Lecture 5 – Multi-service applications

Gianluca Quercini

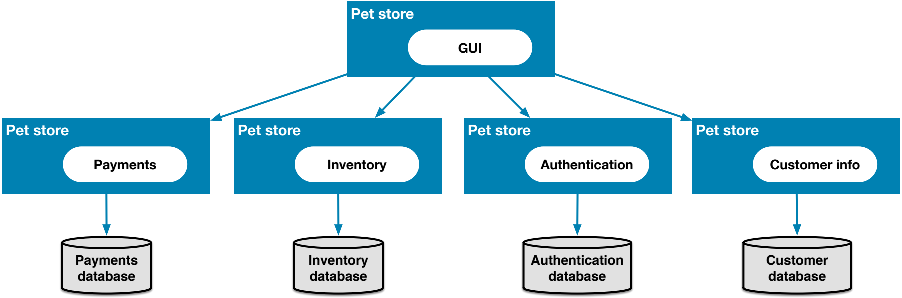
CentraleSupélec

SG8, 2019 – 2020

CentraleSupélec

Previously in Lecture 2: microservices

- Application composed of many **loosely coupled** and **independently deployable** smaller components, called **services**. ▸ Ref
- Each service implements a specific feature of the application.
- Services interact by using APIs.



Services, images and containers

Definition (Services and containers)

A **service** is an application that is packaged as a container **image** from which one to several **containers** can be created and run.

- **Service** $\equiv$ **image**
  - with some configuration options.
- **Service instance** $\equiv$ **container**.
- Multiple service instances (i.e., containers) can be created and run.
  - To serve many requests.
- Services are **independent** from one another.
  - Containers provide the **isolation** properties that we need.
- **Multi-service application**: application composed of more than one service.

What we'll learn in this lecture

1. Build and run **multi-service applications** on a **single host**.
   - Overview of **Docker compose**.
2. Build and run **multi-service applications** across **multiple hosts**.
   - Definition and role of an **orchestrator**.
   - Introduction to **Kubernetes**.
3. Overview of **multi-service applications** in the **Cloud**.
   - Introduction to **Amazon Web Services** (AWS).

Docker compose

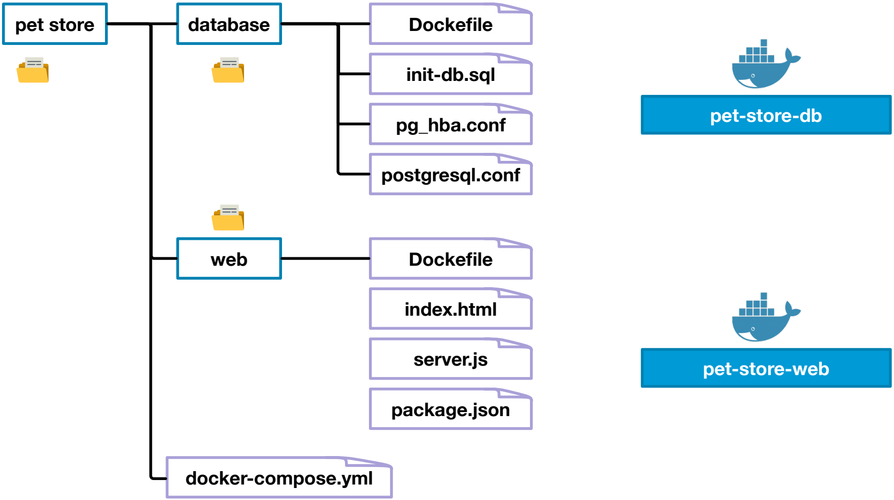
Example: simple pet store

The **simple pet store** is an application consisting of two services: ▸ Credit

- **web**. Web application consisting of HTML and Node.js code.
- **db**. A **database management system** (DBMS) that manages the application data (i.e., pet photos).

- **Docker compose** is a tool provided by Docker for building and running **multi-service applications** on a **single host**.
- The application is described in a YAML file, usually named *docker-compose.yml* (key-value pairs).
  - **Declarative** way of building and describing an application.

Pet store example: file hierarchy



Pet store example: docker-compose.yml

```yaml
version: "3.6"
```

Definition of the services

```yaml
services:
  web:
    build: web                    where to find the Dockerfile
    image: pet-store-web          name of the image
    networks:
      - backend                   networks of the service
    ports:
      - 5000:3000                 ports of the service
  db:
    build: database
    image: pet-store-db
    networks:
      - backend
    volumes:
      - pets-data:/var/lib/postgresql/data
```

Definition of the volumes

```yaml
volumes:
  pets-data:                      name of the volume
```

Definition of the networks

```yaml
networks:
  backend:                        name of the network
```

Building an application with Docker compose

- Run the following command in the directory **pet-store**.

```
docker-compose build
```

- A Docker image is created for each service for which the key **build** is specified.
- Value of the key **build**: the directory where the **Dockerfile** is.
- Value of the key **image**: name of the output Docker image.

Deploying an application with Docker compose

- Run the following command in the directory **pet-store**.

```
docker-compose up
```

- All networks defined in the section **networks** are created.
- All volumes defined in the section **volumes** are created **only if they don't exist yet**.
- A container is created and run **for each service**.
- The application is available at http://localhost:5000

Stopping an application with Docker compose

- Run the following command in the directory **pet-store**.

```
docker-compose down
```

- The containers associated to each service are **stopped** and **removed**.
- All networks defined in the section **networks** are removed.
- Volumes are **not removed**.
  - If we want to restart the application, we want the data to be still there.

Scaling a service with Docker compose

- Run the following command in the directory **pet-store**.

