An Overview of Kubernetes Scheduling

Hailiang ZHAO @ ZJU-CS

http://hliangzhao.me

October 13, 2022

The content of the slide is based on the survey *Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges* and related official documents (https://kubernetes.io/docs/home/).

Outline

Outline

What is Cloud-Native?

The Cloud Native Computing Foundation (CNCF) defines Cloud-Native as a new computing paradigm in which applications are built based on a **microservice** architecture, packaged as **containers**, and dynamically scheduled and managed by an orchestrator.

- Microservices are deployed and managed independently and communicate with each other over a network
- Containers are the *de facto* standard for implementing these microservices

How Container Works

Containers make use of the *native* isolation capabilities of modern operating systems with a low overhead in resource consumption and obtaining great flexibility in their deployment.

- Different services are packed in separate and intercommunicating containers
- Container Orchestration is needed to automate the deployment, management, scaling, interconnection, and availability of the container-based applications

$\Longrightarrow$ **Container as a Service (CaaS)**: As a service model to simplify the deployment of containerized applications in the cloud. It undertakes *authentication, logging, security, monitoring, networking, load balancing, auto-scaling*, and *continuous integration/continuous delivery (CI/CD) functions*.

Container Orchestration Toolkits

The orchestrators handle a cluster of physical or virtual machines to host the containerized application. In particular, the task of *assigning physical resources to containers* is performed by the scheduler $\Longrightarrow$ The focus of this slide!

Target of scheduler: improve resource utilization, reduce energy consumption, satisfy the latency requirements, ...

Trending orchestrators:

1. Kubernetes (and K3S, KubeEdge, ...)
2. Docker Swarm
3. Mesos
4. YARN
5. ...

Outline

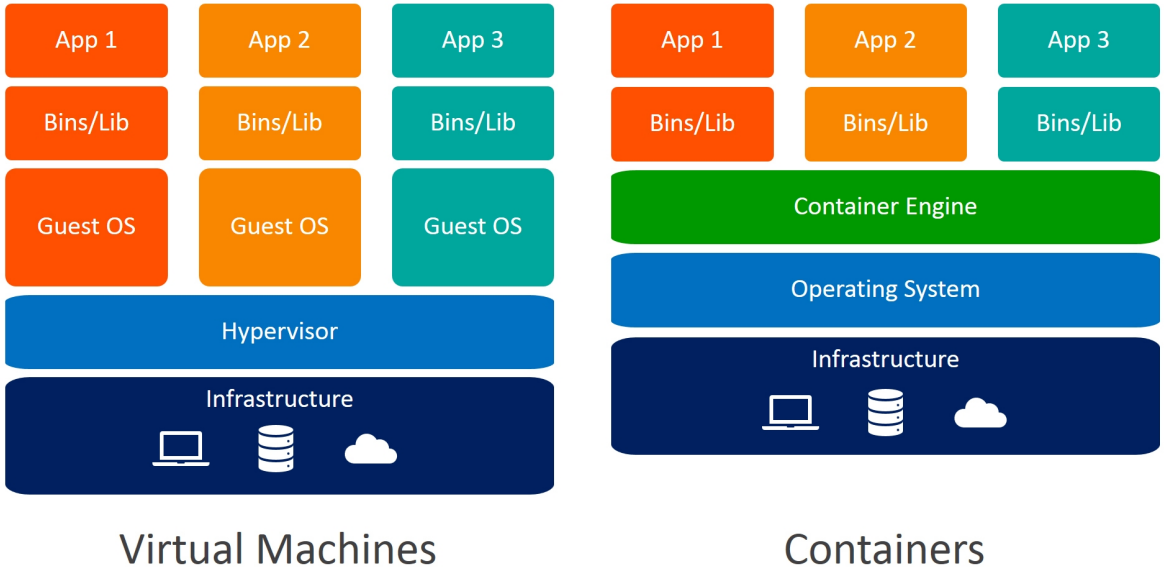
Virtualization and Docker

Virtualization is a technology that enables the creation of *logical* services by means of resources running on hardware. A container is a group of one or more *processes* that are **isolated** from the rest of the system. It comprises an application and all its library dependencies and configuration files.

