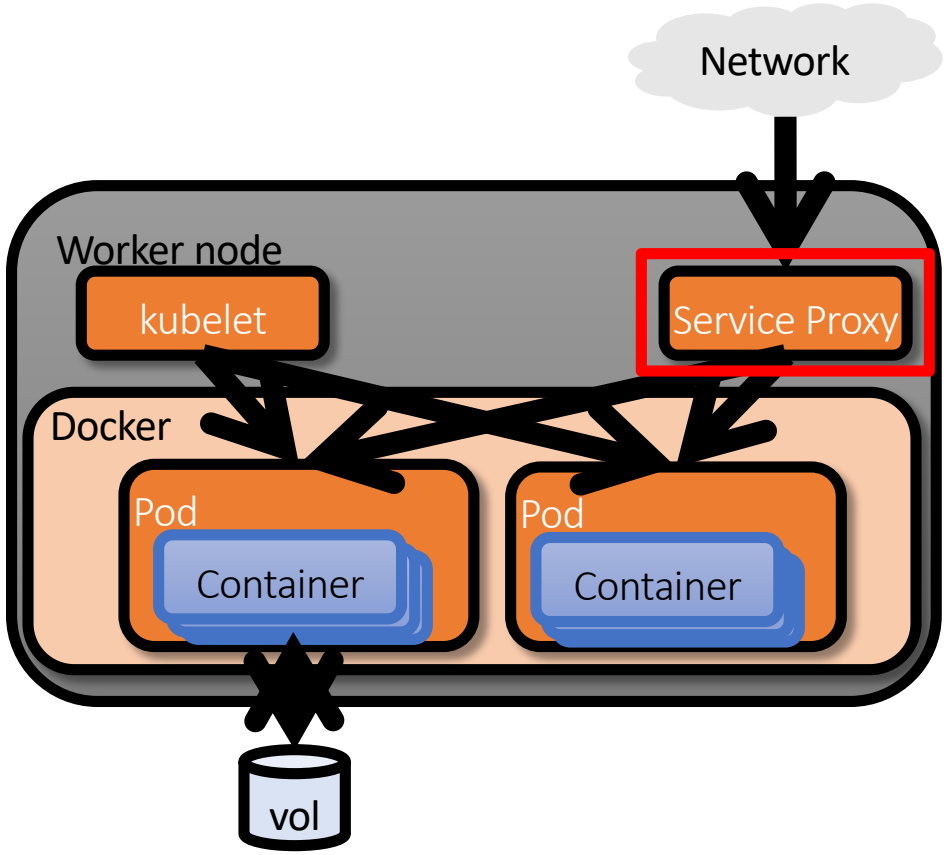
# Kubernetes Worker Node Components

- **kubelet**: local K8s agent that is responsible for operations on the node, including
  - Watching for pod assignments
  - Mounting pod required volumes
  - Running a pod's containers
  - Executing container liveness probes
  - Reporting pod status to system
  - Reporting node status to system
- **Service proxy (kube-proxy):** enables K8s service abstractions by maintaining host network rules and forwarding connections
- **Docker:** runs the containers



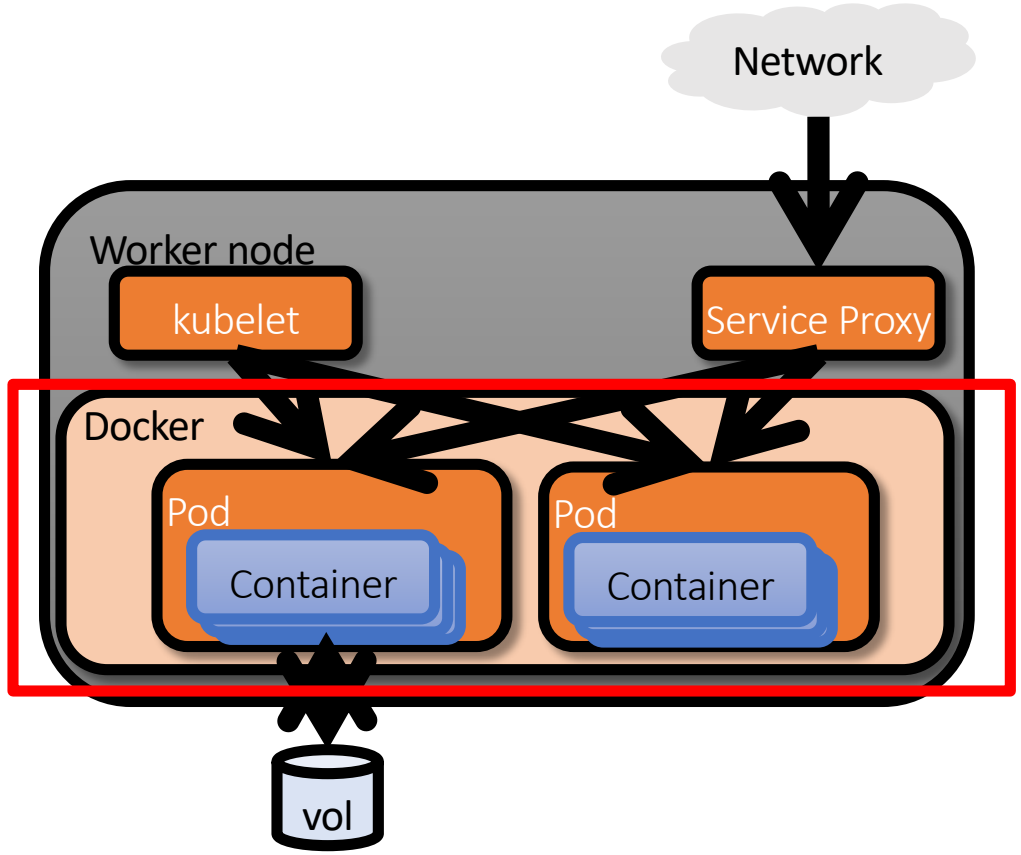
# Kubernetes Worker Node Components

- **kubelet**: local K8s agent that is responsible for operations on the node, including
  - Watching for pod assignments
  - Mounting pod required volumes
  - Running a pod's containers
  - Executing container liveness probes
  - Reporting pod status to system
  - Reporting node status to system
- **Service proxy (kube-proxy):** enables K8s service abstractions by maintaining host network rules and forwarding connections
- **Docker:** runs the containers



# Kubernetes Worker Node Components

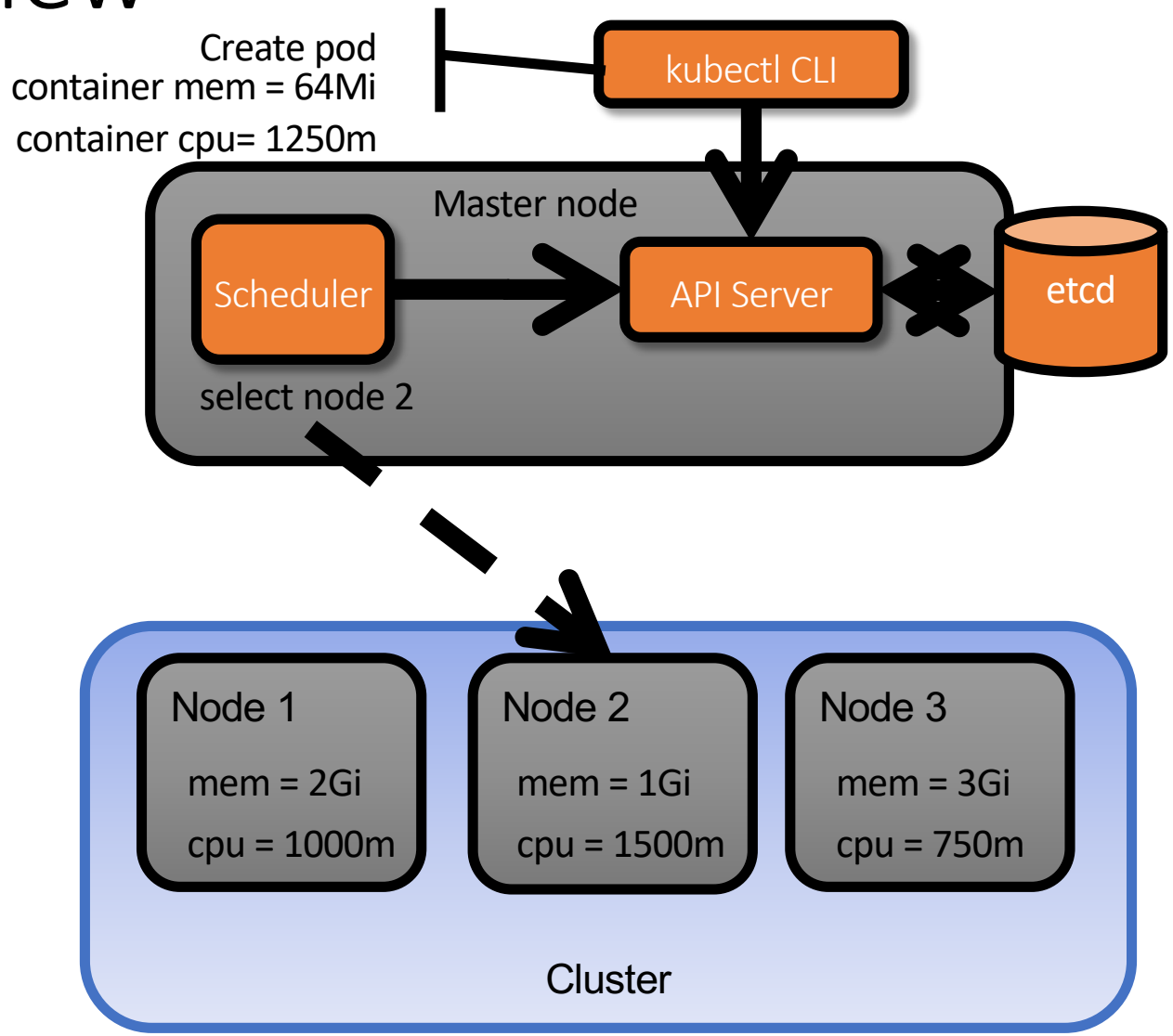
- **kubelet**: local K8s agent that is responsible for operations on the node, including
  - Watching for pod assignments
  - Mounting pod required volumes
  - Running a pod's containers
  - Executing container liveness probes
  - Reporting pod status to system
  - Reporting node status to system
- **Service proxy (kube-proxy):** enables K8s service abstractions by maintaining host network rules and forwarding connections
- **Docker:** container runtime