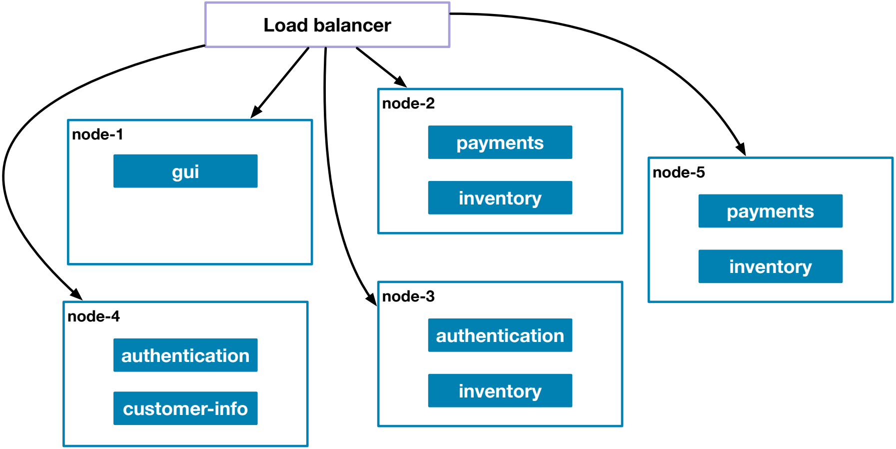
```
docker-compose up –scale web=3
```

- The command creates three containers for the service web.
- The file *docker-compose.yml* must be changed to specify a **range** of port numbers in the host.

```
version: "3.6"


services:
  web:
    build: web
    image: pet-store-web
    networks:
      - backend
    ports:
      - 5000-5005:3000
```

Multi-service applications across multiple hosts



Credit

Terminology

- **Node**. Individual (physical or virtual) host used to run one or more service instances.
- **Cluster**. Group of nodes connected by a network.
- **Network**. Physical and virtual communication paths used to connect nodes in a cluster.
- **Port**. Channel on which a service instance listens for incoming requests. ▸ Source

Definition (Distributed containerized application)

We define a **distributed containerized application** as a **multi-service application** such that each service has one to several running instances, each being a container, deployed across multiple nodes of a cluster.

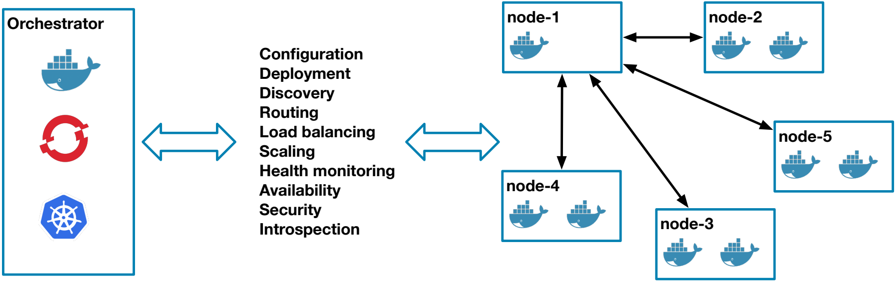
Challenges

- **Locate** services in the cluster.
- **Routing** messages from one service instance to another.
- **Balance** the load across all service instances.
- **Scale** the workload based on the number of requests.
- **Monitor** the health state of the service instances.
- **Ensure the security** of the application.

Definition (Orchestrator)

An **orchestrator** is a tool that handles the challenges of managing a distributed containerized application.

Tasks of an orchestrator



▸ Inspired to

Configuration and deployment

- **Declarative configuration** of the application.
  - Similar to Docker compose.
  - Images to run, number of instances, ports to open...

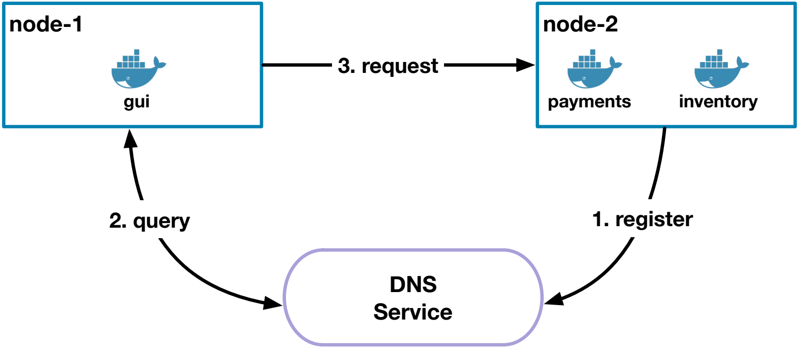
Definition (Desired state)

The set of properties of an application specified in the declarative configuration is called the **desired state** of the application.

- The orchestrator **deploys** the application while complying with the **desired state**.
- The orchestrator **corrects** any deviation from the desired state.

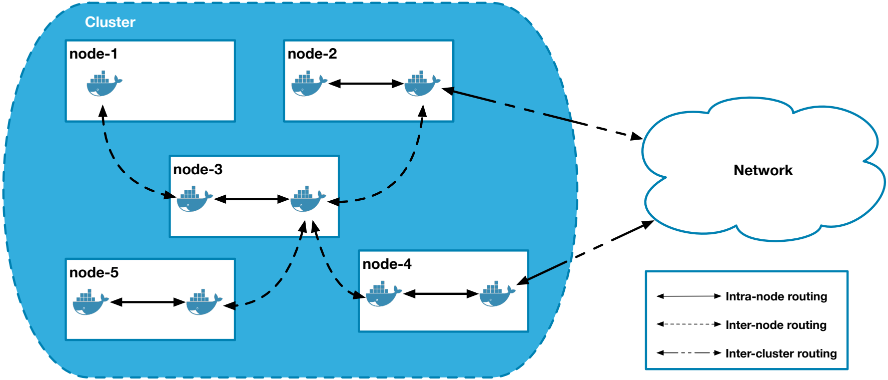
Service discovery

- Services usually don't have a fixed IP address or port number.
- They may be moved from one node to another.
- Service discovery: **environment variables** or (better) **DNS service**.



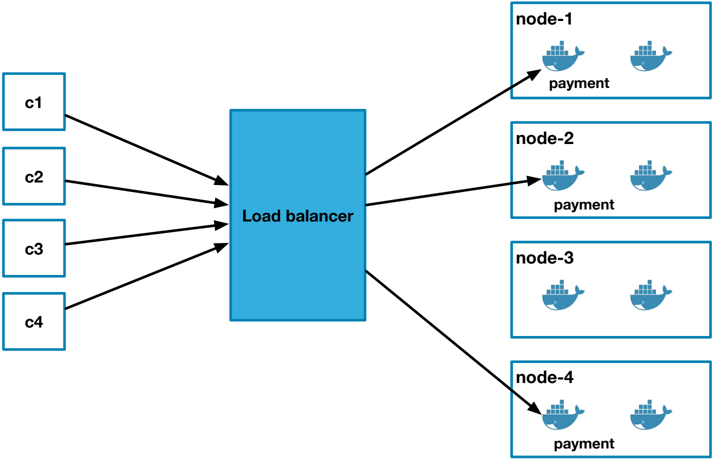
Routing

- **Move** messages from one service instance to another.
- Intra-node, inter-node, inter-cluster routing.