Docker, the most famous container manager, leverages kernel tools such as *cgroups*, *namespaces*, *chroot*, etc., for its implementation. Docker is a *layered* container platform that comprises several software components for developing, transporting, and running containerized applications. It consists of *Docker Daemon*, *Containerd*, and *Docker Registry*, etc.

Virtualization and Docker

Docker is more lightweight compared with VM.

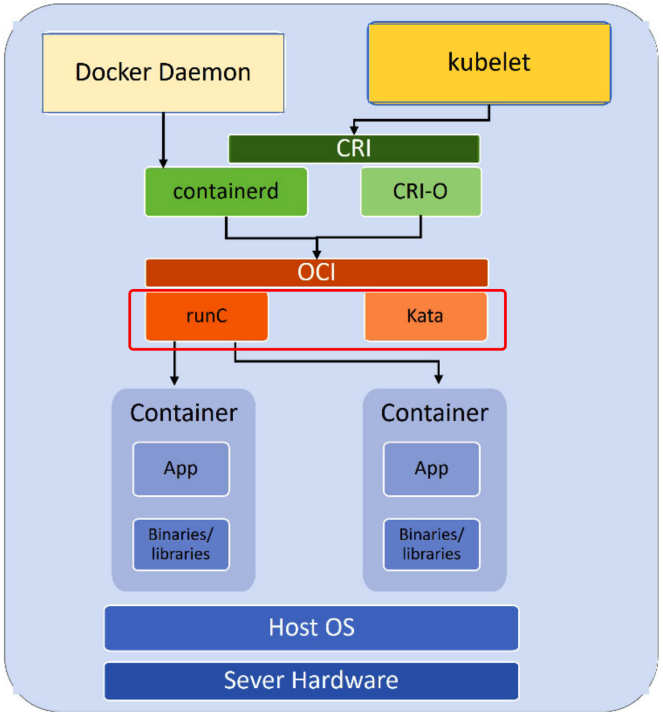


Virtual Machines Containers

- With Docker, application is sandboxed inside of the isolation features that a container provides, but still *shares the same kernel* as other containers on the same host
- With a VM, everything running inside the VM is *independent* of the host operating system, or hypervisor

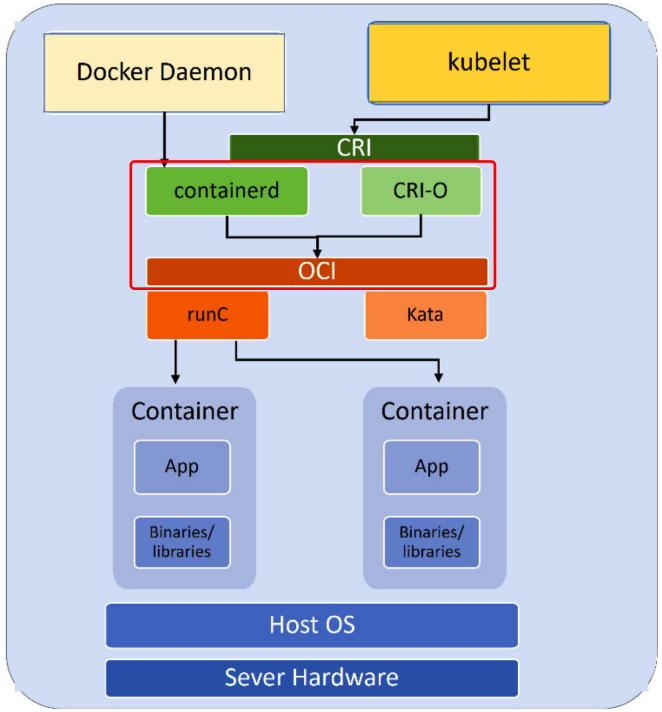
Container Management on a Node

a) At the lowest level of container technology rest **container runtimes**, such as LXC, RunC, CRun, or Kata, that create and run the container processes.



