University of York
Europe Campus
CITY College

Kubernetes

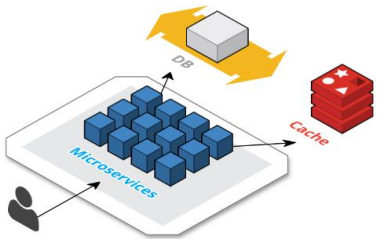
CCS3341 Cloud Computing

Dr S. Veloudis

Overview

Kubernetes (K8s) is an open-source system for automating **discovery**, **deployment** and **management** of containerised applications

Motivation?



Multi-container apps comprise dozen of independent containerized services each typically deployed across pools of VMs running on datacentre infrastructure

Resource efficiency

Dynamically deciding how to optimally place containers across a pool of VMs (a cluster) such that **QoS constraints** attached to these containers are met in the face of fluctuating workloads cannot be a manual task

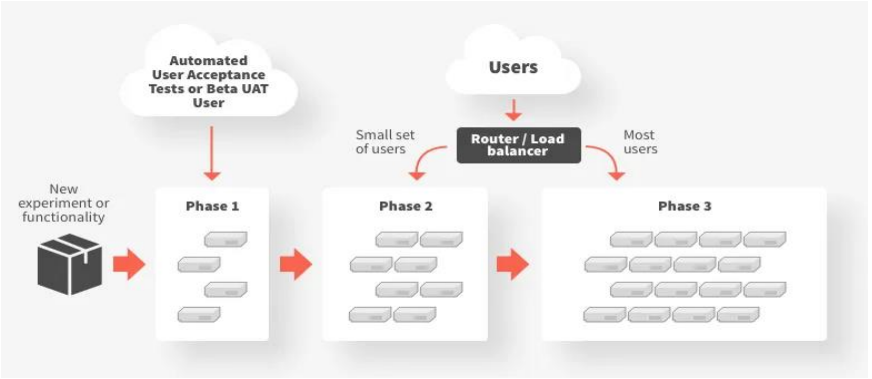
Fluctuating workload trigger dynamic container placement and orchestration

Qos constraints:

- CPU cores
- RAM
- Taints and affinities

Dynamic Scaling/Updating

- Rapid scaling in the face of **fluctuating demand** cannot be achieved manually
- Deployment velocity of **rolling updates** in **Canary Deployments** cannot be achieved manually



Canary deployments – minimize risk of new SW versions by gradually scaling down old versions and scaling up new versions