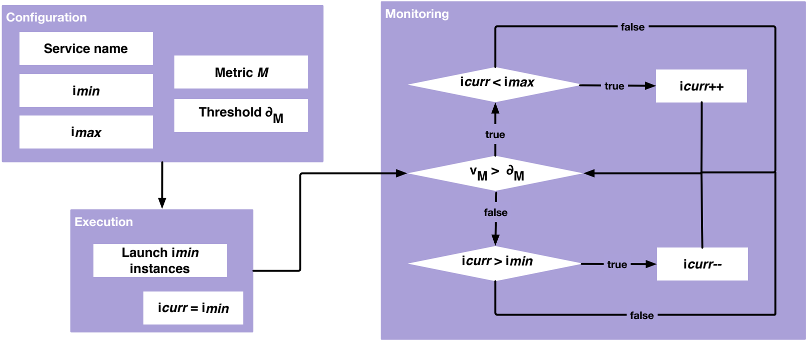
Load balancing

- Requests are **equally distributed** to all instances of a service.



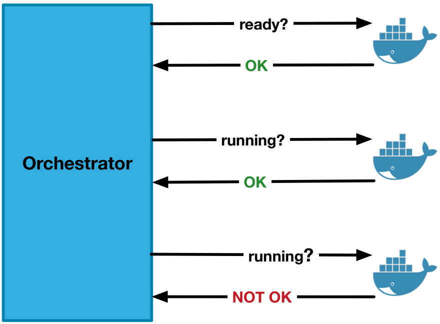
Scaling

- Set min (i_{min}) and max (i_{max}) number of instances of a service.
- Set a metric M (e.g., CPU utilization) and a threshold value δ_M.
- Launch $i_{curr} = i_{min}$ instances of the service.
- If the metric current value v_M exceeds δ_M, increment (i_{curr}) up to i_{max}.
- If v_M is lower than δ_M, decrement (i_{curr}) down to i_{min}.



Health monitoring

- The orchestrator executes **liveliness** and **readiness probes**.
- Instances that are not ready won't get any workload.
- Instances that are not running will be restarted.
- The orchestrator guarantees a **self-healing system**.



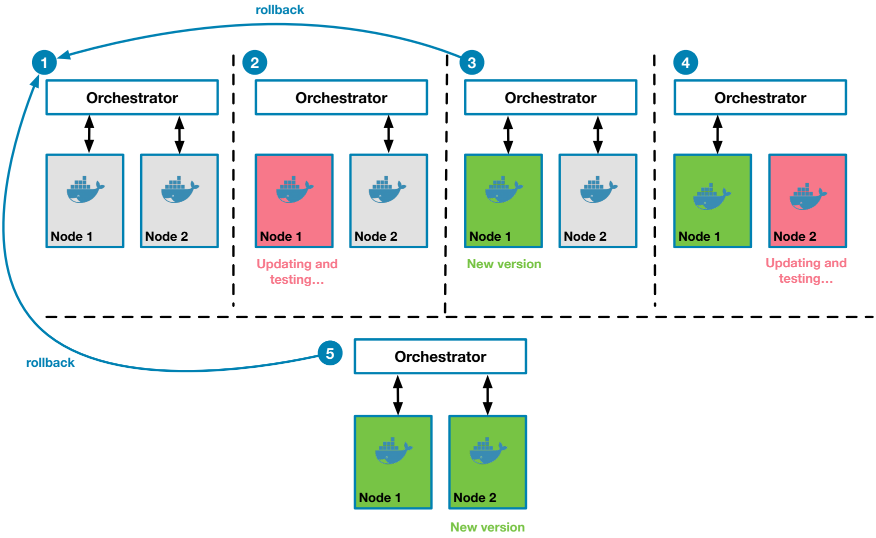
Availability

- Ideally, applications should be **available** 24/7.
  - Think of e-commerce websites.
- What about **maintenance**?
  - Containers can be scheduled on another machine.

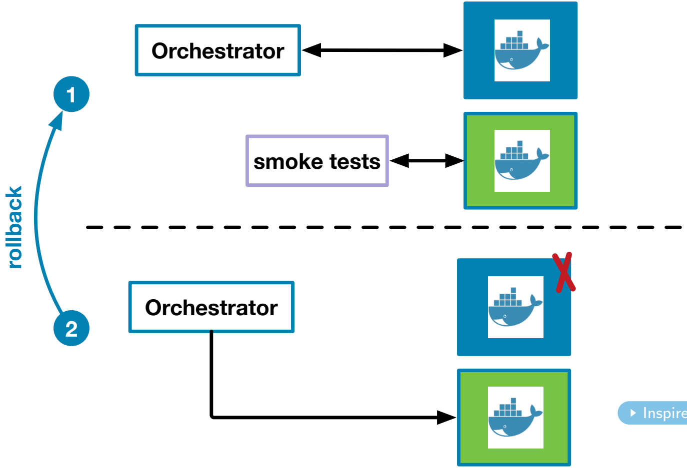
What about updates?

- **Solution 1.** Take down the application during the update.
  - Not acceptable if availability is critical.
- **Solution 2. Zero downtime** deployment.
  - Update and keep the application running.
- Different approaches: **rolling updates**, **blue-green deployments**, **canary releases**.