Container Management on a Node

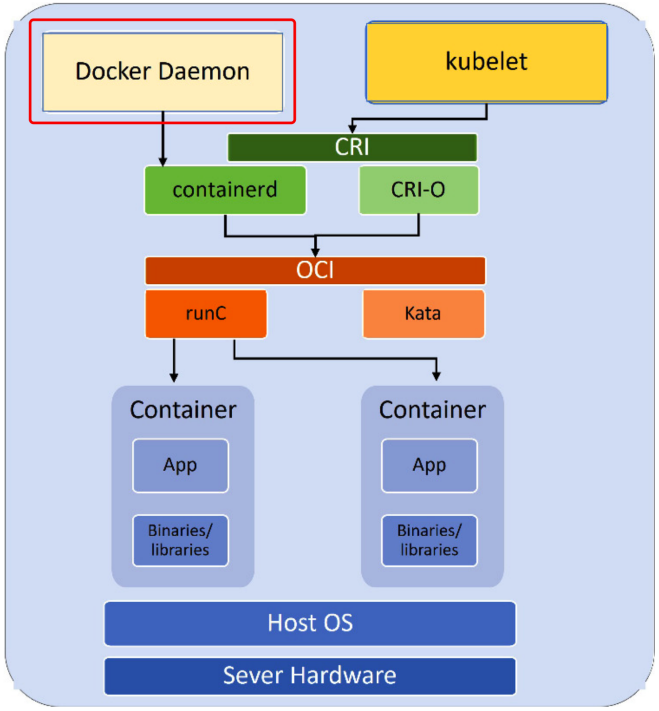
b) A high-level container runtime manages the lifecycle of the container. It pull container images from registries, manage images and hand them over to the lower-level runtimes.



Containerd and CRI-O are typical high-level container runtimes that implement *the Open Container Initiative* (OCI), a standard specification for image formats and runtimes requirements providing container portability.

Container Management on a Node

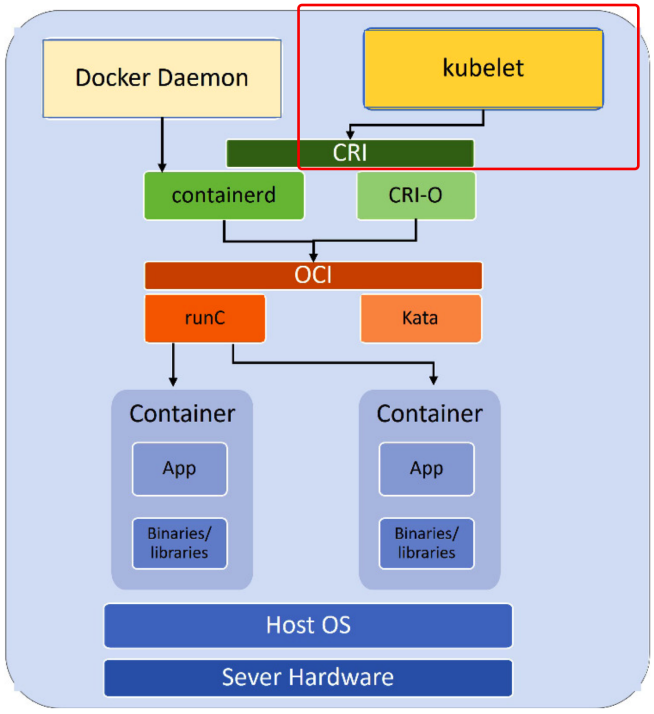
c) Usually, the container runtimes are wrapped in software components called **container managers** or engines that increase the level of abstraction.



Docker Daemon acts as a container manager with an API that simplifies the management of the lifecycle of the containers and communicates with Containerd.

Container Management on a Node

d) kubelet is a component of Kubernetes that communicates with the high-level container runtimes which implement the standard *Container Runtime Interface* (CRI).