# Kubernetes Pod Scheduling

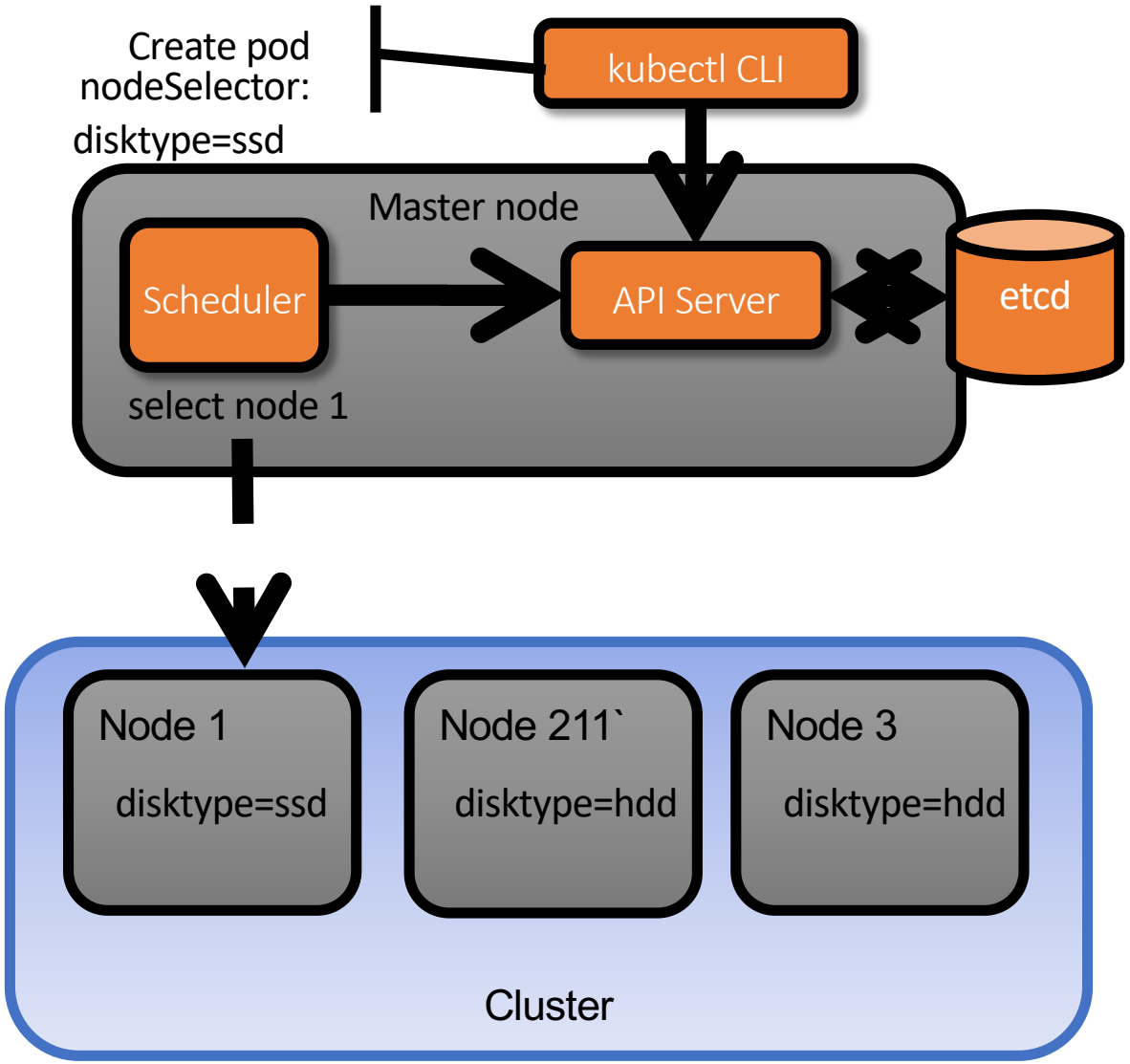
# Pod Scheduling Overview

- kube-scheduler on the master node assigns new pod requests to appropriate worker nodes
- Default scheduler takes account of
  - Available node CPU/RAM
  - Resource requests from new pod – sum of resource requests of pod containers
- Default scheduler will automatically
  - Schedule pod on node with sufficient free resources
  - Spread pods across nodes in the cluster
- Can specify custom schedulers for pods



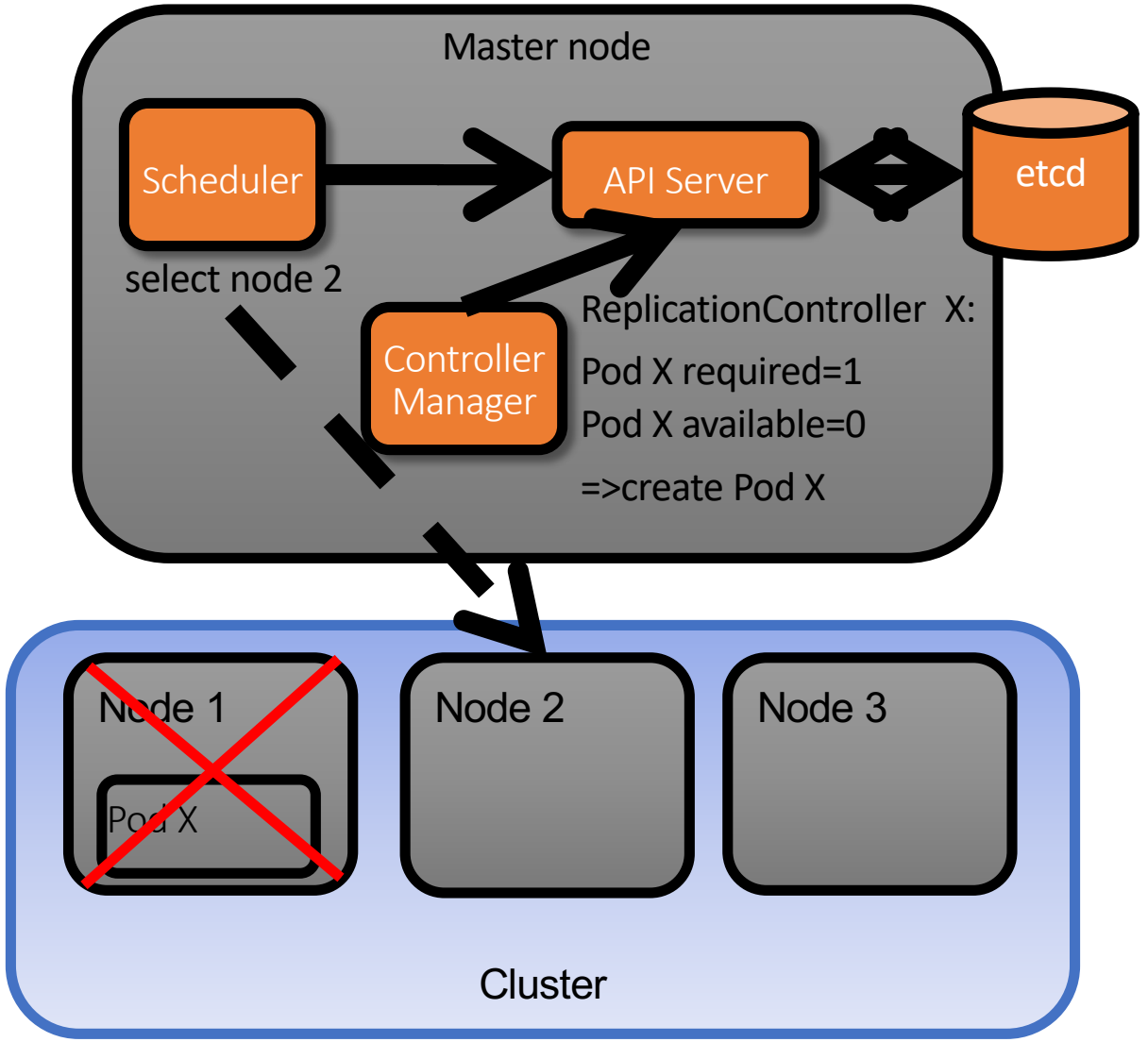
# Pod Scheduling with Default Scheduler

- Nodes carry labels indicating topology and other resource notes
- Users can require pods to be scheduled on nodes with specific label(s) via a nodeSelector in container spec
- K8s 1.6 adds more advanced scheduling options, including
  - Node affinity/anti-affinity generalizing the nodeSelector feature
  - Node taints that prevent scheduling of pods to nodes unless pod 'tolerates' the taint
  - Pod affinity/anti-affinity to control relative placement of pods



# Pod Re-creation Driven by Control Loops

- kube-scheduler performs same node selection operation when new pod created due to e.g. node loss
- kube-controller-manager runs controllers like ReplicationController managing number of pod instances available
- kube-controller-manager will initiate request for new pods as needed, which will be scheduled by kube-scheduler per pod/container spec