Rolling updates

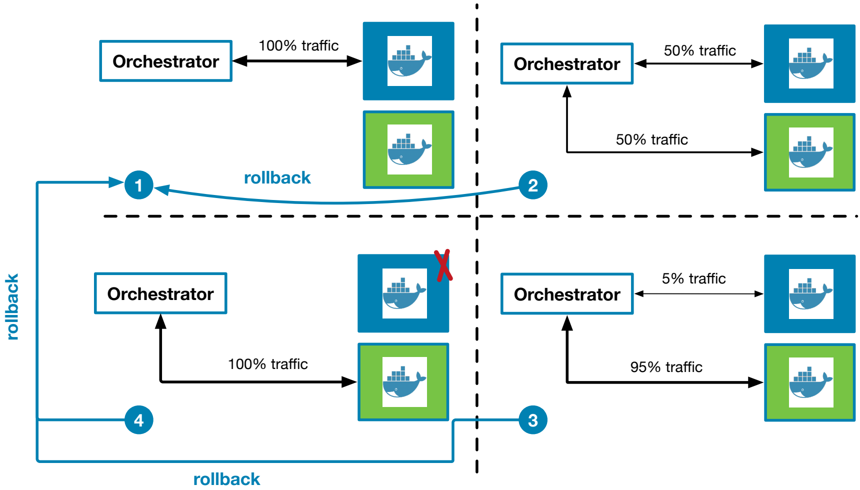


Blue-green deployments



Inspired to

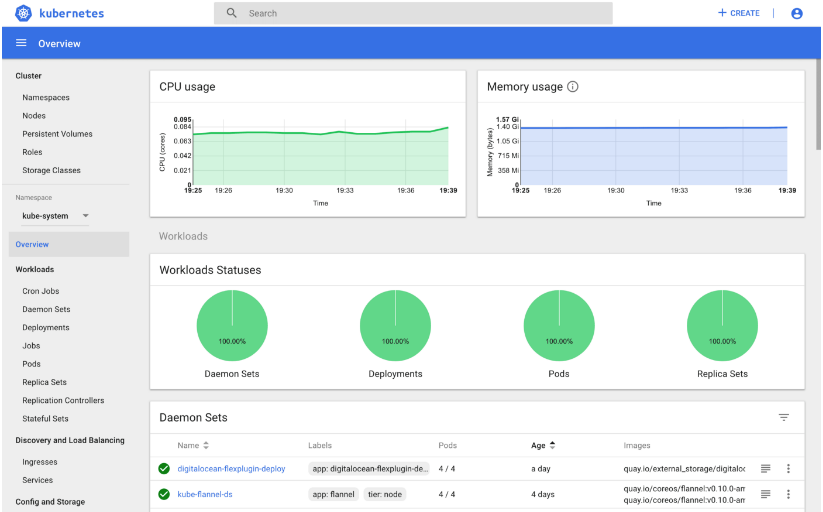
Canary releases



Security

- **Identity.** Each node has a cryptographic node identity.
- **Authentication.** Nodes authenticate with each other with certificates.
  - Mutual transport layer security.
- **Sandboxing.** Use of **software defined networks** to group services that need to communicate.
  - Avoid to attach all services to the same network.
- **Role-base access control (RBAC).** Access to the cluster resources depend on the **role**.
- **Secrets.** Objects containing small (encrypted) amounts of sensitive information.
  - Example: a password, a token to access an API...
- **Reverse uptime.** Limit the lifespan of a node.
  - Limit the duration of a potential attack.

Introspection



Popular orchestrators

- **Kubernetes.**
  - Modeled after Google Borg, designed for massive scalability.
  - Provides a complete set of features.
  - Difficult to configure.
- **Docker Swarm.**
  - Orchestrator provided by Docker.
  - Less complete than Kubernetes.
  - But way easier to configure.
- **Amazon Elastic Container Service (ECS)**
  - Integrated into the Amazon AWS ecosystem.
  - Less complete than Kubernetes and Docker Swarm.
  - Only available on Amazon AWS (Cloud lock-in).

Kubernetes

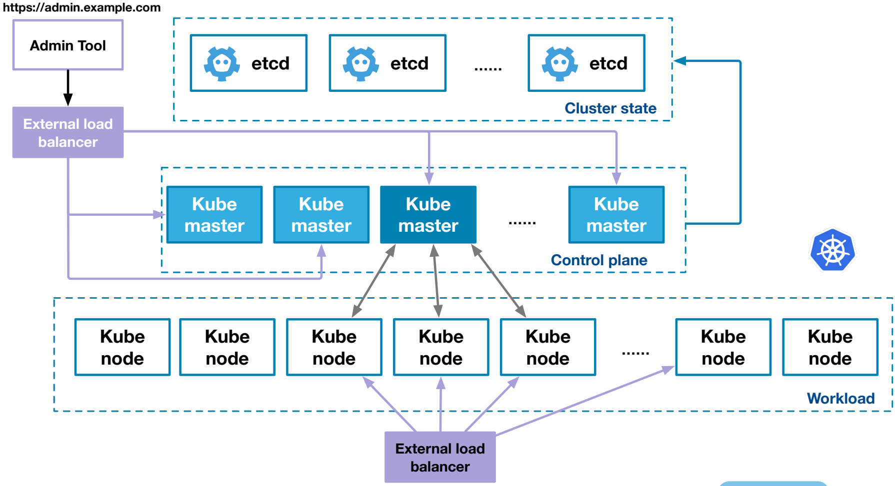
- **2003-2004.** Google introduced **Borg**, a large-scale internal cluster management system.
- **2014. Kubernetes** introduced as an open source version of Borg.
- **2015.** First Kubernetes community conference.
- **07/2016.** Release of **Minikube**, a tool to run Kubernetes locally.
- **10/2016.** Release of **Pokemon Go**, the largest Kubernetes deployment on Google container engine.
- **08/2017. Github** web and API requests are served by containers orchestrated by Kubernetes.
- **10/2017.** Docker embraces Kubernetes.

> Were it not for Docker's shifting of the cloud developer's perspective, Kubernetes simply would not exist.
>
> — Brendan Burns, ▸ Reference

Terminology

- Kubernetes: used to **orchestrate** a multi-service containerized application running in a cluster.
- Each node in the cluster has one of two roles: **master** or **worker**.
- **Master nodes.** They manage the cluster.
  - Small and odd number of masters.
- **(Worker) nodes.** They run the containerized application.
  - As many worker nodes as needed.
- Nodes are connected by a physical network (**underlay network**).