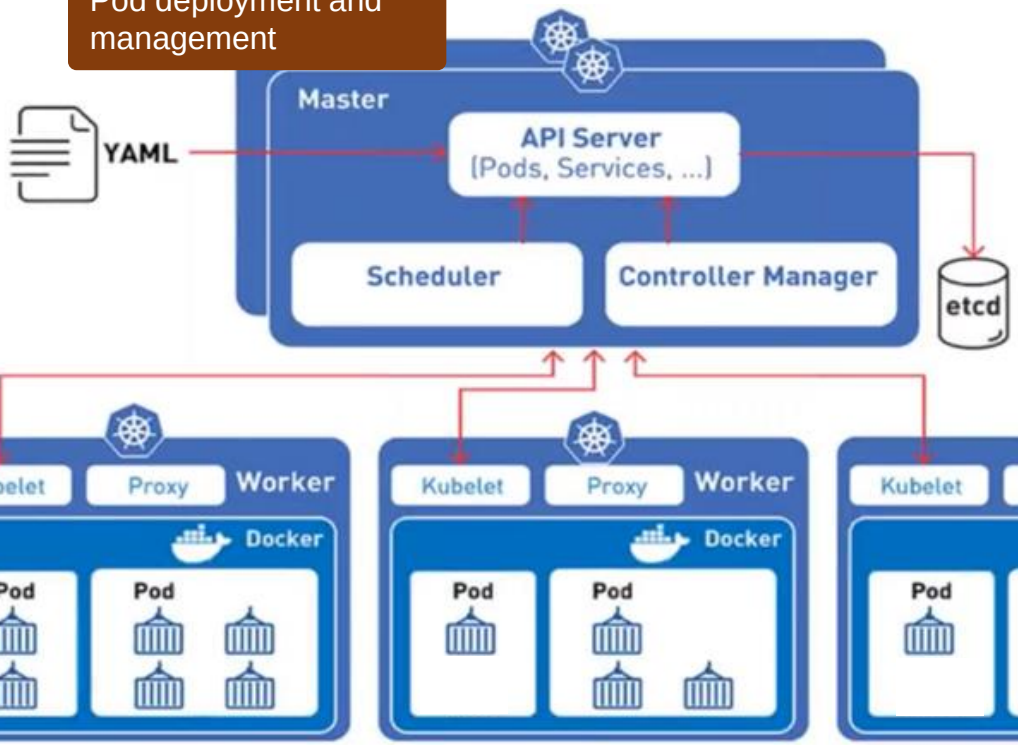
Reliability

Dynamically monitoring the state of each container and taking self-healing actions in case of **failures** (e.g. migrating to different nodes) cannot be a manual task

Failures occur when deviations occur from a **prescribed 'ideal' state**

Cluster

Pod deployment and management



Key-value store for cluster – includes:

- Cluster configuration – network topology, API server endpoints, authentication/authorization settings
- API objects
- Services running in the cluster (IP addresses, port numbers, etc.)

Nodes or Compute Instances

In GCloud, a worker node is a VM on Google's hardware infrastructure preloaded with a lightweight Linux-based OS

Kubelet

Manages individual nodes in the cluster

- Starts/stops containers in pods
- Registers a node with the Master and provides information about its resources and capacity; it then monitors these resources
- Checks container health within a pod using readiness and liveness probes
- Attempts to restart failed pods

Workloads & Pods

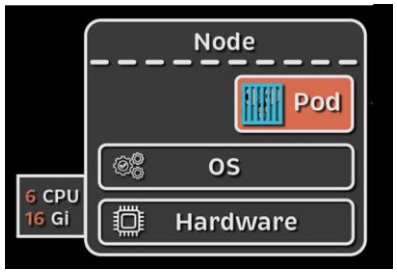
A **workload** in K8s refers to an application running on a K8s cluster

i.e., on a pool of VMs running on typically shared infrastructure

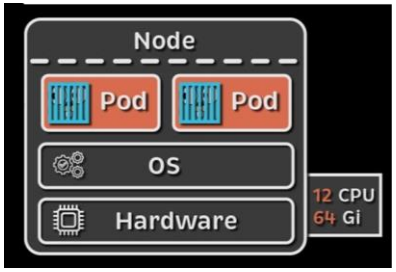
- A workload may consist of one or more **containerised** apps
- A workload always executes within a **pod**

Pod

- Group of one or more containers
- Smallest deployment unit in K8s



Typical scenario (one container/pod)



A pod may comprise several containers

- A **pod** is a conceptual abstraction
- It is also a **mechanism**: it provides the necessary runtime env for containers to execute

Container Runtime Interface (CRI)

Standardised API through which Kubelets interact with containers running in pods

Support for:
- Docker
- Containerd
- CRI-O
- …

- Pod creation
- Starting/stopping/removing containers from pods
- Image pulling from container registries
- Assignment of IP addresses
- Mounting volumes
- Enforcing resource limits (cgroups)
- Logging and monitoring

Pods

A pod's contents are always co-located and co-scheduled and run in **a shared context**

Shared context

- Containers in pods are tightly-coupled as they share the same net and mnt namespaces
- They may also share the same pid namespace

- Practically this means that containers in a pod share the same internal (private) IP address and have access to the same filesystems or block storage
- It also means that pods cannot be deployed across nodes

Single-container pods

- Commonest use case
- A pod is a wrapper around a container; Kubernetes manages the pods rather than the container directly

Pods are designed to be **ephemeral** and can be created, destroyed, and replicated dynamically as needed

Operations such as pod relocation (e.g., for meeting QoS requirements), as well as pod scaling and updates, entail frequent pod deletions and recreations…

Workload Resources

A **workload resource** is an abstraction that represents an application running on the cluster

- Defines the desired state for one or more **pods**
- Manages the lifecycle of these pods to ensure that the application runs as specified

Key characteristics

- Declarative configuration
  Workload resources are expressed in YAML and are orthogonal to any implementations
- Controller management
  A controller is associated with each workload resource which continuously operates to ensure that the cluster's state matches that of the specified in the workload resource
- Pod lifecycle automation
  The workload resource automates the creation, scheduling, scaling, and termination of pods
- Fault tolerance
  Creates replacement pods for failed pods running on failed nodes