It manages the containers by automating the scheduling, deployment, availability, load balancing, and networking, etc.

What Orchestrators Do

A container orchestrator manages and organizes microservice architectures *at scale*, dealing with the automation and lifecycle management of containers and services in a cluster.

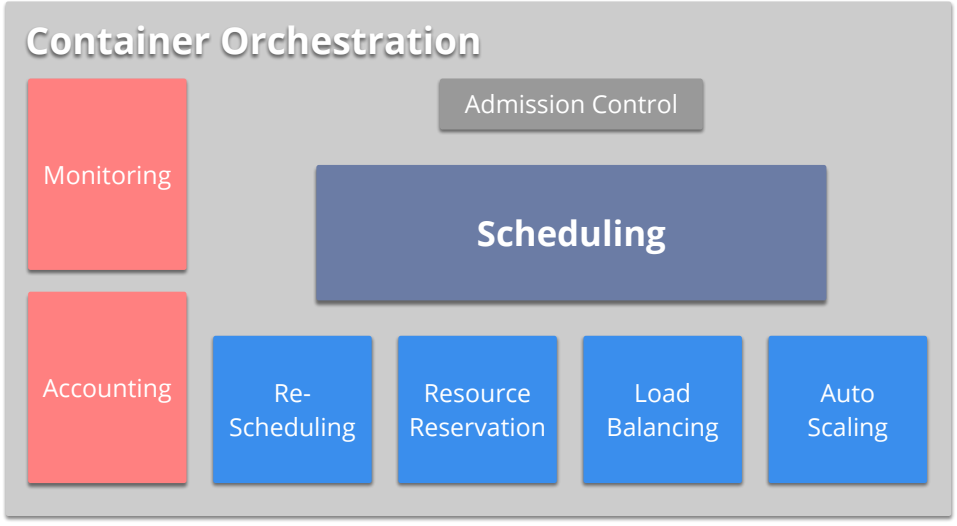
1. End users submit jobs or tasks
2. The orchestrator assigns the jobs and tasks (their container instances) to the worker nodes

Orchestrators can be classified as on-premise or managed solutions:

- **On-premise**: Borg, Kubernetes, Mesos
- **Managed solutions**: Google Kubernetes Engine (GKE), Microsoft Azure Kubernetes Service (AKS), Amazon Elastic Kubernetes Service (Amazon EKS), etc.

Components of Orchestrators

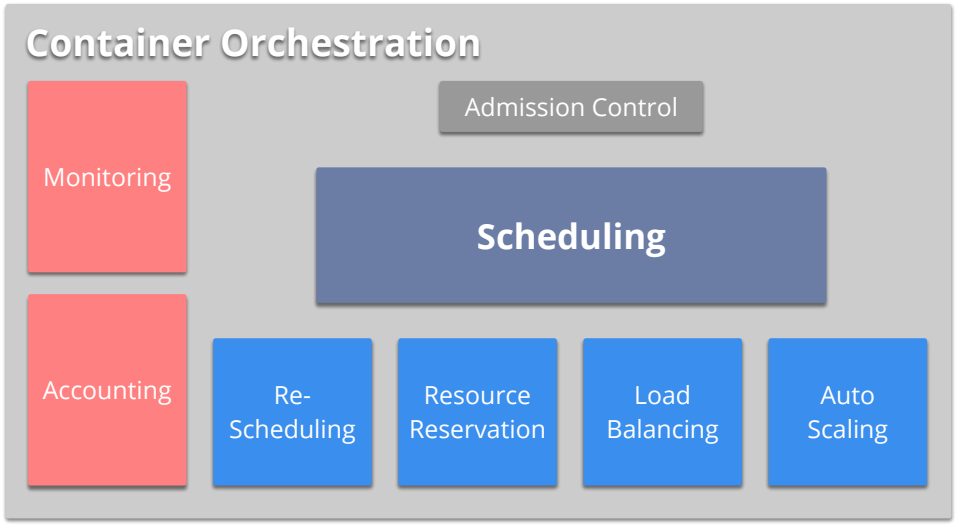
The figure shows the most characteristic components of both on-premise and managed cluster orchestrators.