High-level architecture

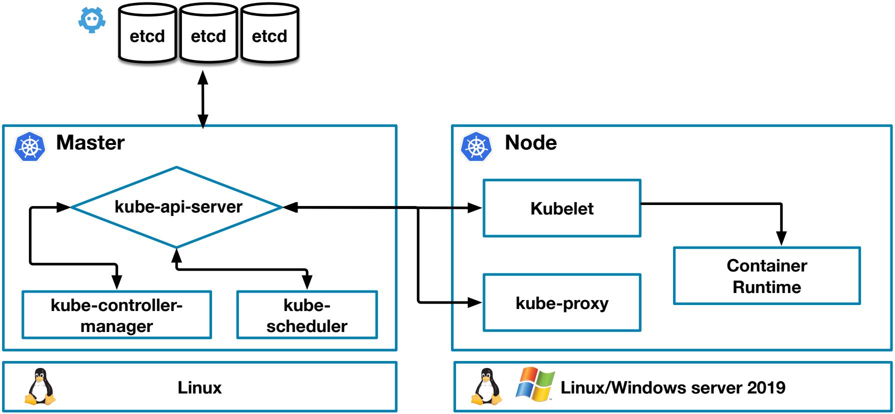


Inspired to

High-level architecture

- **etcd nodes.** etcd is a **distributed** key-value database that stores the **state** of the cluster.
  - Type of services and running instances.
  - Network settings.
  - Secrets.
  - The state **doesn't** include data produced/consumed by the application.
- **Master nodes.** They manage the cluster.
  - Check the consistency of the cluster **actual state** with the **desired state**.
- **Worker nodes.** They execute the application workload.
- **Load balancer.** Also called **reverse proxy**, its role is to route the external traffic to the appropriate service.

Kubernetes nodes



Master components

- **kube-apiserver.** REST interface to list, create, modify or delete resources in the cluster.
    - Scales horizontally (multiple running instances).
- **kube-controller-manager.** Reconcile the actual state with the desired state.
    - **Node controller.** Notices and responds when nodes go down.
    - **Replication controller.** Maintains the correct number of service instances.
- **kube-scheduler.** Assigns newly created **pods** (i.e., groups of containers) to a node so they can be executed.
    - hardware/software constraints, data locality, affinity specifications.

Master nodes run on Linux only.

Node components

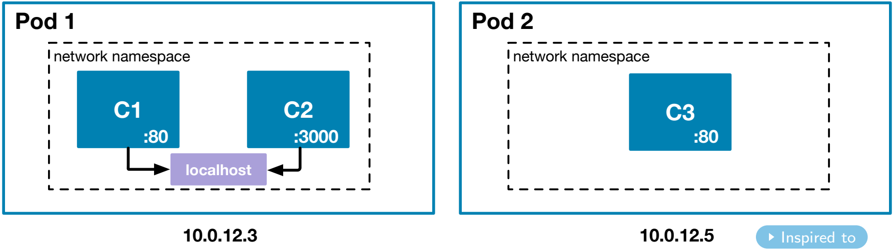
- **kubelet.** Makes sure that the containers in a pod are running according to the specifications.
- **kube-proxy.** A network proxy that allows network communication between containers.
- **Container runtime**. Software responsible for running containers.
  - Kubernetes supports Docker, containerd, CRI-O and any implementation of the Kubernetes CRI (Container Runtime Interface).
  - By default, Kubernetes uses the CRI-Docker integration.

Pods

Definition (Pod)

A **pod** is an abstraction of many co-located containers that share the same Kernel namespaces.

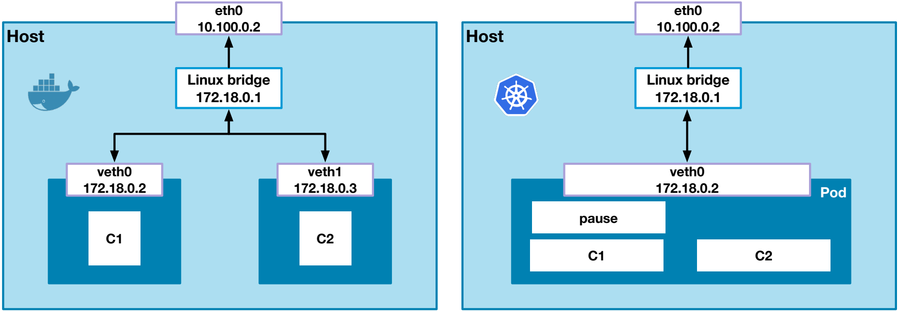
- Each pod gets an IP address (unique across the cluster).
- Two containers in the same pod must use **different port numbers**.
- Two containers in the same pod can communicate through **localhost**.



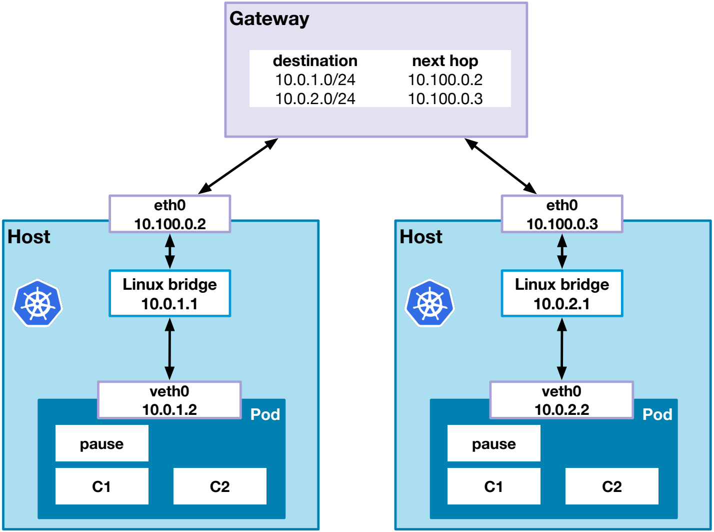
Inspired to

Kubernetes networking model

- In Docker, every container has its own network namespace.
- When a pod is created, Kubernetes creates a container called **pause**.
- **pause** creates and manages the namespaces shared by all the containers in the pod.
- The other containers: created with the option **–net container:pause**
  - They'll share the network namespace of the container **pause**.



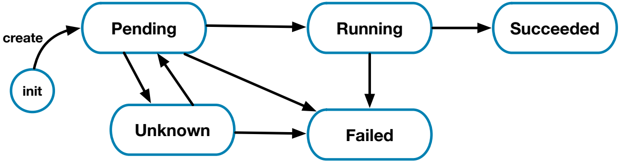
Kubernetes networking model



Source

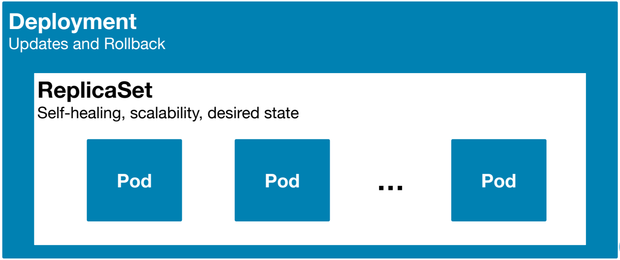
Pod life cycle

- **Pending.** The pod is accepted but one or more container has not been created (scheduling and image download).
- **Running.** The pod is assigned to a node and all containers have been created. At least one container is running or is about to (re)start.
- **Succeeded.** All containers have succesfully terminated and will not be restarted.
- **Failed.** All containers have terminated but at least one with errors (non-zero status), or has been terminated by the system.
- **Unknown.** The pod state cannot be obtained due to a communication error.