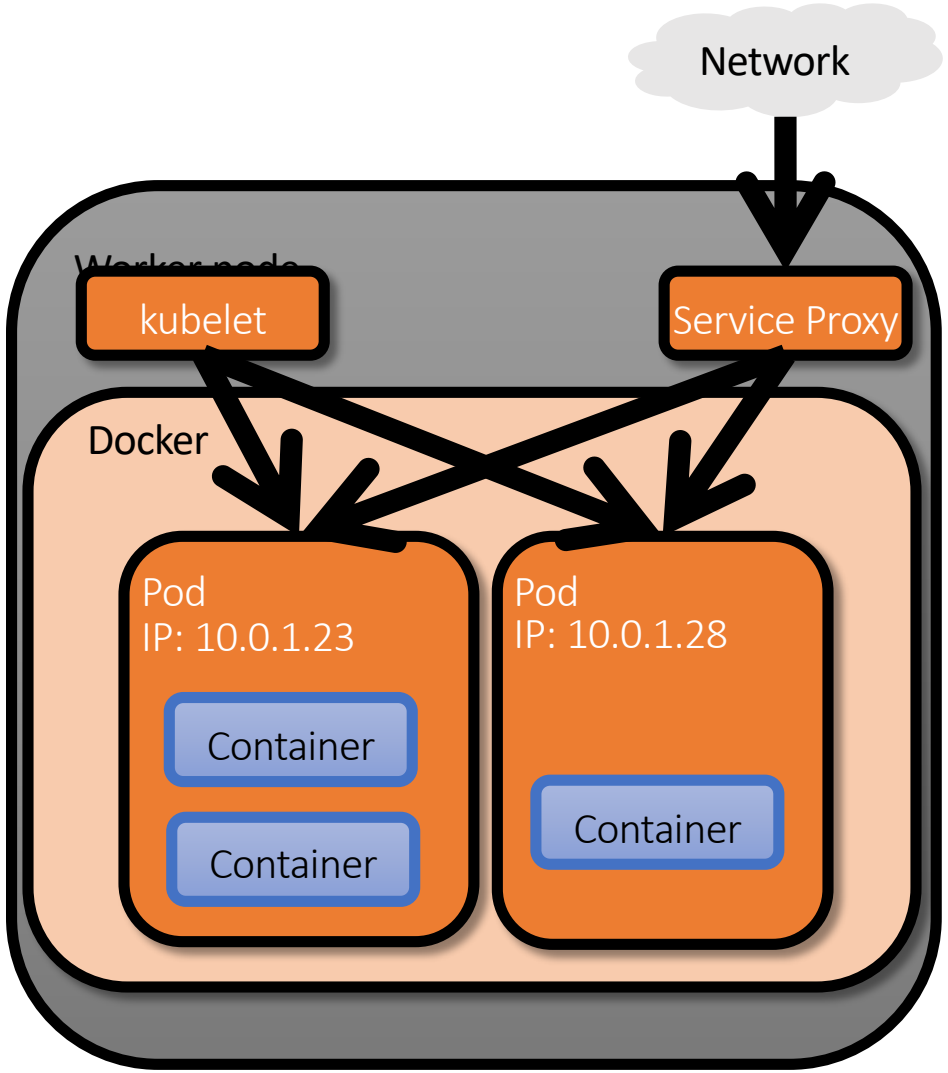



Questions

# Kubernetes Network Models

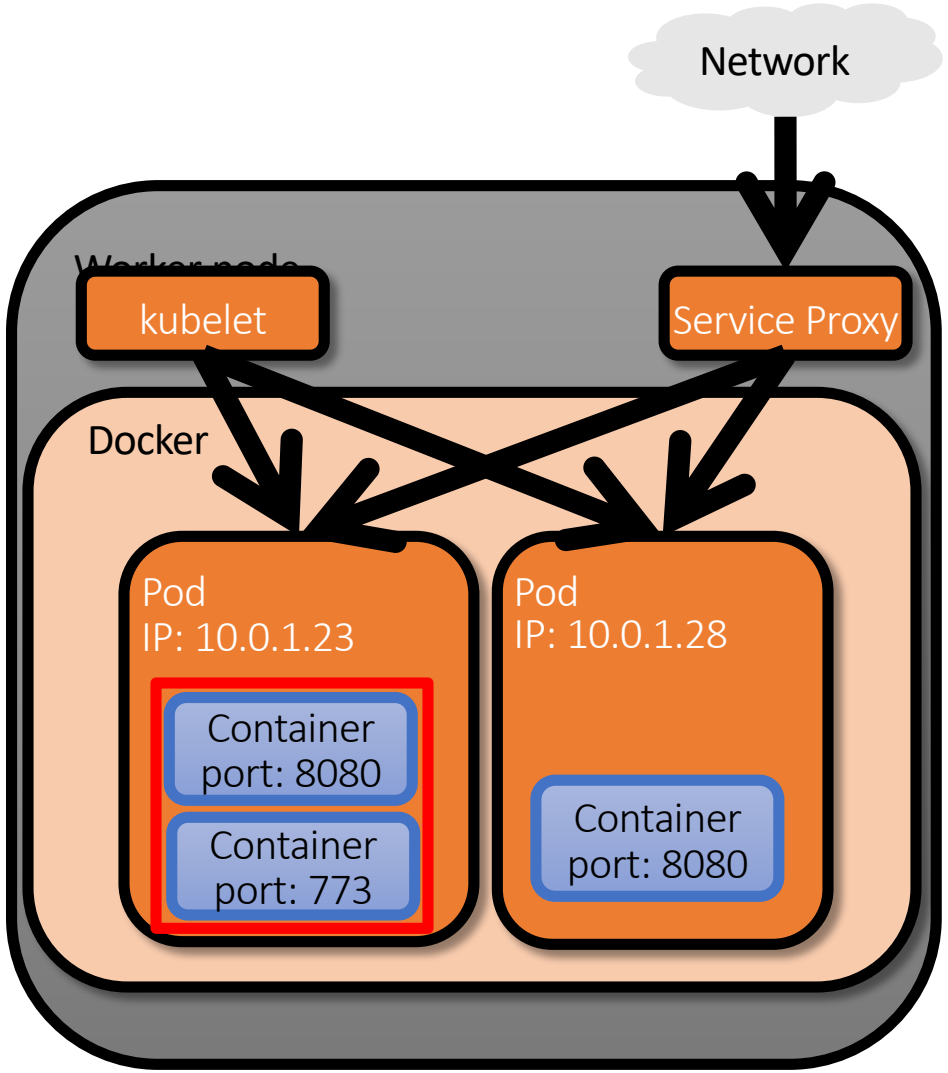
# Networking in Kubernetes – One IP per Pod

- Kubernetes design goal is to simplify configuration – tracking lots of port mappings is hard!
- K8s networking principles:
  - All containers can communicate with other containers without NAT
  - All nodes can communicate with containers without NAT
  - Container sees its own IP exactly as other containers and nodes do
- K8s assigns IP to every **pod** in the cluster as part of flat shared network



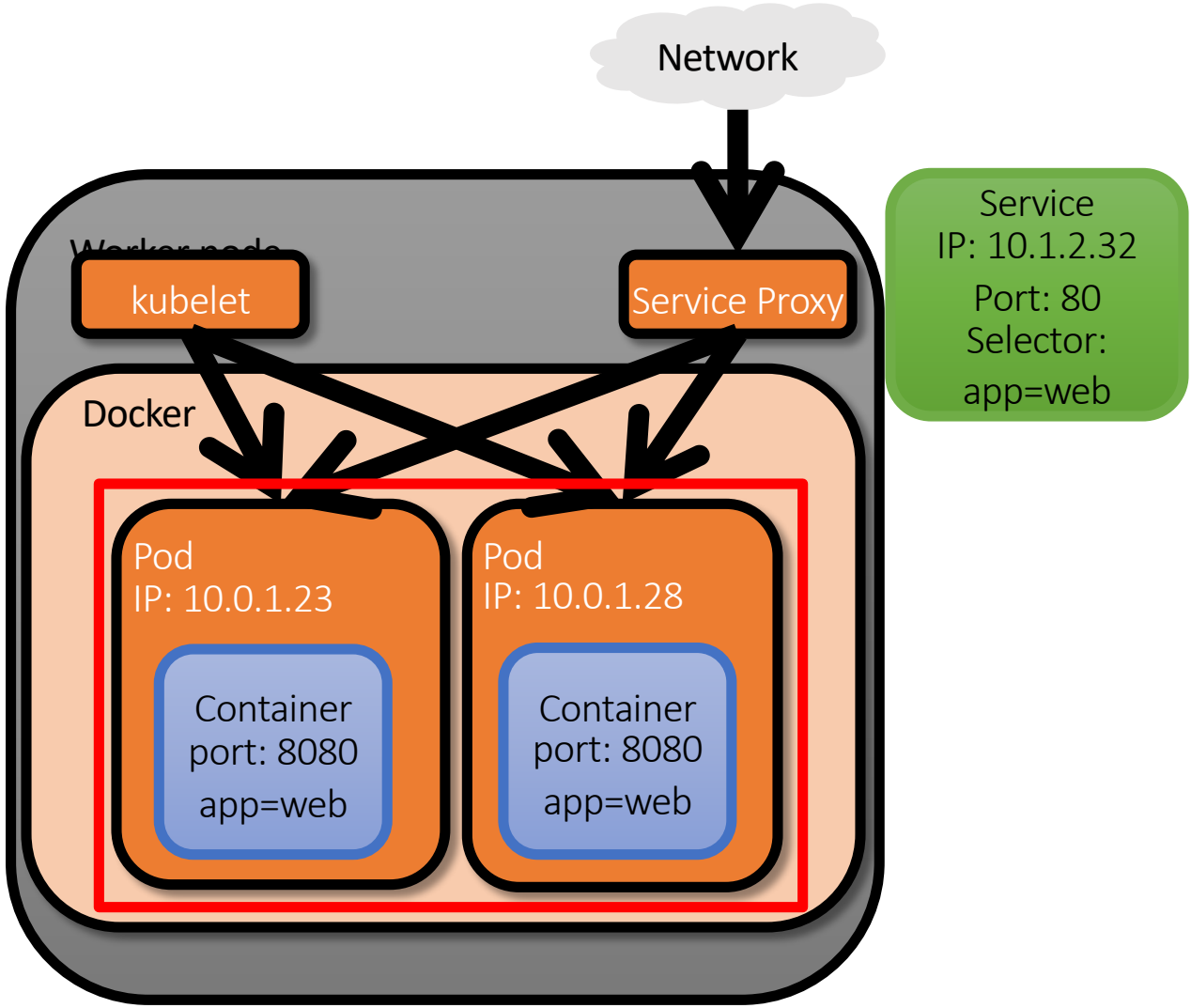
# Container Communications – Same Pod

- Containers in the same pod share the same network namespaces and IP address
- Containers in same pod can reach each other's port on 'localhost'
- Need to manage ports used by co-located containers to avoid conflicts, but done in same pod spec
- Other containers in the cluster can connect to the pod's containers via their ports on the pod's IP

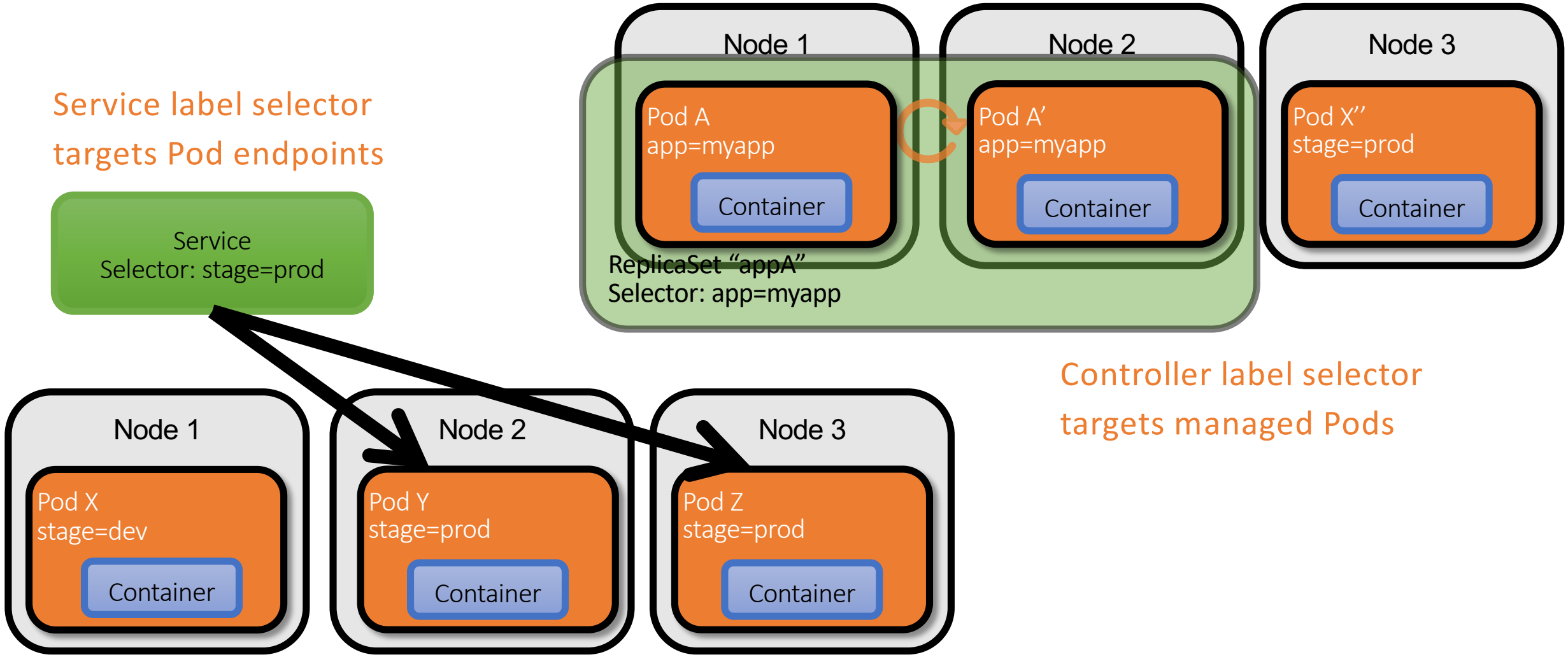


# Abstraction Layer for Applications

- Pod IP's are simple but tricky to track across replicas, restarted pods, etc.
- Service objects permit allocation of a stable IP that forwards to multiple pods matching the service's selector
- kube-proxy maintains iptables rules for service objects on each worker node
  - NATs requests to a backend pod
  - Maps each service port to backend pod port