The *scheduling* module is responsible for identifying the best location to complete incoming tasks. Most scheduling policies map containers based on **the state of the system** (resource constraints, node affinity, data location) as well as **metrics** such as power consumption, response time, or makespan.

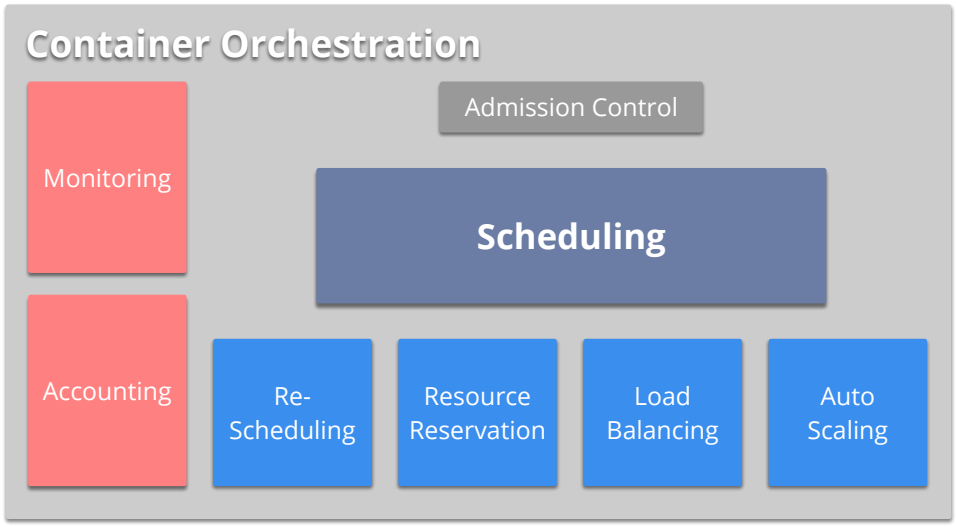
Components of Orchestrators

The *rescheduler* component implements a task relocation policy that determines a new location for a task. This relocation can be triggered for **preemption** reasons or by the system state to consolidate the load or improve resource utilization.



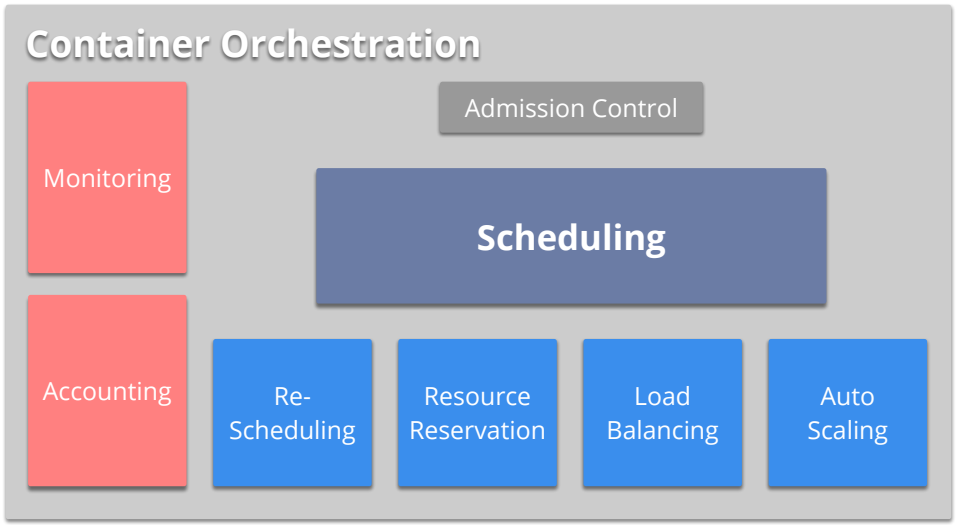
Components of Orchestrators

The *resource reservation* module **reserves** the cluster resources following a request-based approach that can be static or dynamic over time.