Controllers

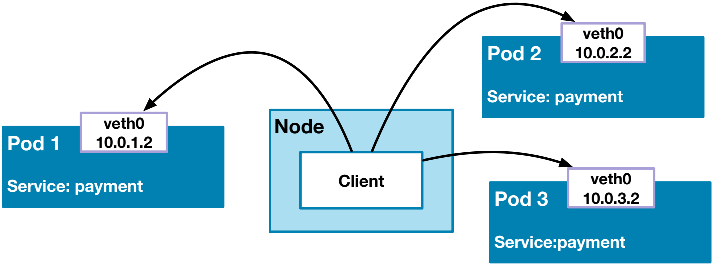
- Pods don't handle the following events:
  - Failure of the node where the pod is running.
  - Eviction of the pod for node maintenance or lack of resources.
  - Failure in scheduling.
- Kubernetes provides **controllers** to create/manage multiple pods.
  - **ReplicaSet**. Handles a collection of identical pods
  - **Deployment.** Augments a ReplicaSet by providing rolling updates and rollbacks.



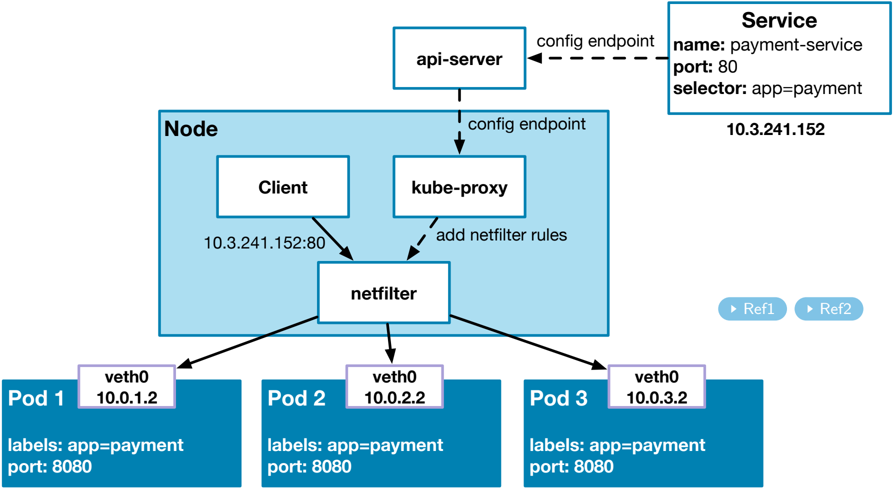
Source

Services

- Pods are associated with an IP address.
- It's possible to send a pod a request by using its IP address.
- However, pods are **ephemeral**.
  - They might need to be stopped.
  - When they restart, they are assigned a different IP address.
- **Service discovery.** How a client (container/pod) can send a pod a request?

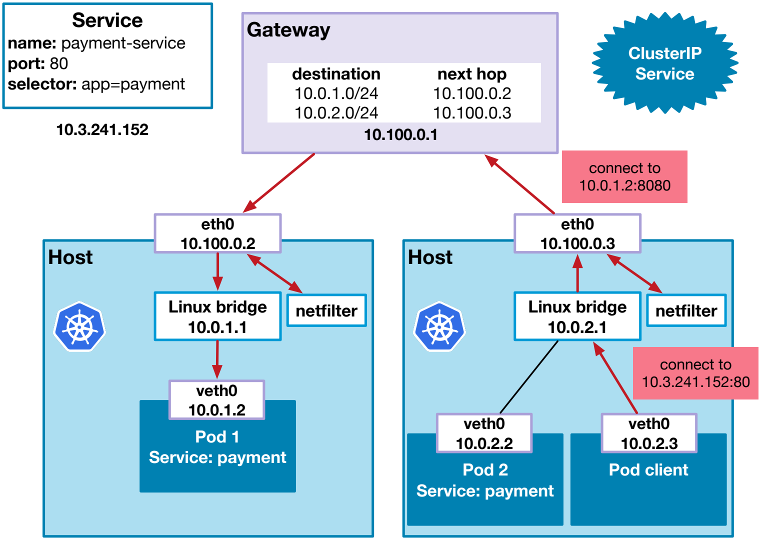


Services



Ref1 Ref2

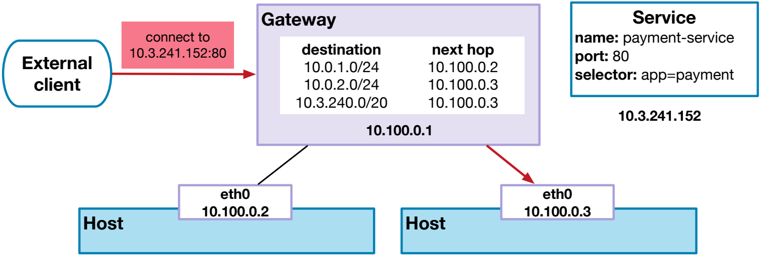
Internal traffic: ClusterIP services



Source

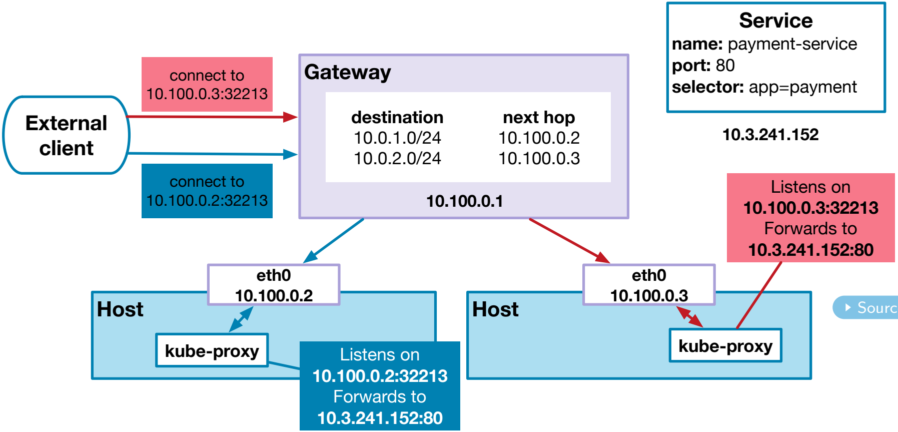
External traffic

- ClusterIP services can be accessed **from within the cluster**.
- Not a good solution to route external traffic to a ClusterIp service.
- The traffic must be redirected explicitly to either node providing the target service.
- But nodes are ephemeral.