# Object Grouping and Selection via Labels

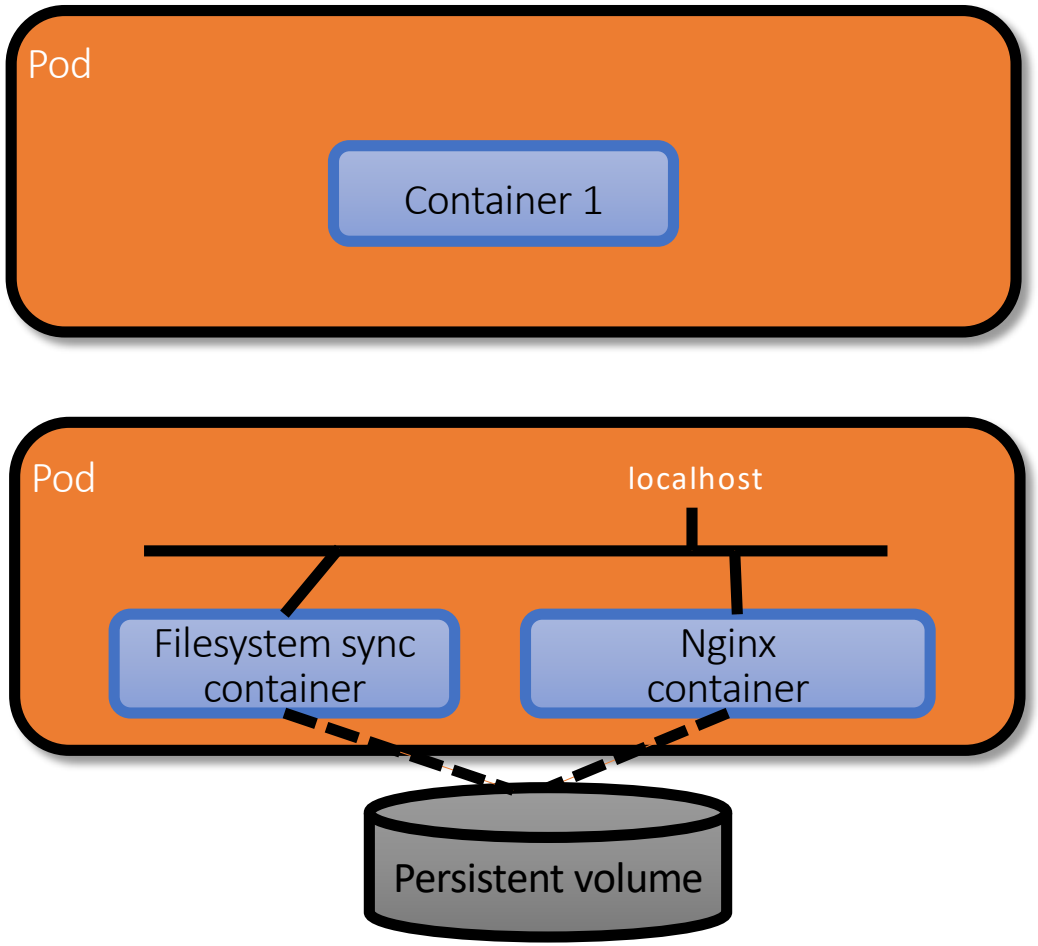


# Kubernetes Pods

# What is a Kubernetes Pod?

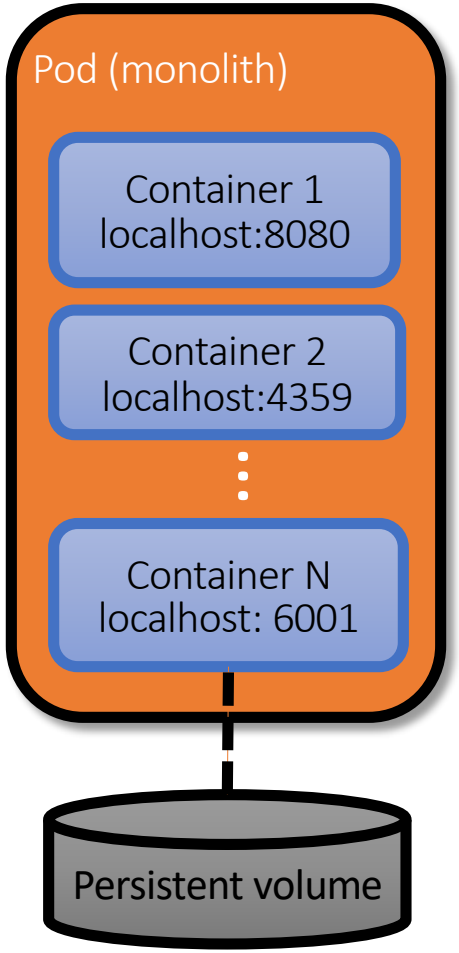
*Kubernetes design intention: Pod == application instance*

- Basic unit of deployment is the **pod,** a set of co-scheduled containers and shared resources
- Pods can include more than one container, for tightly-coupled application components, e.g.
  - Sidecar containers : nginx + filesystem synchronizer to update www from git
  - Content adapter: transform data to common output standard
- Containers in a pod share network namespaces and mounted volumes



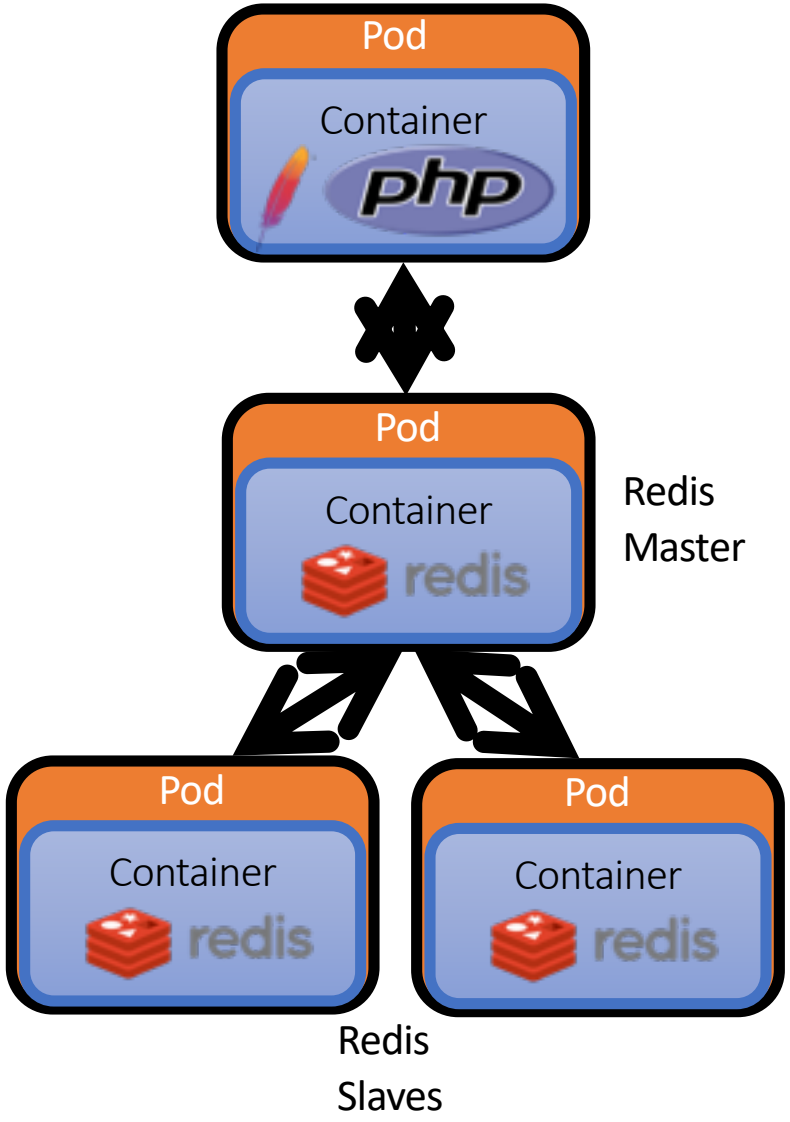
# Pods Enable Deployment Flexibility

- Possible to use a single pod to run a monolithic application
  - Each application process can be built as a container
  - All containers can access each other's ports on localhost
- More advanced features of the K8s system available if application built instead from assemblages of pods, e.g.
  - Web tier: Apache pods
  - Data tier: Redis master/slave pods
- Pods provide scale and elasticity via replication – not possible in the monolith
- Best practice: assume every pod is mortal



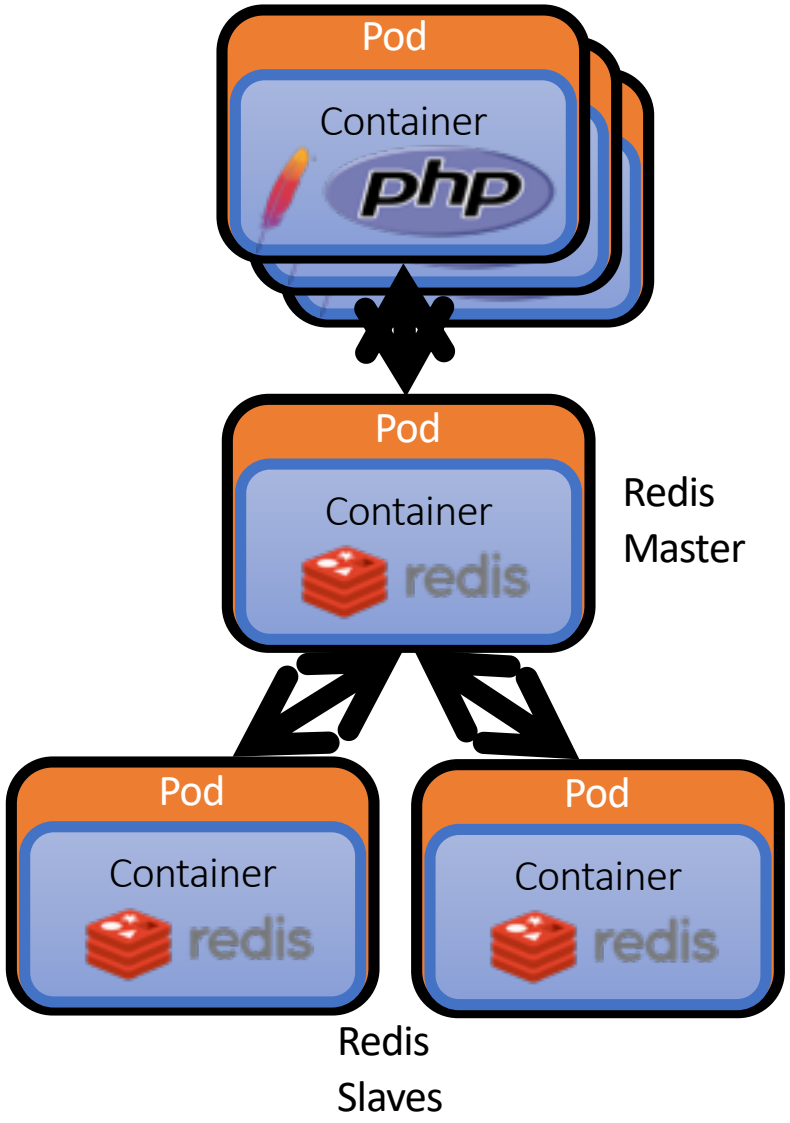
# Pods Enable Deployment Flexibility

- Possible to use a single pod to run a monolithic application
  - Each application process can be built as a container
  - All containers can access each other's ports on localhost
- More advanced features of the K8s system available if application built instead from assemblages of pods, e.g.
  - Web tier: Apache pods
  - Data tier: Redis master/slave pods
- Pods provide scale and elasticity via replication – not possible in the monolith
- Best practice: assume every pod is mortal