Main types

Deployment
- Used for stateless apps (e.g., web servers RESTful interfaces, microservices that use external persistence for maintaining state)
- Manages replicated pods and supports rolling updates

StatefulSet
- Used for stateful apps (e.g., databases or containers that are associated with disk volumes)
- Ensures that pods have stable network ids and persistent storage

Job
- Used for finite computations (e.g., data processing or migration tasks)
- Runs a batch process to completion, managing pods required to complete the task

Workload resources are used by the **kube-controller-manager** to define and manage the Deployment, Scaling, Update, and Operation of pods within a cluster

Workload Resources → Deployments

Deployments specify

- Pod labels
- Number of replicas for a pod (fault tolerance)
- QoS requirements on pods
- Affinities and taints
- Rolling updates
- Monitoring

Example

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
  labels:
    app: example-app
spec:
  replicas: 3  # Desired number of pod replicas
  selector:
    matchLabels:
      app: example-app
  template:
    metadata:
      labels:
        app: example-app
```

Deployment object name

Deployment object label

Identify which pods belong to the Deployment

Label attached to pods created by the Deployment

Labels

- Logical names used to identify pods irrespective of their network location
- Labels don't change (as opposed to network locations)
- A pod may be assigned multiple labels
- Labels may also be applied to configuration objects and nodes

Such labels are used to group configuration objects and facilitate their management

Names

- Apply to both pods and configuration objects
- Used internally by K8s (e.g., when scheduling or relocating pods)
- Unique identifiers

DNS names

See later…

Workload Resources → Deployments

Example (contd.)

```yaml
spec:
  containers:
  - name: example-container
    image: nginx:1.21  # Container image and version
    ports:
    - containerPort: 80  # Expose port 80
    resources:  # Resource requests and limits
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

QoS

```yaml
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000
globalDefault: false
description: "Priority class for high-priority workloads."
```

QoS classes

- Guaranteed
  When requests and limits are specified and are equal, K8s ensures that the pod gets these resources
- Burstable
  When requests and limits are specified and are not equal, the pod may burst to use all available resources but the pods may be evicted to be replaced by guaranteed ones
- BestEffort
  No resource limits or requests -> lowest priority pods that get resources only when available; may be evicted to be replaced by guaranteed and burstable pods

Custom priority classes may be defined to define priorities within the same QoS class

Workload Resources → Deployments

Setting custom priorities…

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: high-priority-pod
spec:
```

⋮

```yaml
  spec:
    priorityClassName: high-priority
    containers:
    - name: app-container
      image: nginx
      resources:
        requests:
          cpu: "2"
          memory: "2Gi"
        limits:
          cpu: "2"
          memory: "2Gi"
```

Guaranteed but with high priority

Affinities/anti-affinities

Node selection based on affinities and taints

```yaml
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: "example.com/role"
          operator: In
          values:
          - "worker"
tolerations:
- key: "example.com/environment"
  operator: "Equal"
  value: "production"
  effect: "NoSchedule"
```

Location-related affinity: ensures that pods are scheduled on nodes labelled “worker”

Taint spec -> defines which taints on nodes may be ignored (tolerated)

- Taints are placed on nodes
- Taints are distinguished by key-value pairs
- Taints also have effects (NoSchedule, PreferNoSchedule, NoExecute)
- In the above, pods can tolerate NoSchedule taints on nodes that match the provided spec

```
kubectl taint nodes <node-name> key=value:NoSchedule
```

Workload Resources → Deployments

Rolling updates

Example

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app-container
        image: my-app-image:v1
        ports:
        - containerPort: 8080
      imagePullSecrets:
      - name: my-app-registry-secret
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  minReadySeconds: 30
```

- Describes how a rolling update is to be performed: replace one old pod by a new one
- Defines the secret that needs to be provided for pulling the image from the repository at which it resides

- maxSurge: max number of additional replicas (over the replica limit) that can be created during a rolling update
- maxUnavailable: max number of replicas that can be unavailable during the update
- minReadySeconds: time after which an updated replica starts serving requests