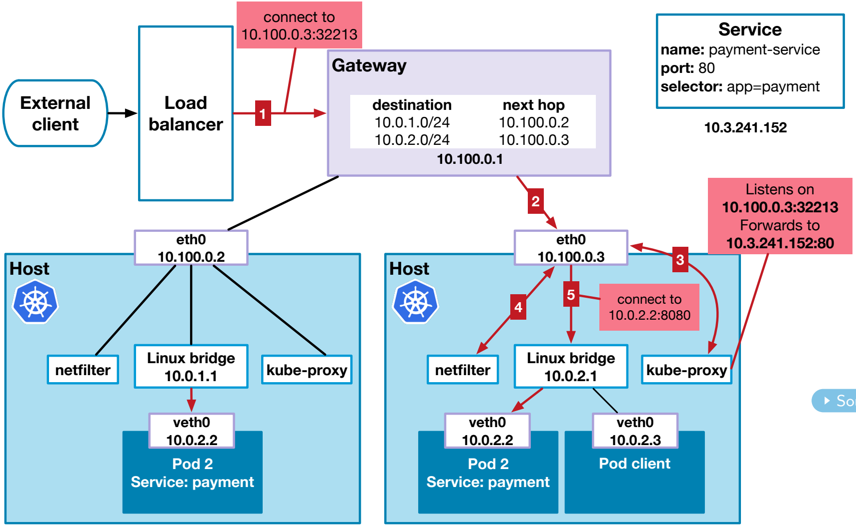
External traffic: NodePort services

- Opens a port in the range [30000–32767] **on each node**.
- This port is used to forward traffic to the service.



Source

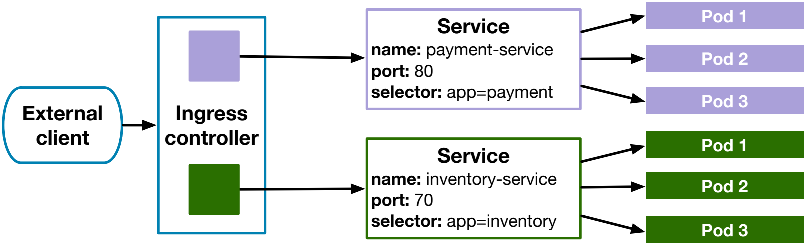
External traffic: LoadBalancer services



Source

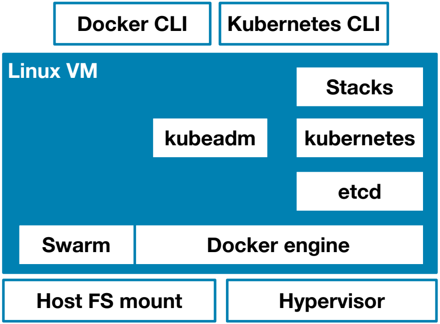
External traffic: Ingress controllers

- A LoadBalancer service cannot proxy multiple services.
- Each service gets its own load balancer (and IP address).
- A cloud provider may bill based on the number of load balancers.
- An **ingress** is a resource independent of the services.
  - Specifies how to route traffic to services.
- An **ingress controller** enforces the specifications.
- Handles multiple services with a unique IP address.



Integration of Docker with Kubernetes

- The latest version of Docker Desktop support Kubernetes.
  - Both MacOS and Windows editions.
- All Kubernetes components run in containers in the Linux VM.



- If you installed Docker Toolbox, you can still use Kubernetes by installing **Minikube**. Link

Deploying the pet store with Kubernetes

- The pet store consists of two components:
  - **web.** The Web interface of the store.
  - **db.** The backend database of the store.
- We need to define both components in Kubernetes.
- We use a **declarative approach** to define the components.
  - Description in a YAML file.
- For each component, we need to define:
  - A **deployment** object.
  - A Kubernetes **service** to expose the component.

The **web** deployment

```yaml
apiVersion: apps/v1
kind: Deployment          we define a deployment
metadata:
 name: web                name of the deployment object
spec:
 replicas: 5              number of replicas
 selector:
  matchLabels:
   app: pets              labels that identify the
   service: web           pods composing this deployment
 template:
  metadata:
   labels:
    app: pets                            Template section: specify the
    service: web                         containers with their parameters
  spec:
   containers:
   - image: quercinigia/pet-store-web:1.0
    name: web
    ports:
    - containerPort: 3000
     protocol: TCP
```

The **web** service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  type: NodePort
  ports:
  - port: 3000
    protocol: TCP
  selector:
    app: pets
    service: web
```

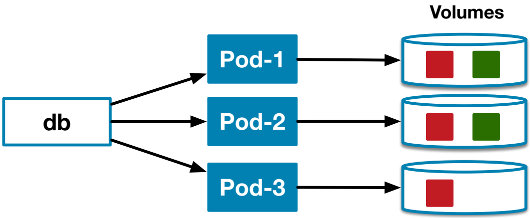
Service of type NodePort

Port to expose

Pods that compose the service

The **db** StatefulSet

- The **web** component is **stateless**.
  - It doesn't create or modify any persistent data.
- The **db** component is **stateful**.
  - By definition, a database creates/modifies persistent data.
- Each pod has its own **state**.
  - **Identity** matters.
  - **Ordering** might matter too.
- A **deployment** object is not suitable for stateful components.
- We can use **StatefulSets**.



The **db** StatefulSet

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
 name: db
spec:
 selector:
  matchLabels:
   app: pets
   service: db
 serviceName: db
 template:
  metadata:
   labels:
    app: pets
    service: db
```