# Pods Enable Deployment Flexibility

- Possible to use a single pod to run a monolithic application
  - Each application process can be built as a container
  - All containers can access each other's ports on localhost
- More advanced features of the K8s system available if application built instead from assemblages of pods, e.g.
  - Web tier: Apache pods
  - Data tier: Redis master/slave pods
- Pods provide scale and elasticity via replication – not possible in the monolith
- Best practice: assume every pod is mortal



# Defining a Pod via a Manifest File

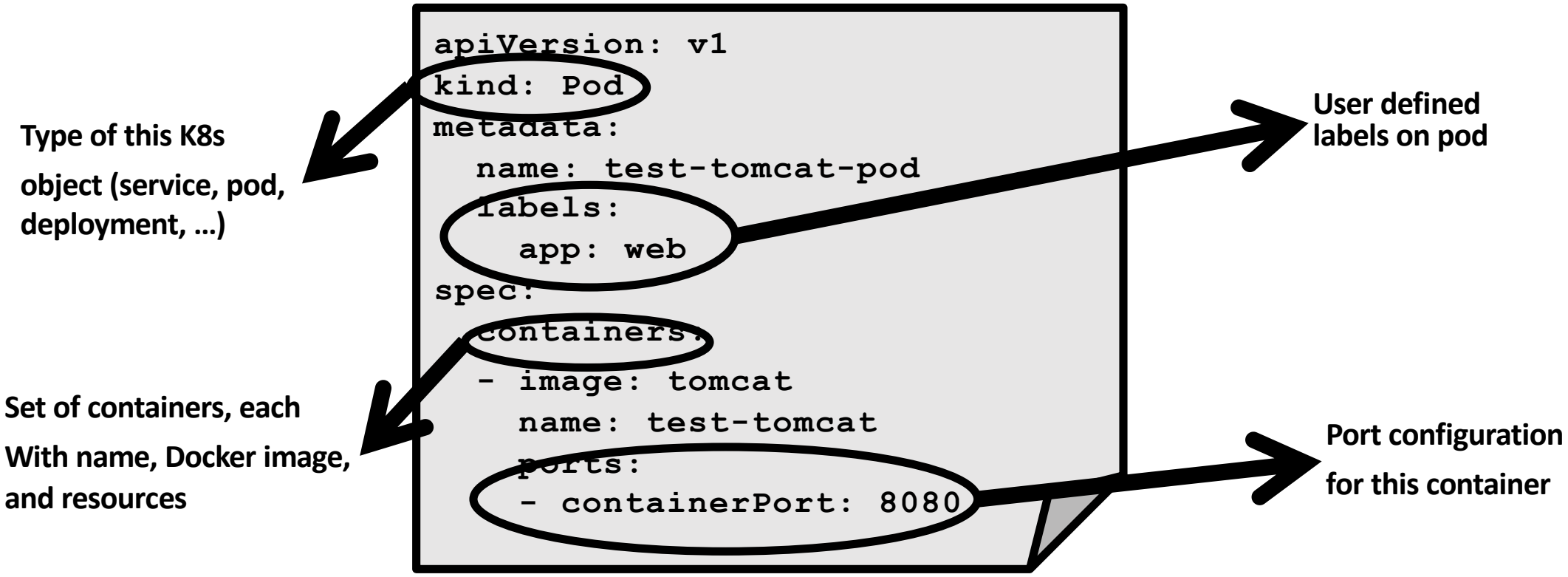
*Like other K8s objects, pods can be defined in YAML or JSON files*

- K8s API accepts object definitions in JSON, but manifests often in YAML
- YAML format used by a variety of other tools, e.g. Docker Compose, Ansible, etc.
- **kind** field value is 'Pod'
- **metadata** includes
  - **name** to assign to pod
  - **label** values
- **spec** includes specifics of container images, ports, and other resources

```
apiVersion: v1
kind: Pod
metadata:
  name: test-tomcat-pod
  labels:
    app: web
spec:
  containers:
  - image: tomcat
    name: test-tomcat
    ports:
    - containerPort: 8080
```

# Looking at a Pod Manifest File

simplepod.yaml




- Configuration options similar to creating Docker container directly

# Defining a Pod with Multiple Containers

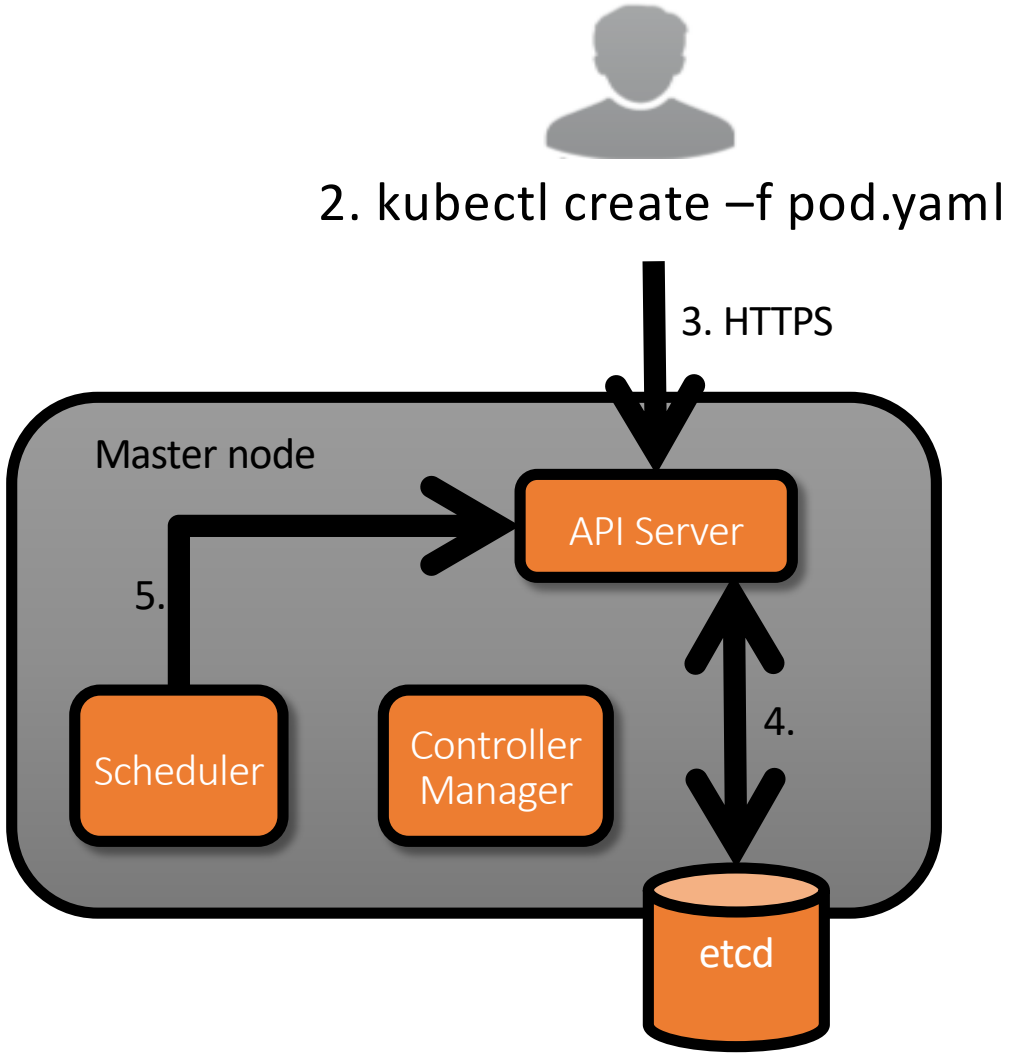
```
apiVersion: v1
kind: Pod
metadata:
  name: test-tomcat-pod
spec:
  containers:
  - image: tomcat
    name: test-tomcat
    ports:
    - containerPort: 8080
  - image: mysql
    name: test-mysql
    ports:
    - containerPort: 3306
```

multipod.yaml

- Pod spec can contain multiple containers from different images
- Containers in pod share local network context and cluster IP for pod