Workload Resources → Deployments

Monitoring

Example

```
⋮
readinessProbe:  # Ensures traffic is only sent to healthy pods
  httpGet:
    path: /
    port: 80
  initialDelaySeconds: 5
  periodSeconds: 10
livenessProbe:  # Checks if the container is still running
  httpGet:
    path: /
    port: 80
  initialDelaySeconds: 10
  periodSeconds: 20
```

- Readiness check: ensure pods are ready to serve traffic
- Liveness check: detects pods are responsive

Workload Resources → StatefulSets

Recall:

StatefulSet

- Used for stateful apps (e.g., databases or containers that are associated with disk volumes)
- Ensures that pods have stable network ids and persistent storage

Deployments vs StatefulSets

Feature	Deployment	StatefulSet
Pod Identity	Interchangeable; no unique identity	Unique and stable identity (e.g., myapp-0)
Network Identity	Ephemeral IP and hostname	Stable DNS name for each Pod
Persistent Storage	Shared or ephemeral storage	Persistent volume for each Pod (via PVCs)
Creation/Deletion	Pods created/deleted simultaneously	Pods created/deleted in order
Scaling	All Pods are equal; easy scaling	Order-sensitive; scaling may need orchestration
Updates	Rolling updates with identical Pods	Updates maintain Pod order and identity
Use Case	Stateless workloads (e.g., web servers)	Stateful workloads (e.g., databases, queues)

Workload Resources → StatefulSets

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cassandra
  labels:
    app: cassandra
spec:
  serviceName: cassandra-headless # Headless service for stable DNS
  replicas: 3 # Number of Cassandra nodes
```

See later…

A stable identity is important for applications that require consistent network identifiers, or for workloads that rely on persistent storage.

Each pod is associated with its own stable DNS name…

⋮

```yaml
volumeClaimTemplates:
- metadata:
    name: cassandra-data
  spec:
    accessModes: ["ReadWriteOnce"]
    resources:
      requests:
        storage: 10Gi
```

- A Persistent Volume Claim (PVC) is created to request from K8s persistent storage such as network-attached storage or a block or file storage volumes
- A PVC defines access modes (ReadWriteOnce, ReadOnlyMany, ReadWriteMany)
- A PVC can be mounted to several pods; its lifecycle is independent of that of a pod

Each pod is associated with its own PVC, allowing data to be persisted across Pod restarts or rescheduling

PVCs vs DBaaS

Standalone PVC objects may be defined e.g.,

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

A PVC object may then be associated with Deployments

But is a PVC preferrable to a storage solution like DBaaS?

- Storage resources (volumes) allocated to a cluster through PVCs are managed by the developer/admin
- Storage resources may outlive the lifespan of pods

DBaaS

DBaaS solutions on the other hand provide managed storage services

- The developer/admin no longer interacts directly with storage resources
- Instead, interaction is achieved through an API that the DBaaS provides

DBaaS features:

- Automated backups and scaling
- Infrastructure management for high availability, fault tolerance, and security

Workload Resources → Jobs

Recall:

Job
- Used for finite computations (e.g., data processing or migration tasks)
- Runs a batch process to completion, managing pods required to complete the task

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: example-job
spec:
  template:
    spec:
      containers:
      - name: task-container
        image: busybox
        command: ["echo", "Hello from the job!"]
  completions: 1
  parallelism: 1
```

Job completed when 1 pod completes; just one pod running

Features:

- Parallel execution of multiple pods; each pod runs the same task independently of the other pods
- Automatic pod restart (self-healing) upon job failure

Services

A **Service** is a resource that provides a stable and abstracted way to access a set of Pods

It enables network communication between pods and external clients by defining policies to route traffic to the appropriate pods

Services are crucial in enabling communication between different components in a distributed application, as Pods themselves are ephemeral and may change IP addresses over time

ClusterIP Service

- Has its own stable virtual IP address
- This IP address **cannot** be used for external communication (can only be used inside the cluster)
- This IP address is used for communication inside the cluster

The **ClusterIP** acts as the single access point for all the pods targeted by that Service

When a pod in the cluster wants to send traffic to another pod in the cluster, it first sends to the ClusterIP service (using the Service's stable virtual IP address)

K8s then resolves the target pod labels in the service to current pod IP addresses

K8s then further resolves the target pod IP addresses to the corresponding current node IP addresses

The DNS name that K8s provides to a Deployment is resolved to the IP address of the ClusterIP service associated with the Deployment