This name is used by kube-dns

```yaml
 spec:
  containers:
  - image: quercinigia/pet-store-db:1.0
   name: db
   ports:
   - containerPort: 5432
   volumeMounts:
   - mountPath: /var/lib/postgresql/data
    name: pets-data
volumeClaimTemplates:
- metadata:
  name: pets-data
 spec:
  accessModes:
  - ReadWriteOnce
  resources:
   requests:
    storage: 100Mi
```

Mounting the volume pets-data

Declaring the volume pets-data

The volume R/W can be mounted by 1 node

Max storage: 100MB

The **db** service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: db
spec:
  type: ClusterIP
  ports:
  - port: 5432
    protocol: TCP
  selector:
    app: pets
    service: db
```

Service of type ClusterIP (backend)

Port to expose

Pods that compose the service

Deploying the application

- The four YAML definitions are stored in a file **pets.yaml**.
- In the directory of file **pets.yaml** type:

```
kubectl create -f pets.yaml
```

- To see the **status** of the **pods** type:

```
kubectl get pods
```

- To see the **status** of the **services** type:

```
kubectl get services
```

Accessing the application

- By typing the command **kubectl get service web**, you should get something like:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
web	NodePort	10.102.169.204	<none>	3000:30872/TCP	3m

- The web component is **exposed** on port 30872 (your port number might be different).
- Open the browser and type the following URL:
  - If you use **Docker Desktop**: http://localhost:30872/pet
  - If you use **Minikube**: http://minikube-ip:30872/pet, where *minikube-ip* is the IP address of minikube
  - To get the IP address of minikube, type the command **minikube ip** in the terminal.

Rolling updates

- We set the new image (newer version) of the **web** component.

```
kubectl set image deployment/web \
    web=quercinigia/pet-store-web:2.0
```

- The command starts updating all instances.
- To see the status of the update, type:

```
kubectl rollout status deploy/web
```

- To see how the new version has been rolled out, type:

```
kubectl describe deploy/web
```

Rolling back an update

- If we notice that the new version doesn't behave correctly, we can rollback.

```
kubectl rollout undo deploy/web
```

Taking down the application

.

- Delete the service **web**.

```
kubectl delete svc/web
```

- Delete the Deployment **web**.

```
kubectl delete deploy/web
```

- Delete the service **db**.

```
kubectl delete svc/db
```

- Delete the StatefulSet **db**.

```
kubectl delete statefulset/db
```

Blue-green deployment

Web deployment "Blue"

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: web-blue
spec:
 replicas: 5
 selector:
  matchLabels:
   app: pets
   service: web
   color: blue
 template:
  metadata:
   labels:
    app: pets
    service: web
    color: blue
  spec:
   containers:
   - image: quercinigia/pet-store-web:1.0
    name: web
    ports:
    - containerPort: 3000
     protocol: TCP
```

Web deployment "Green"

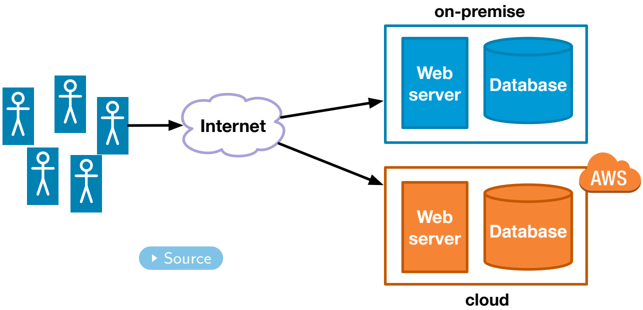
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: web-green
spec:
 replicas: 5
 selector:
  matchLabels:
   app: pets
   service: web
   color: green
 template:
  metadata:
   labels:
    app: pets
    service: web
    color: green
  spec:
   containers:
   - image: quercinigia/pet-store-web:2.0
    name: web
    ports:
    - containerPort: 3000
     protocol: TCP
```

Web service

```yaml
apiVersion: v1
kind: Service
metadata:
 name: web
spec:
 type: NodePort
 ports:
 - port: 3000
  protocol: TCP
 selector:
  app: pets
  service: web
  color: blue
```

Multi-service applications in the Cloud

- A successful pet store needs a **robust infrastructure**. Two options.
- **"On-premise".** The pet store owner buys and maintains the infrastructure.
- **"Cloud".** The pet store owner uses a (public) cloud infrastructure.
- The cloud solution offers numerous advantages.
  - maintenance-free services, virtual servers, load balancing, DNS...

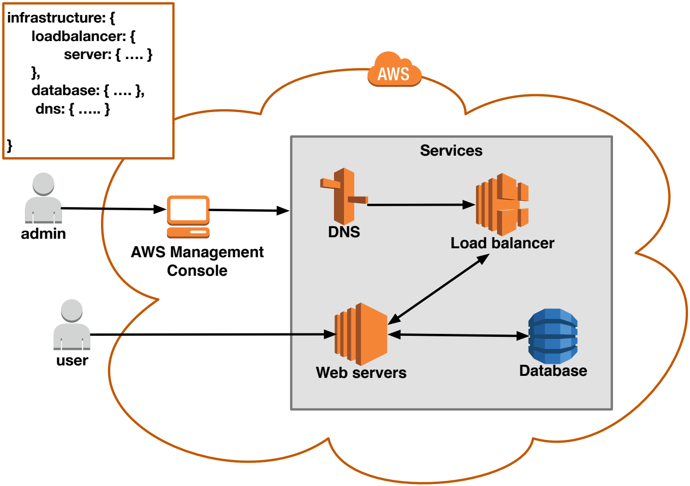


Source

Amazon Web Services (AWS)

- "Platform of Web services with solutions for computing, storing, and networking, at different layers of abstraction."" Source
- AWS services are offered to **solve common problems** when deploying an application.
  - load balancing, storage, scalability, reliability.
- Services are billed based on **usage**.
  - number of load balancers, virtual server uptime...
- Planning costs: Click here

Managing services in AWS



Containerized applications in AWS

- AWS provides the **Elastic Container Service (ECS)**.
  - "fully managed container orchestration service" Source
- ECS is built on a cluster of servers.
  - **Option 1.** Manual creation of the cluster.
  - **Option 2.** Using Fargate.
- ECS can run Docker containers.
- Custom orchestration.
- Possibility of running Kubernetes clusters with **Elastic Kubernetes Service (EKS)**.