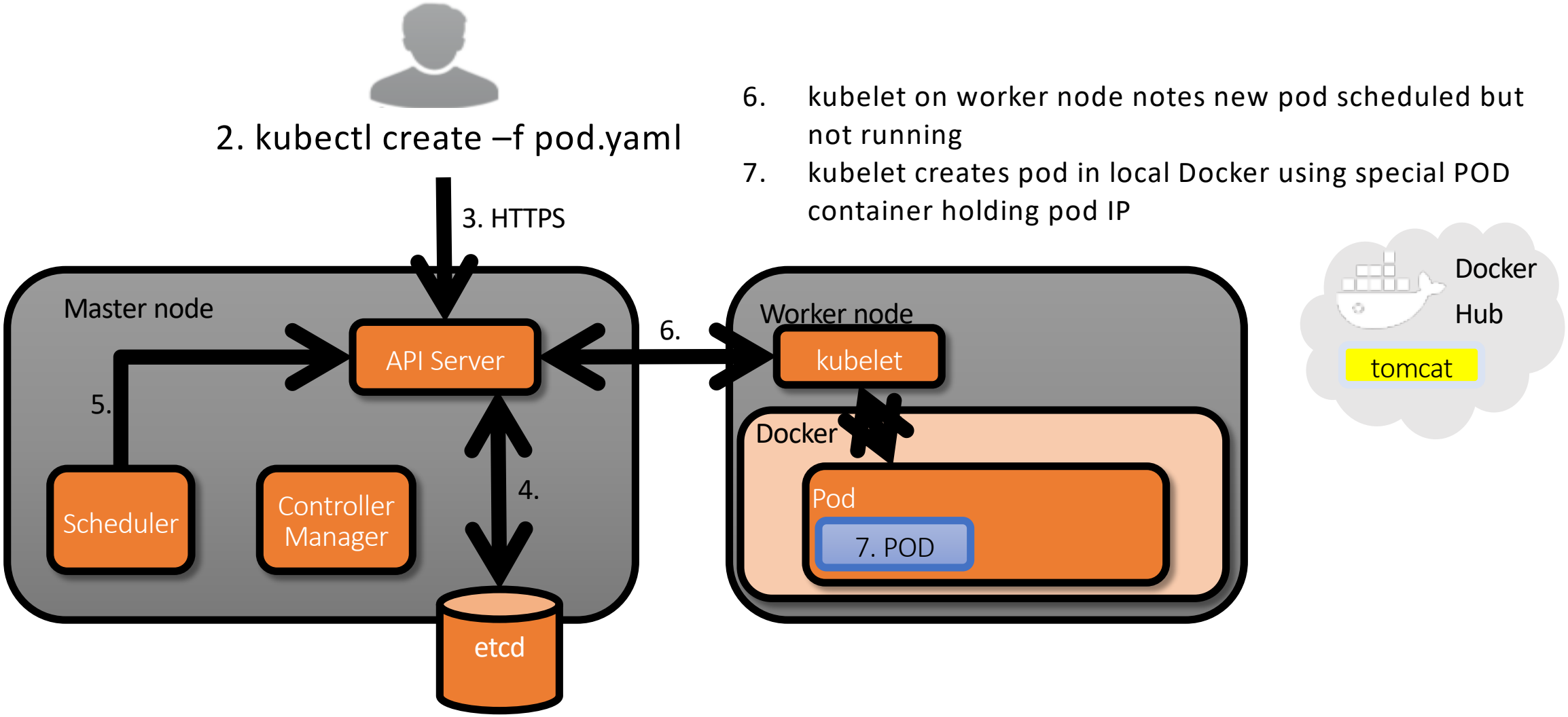
# Pod Creation and Management

# Pod Creation Process



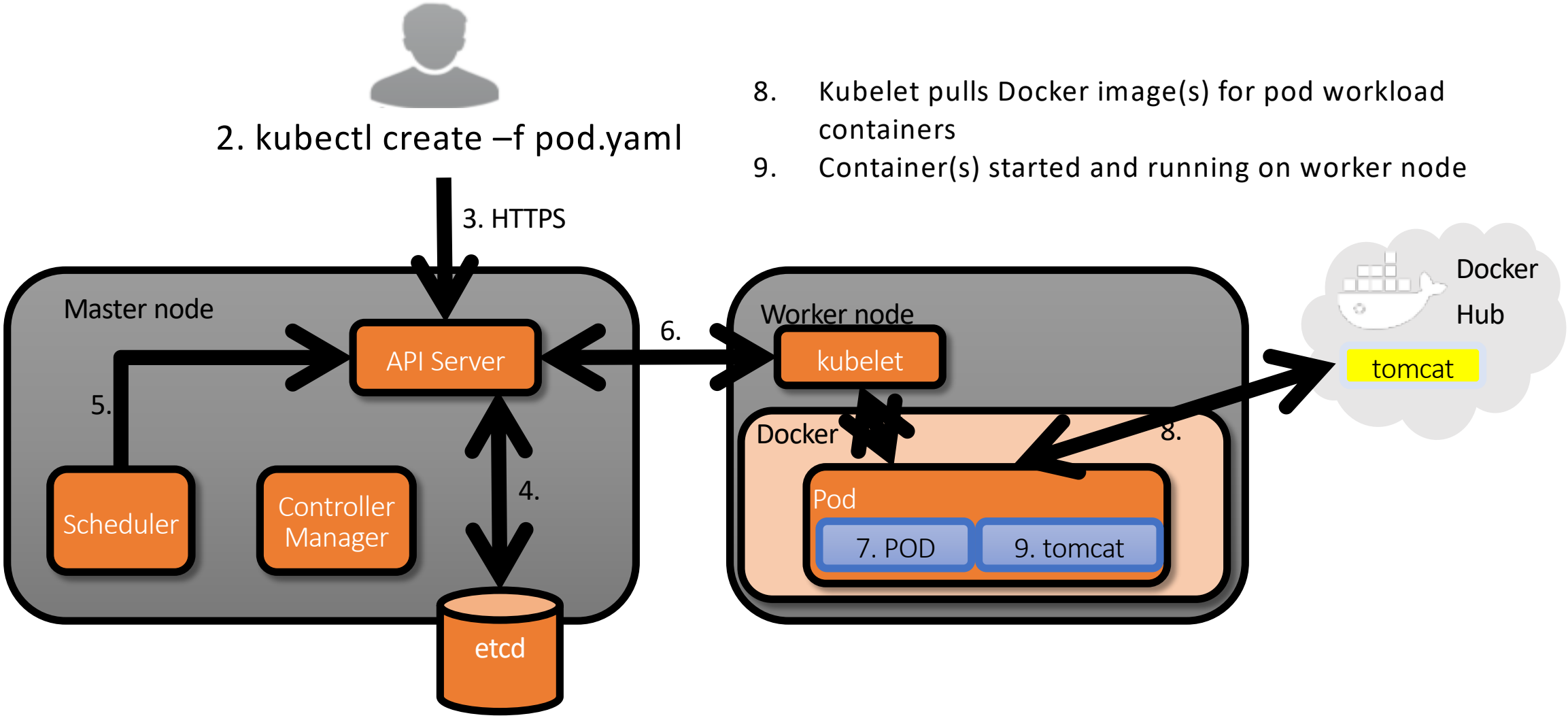

1. User writes a pod manifest file
2. User requests creation of pod from manifest via CLI
3. CLI tool marshals parameters into K8s RESTful API request (HTTP POST)
4. kube-apiserver creates new pod object record in etcd, with no node assignment
5. kube-scheduler notes new pod via API
   a. Selects node for pod to run on
   b. Updates pod record via API with node assignment

# Pod Creation Process




6. kubelet on worker node notes new pod scheduled but not running
7. kubelet creates pod in local Docker using special POD container holding pod IP

# Pod Creation Process

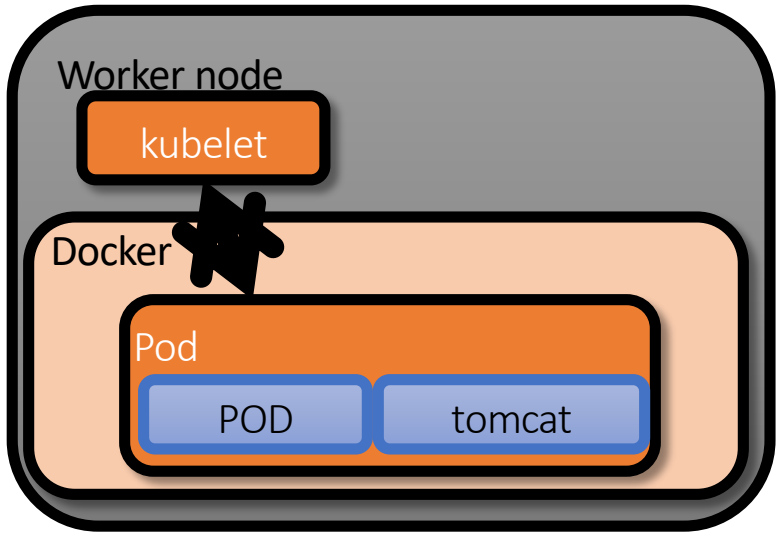



8. Kubelet pulls Docker image(s) for pod workload containers
9. Container(s) started and running on worker node

# Pod Lifecycles

- By default, K8s Pods have an indefinite lifetime, which is not immortality
  - **restartPolicy** of Always by default
  - **restartPolicy** of Never or OnFailure also available for terminating jobs
- Node's kubelet will create and keep running containers for pods assigned to node, per the pod specs
- If a Pod container fails to start, or unexpectedly exits, kubelet will restart it
  - Can see container lifecycle events via 'kubectl describe pod <PODNAME>'
- If node is lost, its Pods are also lost – K8s will not rebind Pods to another node

```
apiVersion: v1
kind: Pod
metadata:
  name: test-tomcat-pod
  labels:
    app: web
spec:
  containers:
  - image: tomcat
    name: test-tomcat
    ports:
    - containerPort: 8080
```



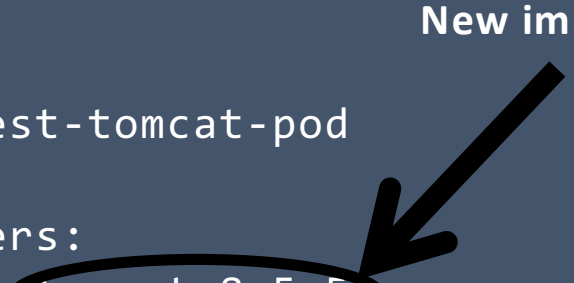
# Modifying a Pod

*Change the container version*

- You can make changes to the desired state of a pod via updating the manifest file
- Changes can then be applied to the pod via the command
  - `kubectl apply –f <manifest.yaml>`
- Changing a container image as shown will result in K8s automatically killing and recreating the pod's workload container

```
$ vi simplepod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: test-tomcat-pod
spec:
  containers:
  - image: tomcat:8.5.5
    name: test-tomcat
    ports:
    - containerPort: 8080
```



# Modifying a Pod

```
$ kubectl apply -f simplepod.yaml
pod "test-tomcat-pod" configured

$ kubectl describe pod test-tomcat-pod
Name:           test-tomcat-pod
Namespace:      default
...
Labels:         tier=frontend
Status:         Running
...
Containers:
  test-tomcat:
    Image:              tomcat:8.5.5
    Image ID:
    Port:               8080/TCP
    State:                      Running
...
```

# Labeling Pods

*User-defined labels help organize K8s resources*

- Labels are key/value pairs that users can assign and update on any K8s resources, including pods
- Other K8s objects, like controllers, use labels to select pods to govern
- Labels can also be used to filter data queries with *kubectl,* e.g.
  - `kubectl get pods -l <label=value>`
- Labels can be used to distinguish pods on any criteria, such as
  - Application, application tier, version, environment state, etc.
- K8s system does not require specific labels to be used – all user-defined

# Labeling a Pod

```
$ kubectl label pod test-tomcat-pod tier=frontend
pod "test-tomcat-pod" labeled


$ kubectl describe pods test-tomcat-pod
Name:           test-tomcat-pod
...
Labels:         tier=frontend


$ kubectl get pods -l tier=frontend
NAME              READY     STATUS    RESTARTS   AGE
test-tomcat-pod   1/1       Running   0          1d
```

# Reviewing Labels on Pods

*Changing kubectl output*

- You can display pod labels via a flag on the *kubectl* command

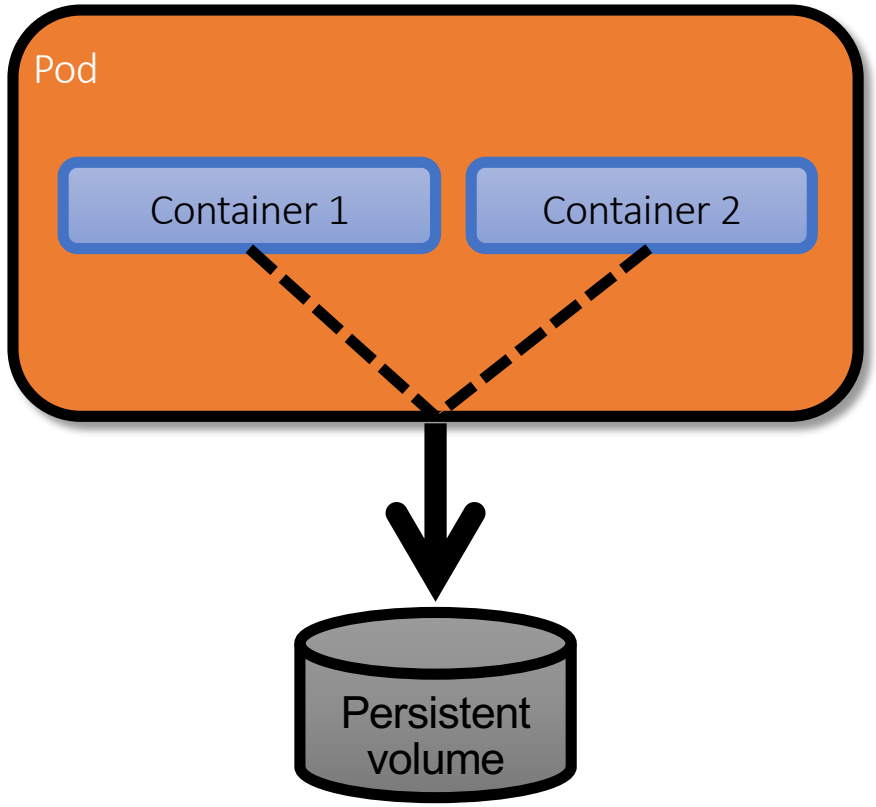
```
$ kubectl get pods --show-labels
NAME            READY   STATUS   RESTARTS AGE  LABELS
test-tomcat-pod 1/1     Running  1         1d  tier=frontend

$ kubectl get pods --show-labels --namespace=kube-system
NAME                          READY     STATUS    RESTARTS   AGE       LABELS
kube-addon-manager-minikube   1/1       Running   3          8d        component=kube-addon-
manager,kubernetes.io/minikube-addons=addon-manager,version=v6.4-alpha.1
kube-dns-v20-mm0zl            3/3       Running   9          8d        k8s-app=kube-
dns,version=v20
kubernetes-dashboard-kc9rk    1/1       Running   3          8d        app=kubernetes-
dashboard,kubernetes.io/cluster-service=true,version=v1.6.0
```