Overlay network

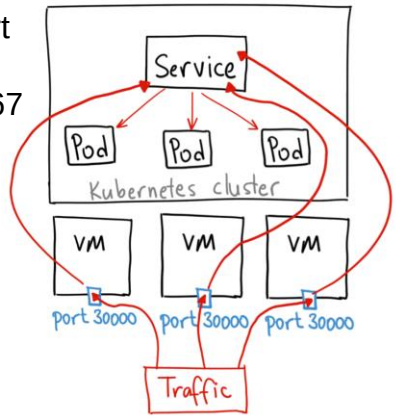
NodePorts and LoadBalancers

For communication outside a cluster, NodePort or LoadBalancer Services must be used

NodePort

Exposes a ClusterIP service to the external network by mapping a static port on each node in the cluster to the Cluster IP

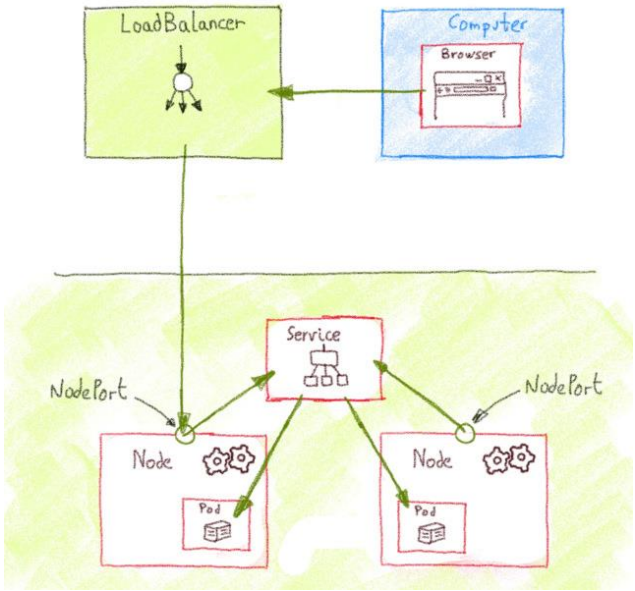
Allowable port numbers:
30000 – 32767



Client sends a request to the **external IP address** of any node in the cluster. This external IP could be the public IP of the node if it's an external client, or a private IP if it's within the same network

LoadBalancer

Operates in the same way as NodePort but now an external load balancer is used



Headless

- Assigns a DNS name to each pod which resolves to the pod's private IP address
- Used in StatefulSets

Descriptors

Services are described through YAML descriptors

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    - name: http
      port: 80
      targetPort: 8080
      nodePort: 30001
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: LoadBalancer
  selector:
    app: my-app
  ports:
    - name: http
      port: 80
      targetPort: 8080
```

kube-proxy

- Implements the K8s service abstraction
- Opens a port on each node in the cluster, and forwards any traffic sent to that port to the backend pods
- Implements Iptables
- Watches EndpointSlices for any changes in endpoints

EndpointSlice

API resource that provides an efficient and scalable way to manage the IP addresses and ports of pods

- Each EndpointSlice contains a subset, or all, of the endpoints of a service (there is a max of 100 endpoints per slice)
- A service with many endpoints may be associated with multiple EndpointSlices
- EndpointSlices provide efficiency by breaking up large lists of endpoints into smaller more manageable units

```yaml
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: my-service-endpoints
  namespace: default
  labels:
    kubernetes.io/service-name: my-service
addressType: IPv4
endpoints:
  - addresses:
      - "10.0.0.1"
    conditions:
      ready: true
    ports:
      - name: http
        port: 80
        protocol: TCP
  - addresses:
      - "10.0.0.2"
    conditions:
      ready: true
    ports:
      - name: http
        port: 80
        protocol: TCP
```

Lists pod IPs and ports for a service

Recap

- **Features:**

Service discovery & load balancing

- Allocation of IP addresses & DNS names & labels/selectors
- Load-balancing across service replicas

Self-healing

- Restarts failed containers
- Replaces and reschedules containers when hosts die
- Kills non-responsive containers

Scheduling and Scaling (horizontal)

May be automated through scaling policies

Automated rollouts/rollbacks

- **CI/CD**
- Rollouts - Progressive updates (to app code or to app configuration)
- Rollbacks – Revert to prior state if unsuccessful update
- Canary deployments