# Deleting Pods

*Pod deletion will discard all local pod resources*

- When deleting a Pod, its containers will be removed and pod IP relinquished
- If an application needs to persist data, its pods must be configured to use persistent volumes for storage
- If a node dies, its local pods are also gone
- Best practice: use controller resources instead of managing pods directly
- Best practice: use service resources to build reliable abstraction layers for clients



# Lab: Create Pods

# Kubernetes Services Overview

# Kubernetes Service Objects:Microservices

Services provide abstraction between layers of an application

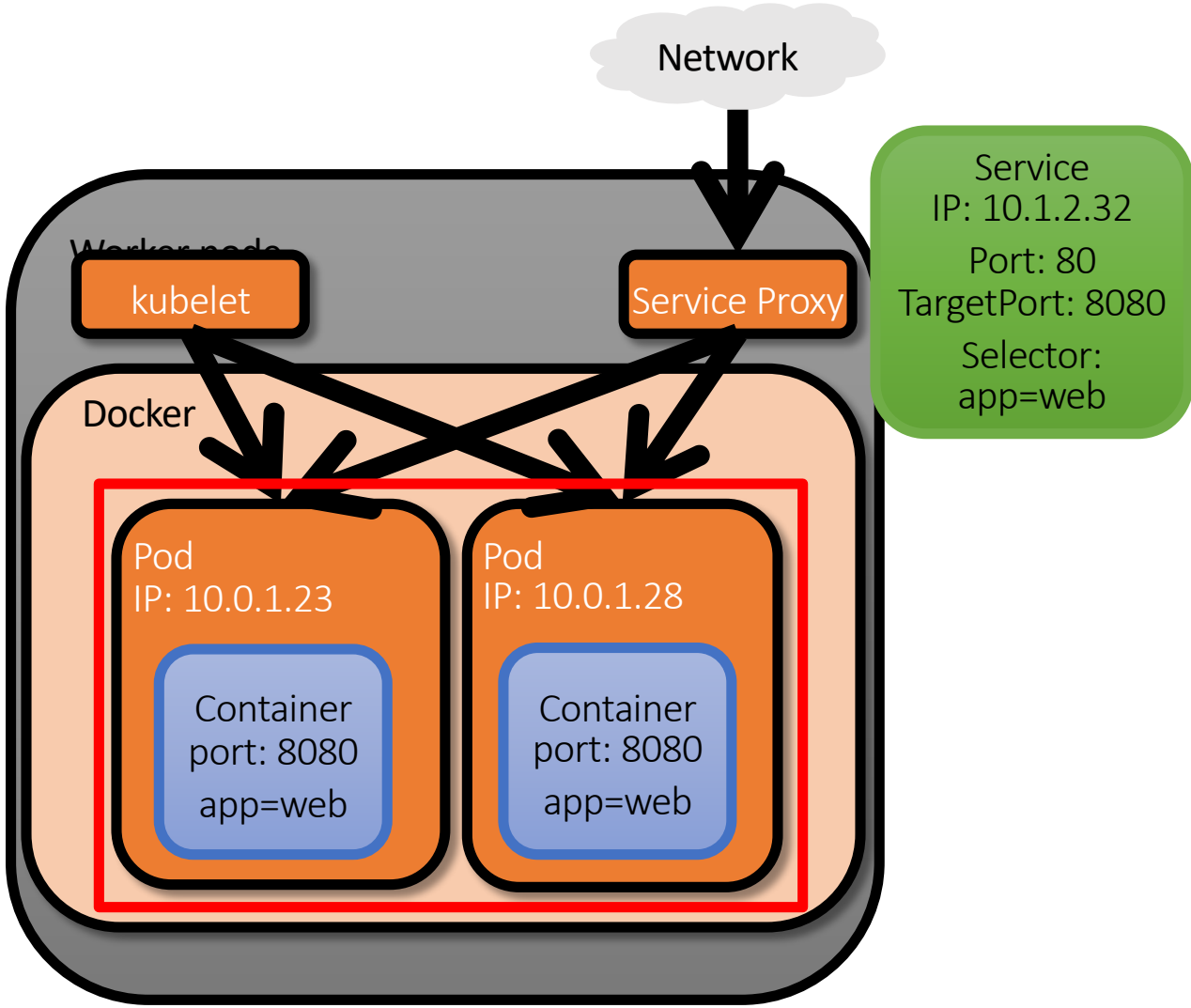
- **Service** object provides a stable IP for a collection of Pods
- Services use a label **selector** to target a specific set of pods as endpoints to receive proxied traffic
- Clients can reliably connect via the service IP and port(s), even as individual endpoint pods are dynamically created & destroyed
- Can model other types of backends using services without selectors

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: wordpress
    tier: frontend
  type: LoadBalancer
```

*sampleservice.yaml*

# Services Provide Abstraction Layer for Applications

- Services can be used for communications between application tiers
- Services can also be used to expose applications outside the K8s cluster
- Services distribute requests over the set of Pods matching the service's selector
  - Service functions as TCP and UDP proxy for traffic to Pods
  - Service maps its defined ports to listening ports on Pod endpoints



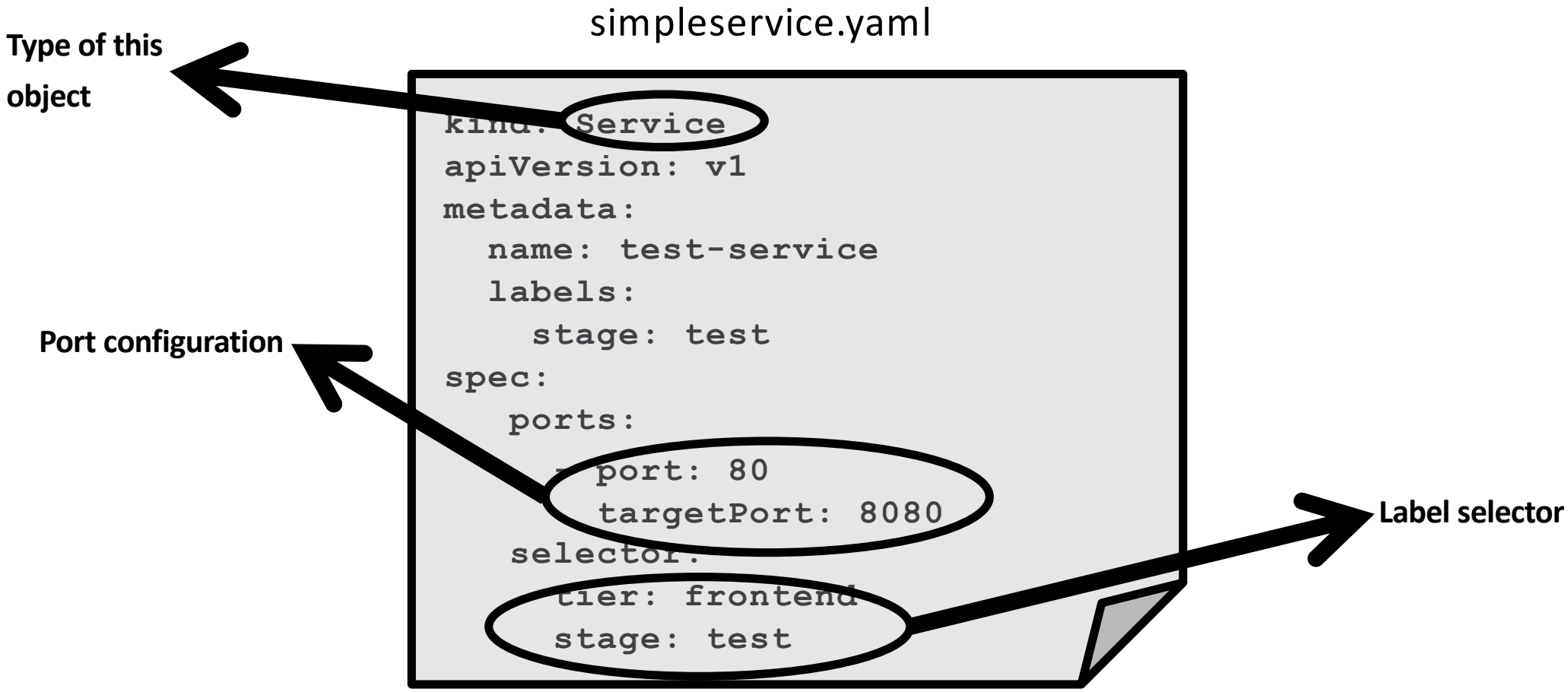
# Defining a Service Using a Manifest File

*Services can be defined in YAML or JSON, like other K8s resources*

- **kind** field value is 'Service'
- **metadata** includes
  - **name** to assign to Service
- **spec** includes the ports associated with the Service
  - **port** is the Service's port value
  - t**argetPort** is connection port on selected pods (default: **port** value)
- **selector** specifies a set of label KV pairs to identify the endpoint pods for the Service

```
kind: Service
apiVersion: v1
metadata:
  name: test-service
  labels:
    stage: test
spec:
   ports:
     - port: 80
       targetPort: 8080
   selector:
     tier: frontend
     stage: test
```

# Reviewing a Service Manifest File



# ServiceTypes and Exposing Applications

- By default, a Service is assigned a cluster-internal IP – good for back-ends
  - **ServiceType** ClusterIP
- To make a front-end Service accessible outside the cluster, there are other ServiceTypes available
  - **ServiceType** NodePort exposes Service on each Node's IP on a static port
  - **ServiceType** LoadBalancer exposes the Service externally using cloud provider's load balancer
- Can also use a Service to expose an external resource via **ServiceType** ExternalName

```
kind: Service
apiVersion: v1
metadata:
  name: test-service
  labels:
    stage: test
spec:
   ports:
     - port: 80
       targetPort: 8080
       nodePort: 30080
   selector:
     tier: frontend
   type: NodePort
```

# Exposing Services at L7 through Ingresses

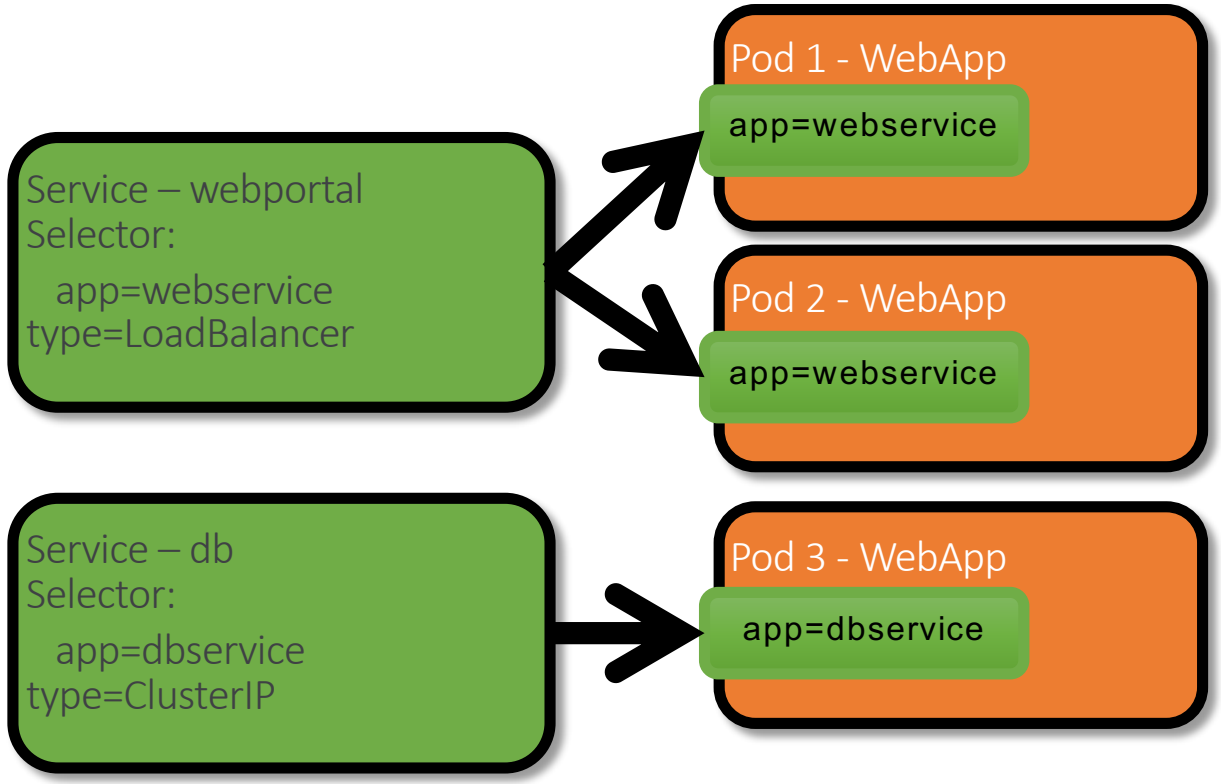
- Kubernetes also provides the facility to define **Ingress** resource to configure an external loadbalancer at L7
- Spec of Ingress resource is a set of rules matching HTTP host/url paths to specific Service backends
- Ingresses require the cluster to be running an appropriately configured Ingress controller to function (e.g. nginx)
- Useful for implementing fanout, Service backends for virtual hosts, etc.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  rules:
  -host: bar.foo.com
   http:
      paths:
      - path: /first
        backend:
          serviceName:
firstservice
          servicePort: 80
      - path: /second
        backend:
          serviceName:
secondservice
          servicePort: 80
```

# Selecting Pods as Service Endpoints

*Service's pod selector based on labels*

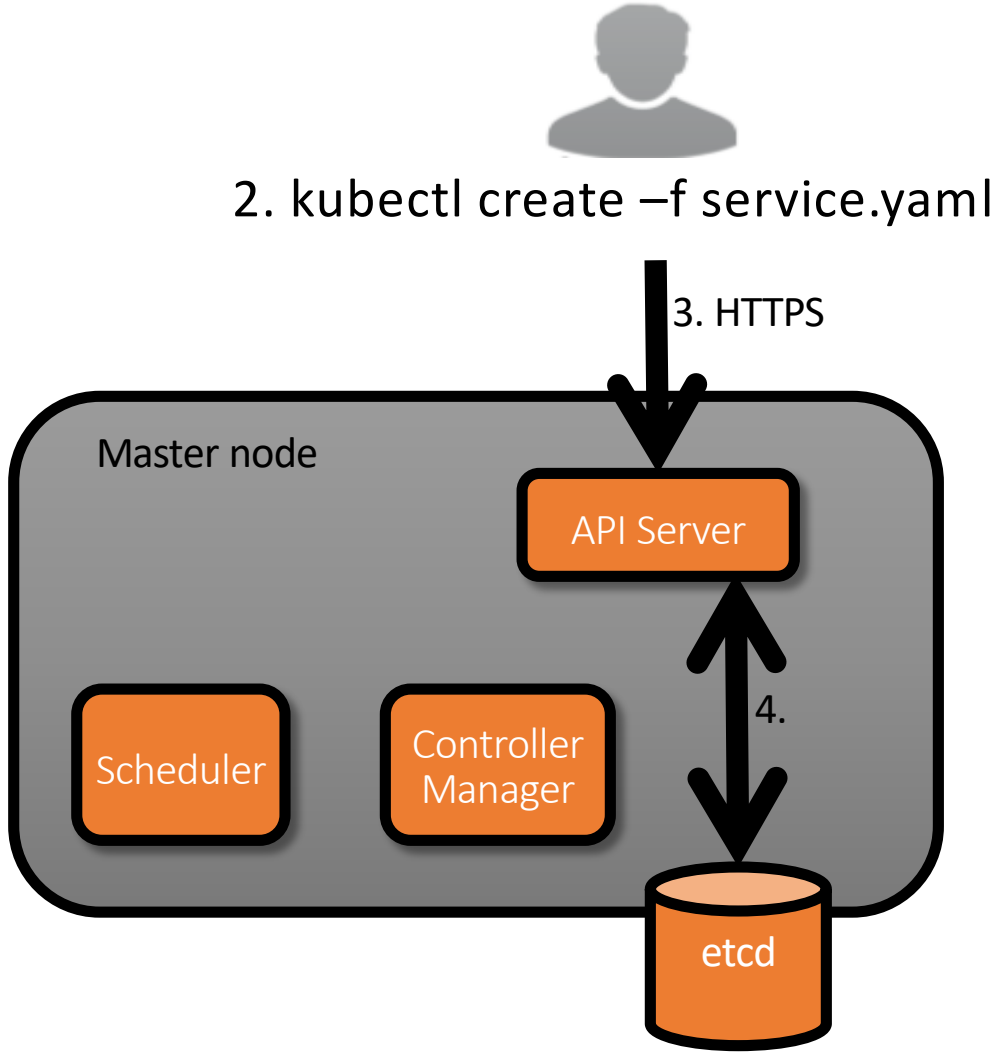
- Multiple pods can have the same label, unlike pod names which are unique in the namespace
- K8s system re-evaluates Service's selector continuously
- K8s maintains **endpoints** object of same name with list of pod IP:port's matching Service's selector




Questions

# Services Management

# Service Creation Process

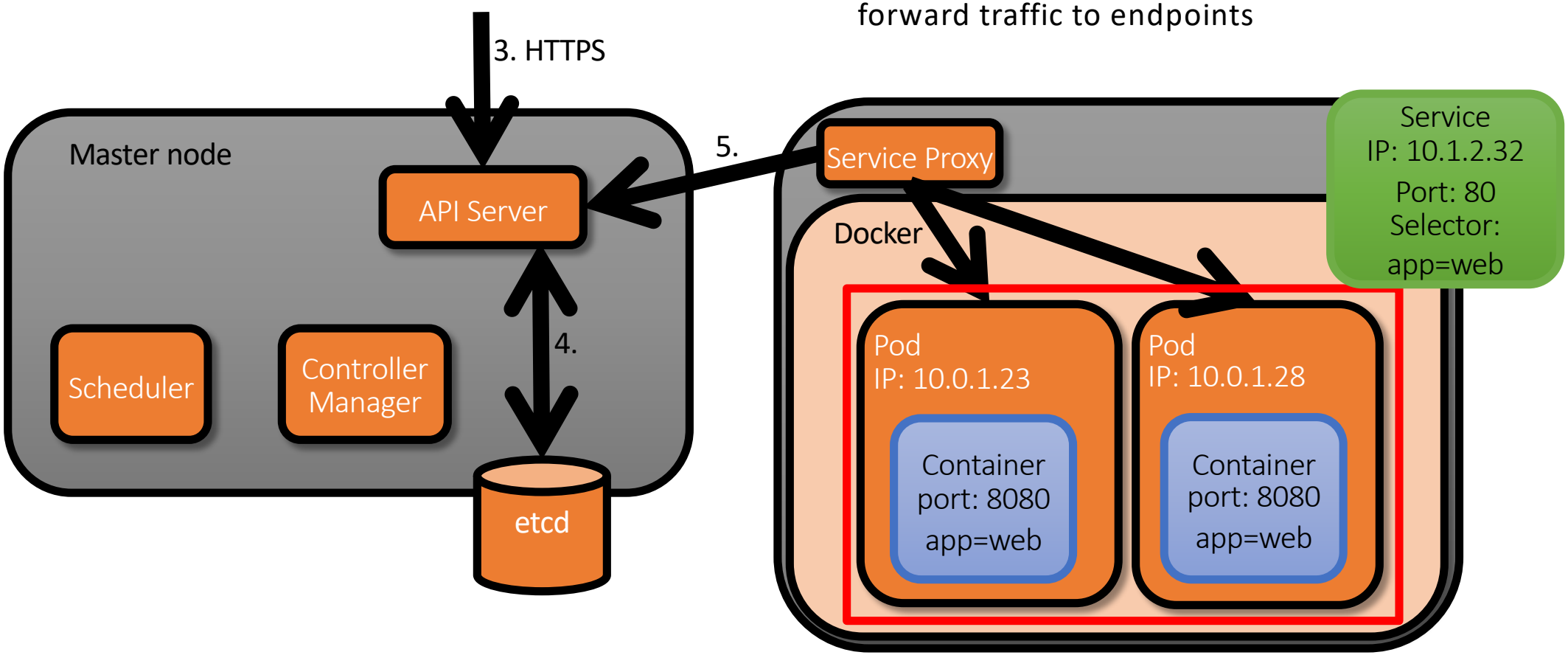



1. User writes a Service manifest file
2. User requests creation of Service from manifest via CLI
3. CLI tool marshals parameters into K8s RESTful API request (HTTP POST)
4. kube-apiserver creates new Service object record in etcd

# Service Creation Process

2. kubectl create –f service.yaml

5. kube-proxy on node sees new Service
6. kube-proxy configures local iptables rules to forward traffic to endpoints



# Accessing Services Externally

## *Checking the service external IP*

- Configured as type LoadBalancer, a configured cluster will provide an externally accessible IP for your Service

```
$ kubectl get services test-service
NAME         CLUSTER-IP   EXTERNAL-IP     PORT(S)    AGE
test-service 10.3.247.123 104.154.105.198 8080/TCP   4h
```

# Deleting Services

*Services can be deleted anytime*

- Service deletion does not affect pods targeted by the Service's selector
- Can be done referencing the Service name or a manifest for the resource

```
$ kubectl delete -f simpleservice.yaml
```

```
$ kubectl delete service test-service
```

# Service Discovery via DNS

*Kubernetes advertises services via cluster DNS*

- Kubernetes uses a cluster-internal DNS addon to create and manage records for all Services in the cluster
- Pod's DNS search list includes its own namespace and cluster default domain by default

```
$ kubectl get svc
NAME           CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
kubernetes     10.0.0.1     <none>        443/TCP          11d
test-service   10.0.0.102   <nodes>       8080:30464/TCP   2d

kubectl exec -ti busybox1 -- nslookup test-service.default
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      test-service.default
Address 1: 10.0.0.102 test-service.default.svc.cluster.local
```

# Service Discovery via Environment Variables

*Kubernetes advertises services via local environment*

- When a Pod is started on a node, kubelet will create environment variables for Services in the Pod's namespace and system namespace

```
$ kubectl exec -ti busybox -- printenv | grep SERVICE
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
TEST_SERVICE_SERVICE_HOST=10.0.0.102
TEST_SERVICE_PORT_8080_TCP_PROTO=tcp
TEST_SERVICE_PORT_8080_TCP_PORT=8080
TEST_SERVICE_PORT=tcp://10.0.0.102:8080
TEST_SERVICE_PORT_8080_TCP_ADDR=10.0.0.102
TEST_SERVICE_SERVICE_PORT=8080
TEST_SERVICE_PORT_8080_TCP=tcp://10.0.0.102:8080
```

# Lab: Multi-container deployment