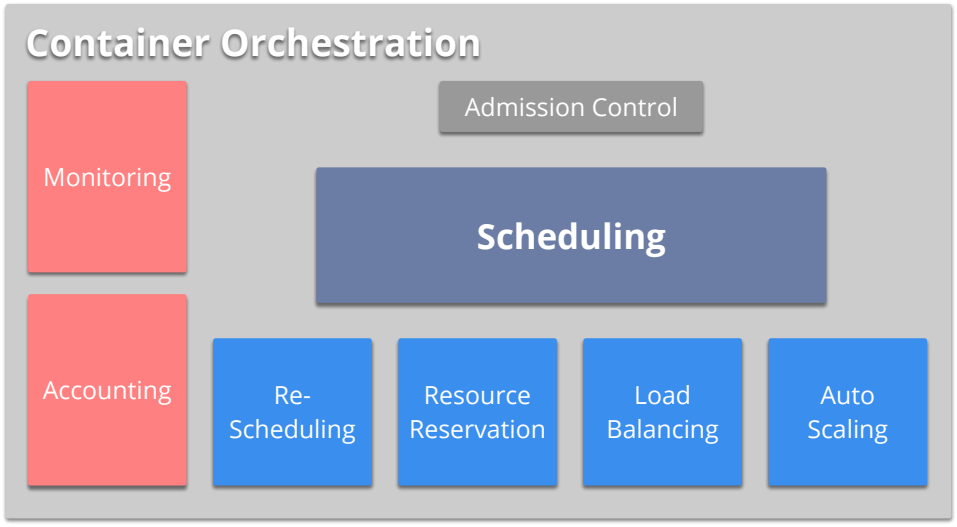
Components of Orchestrators

The *load balancing* module is in charge of distributing tasks/user requests across container instances based on criteria such as **fairness, cost-energy, or priority**. The default policy for load balancing is round-robin.



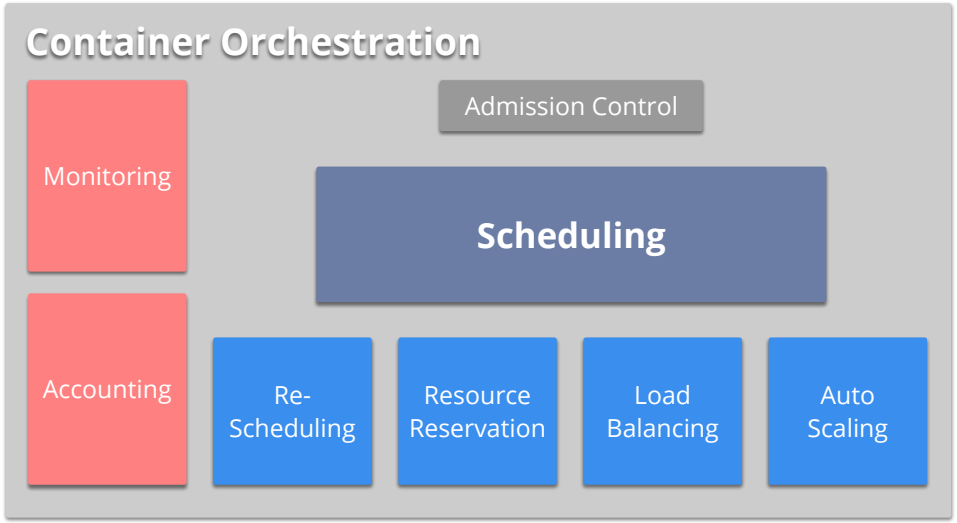
Components of Orchestrators

The *autoscaling* module is in charge of providing horizontal and vertical scaling depending on the workload demand. In the **horizontal** scaling, nodes are added or deleted while in the **vertical** autoscaling the node resources associated with a task are increased or reduced.



Components of Orchestrators

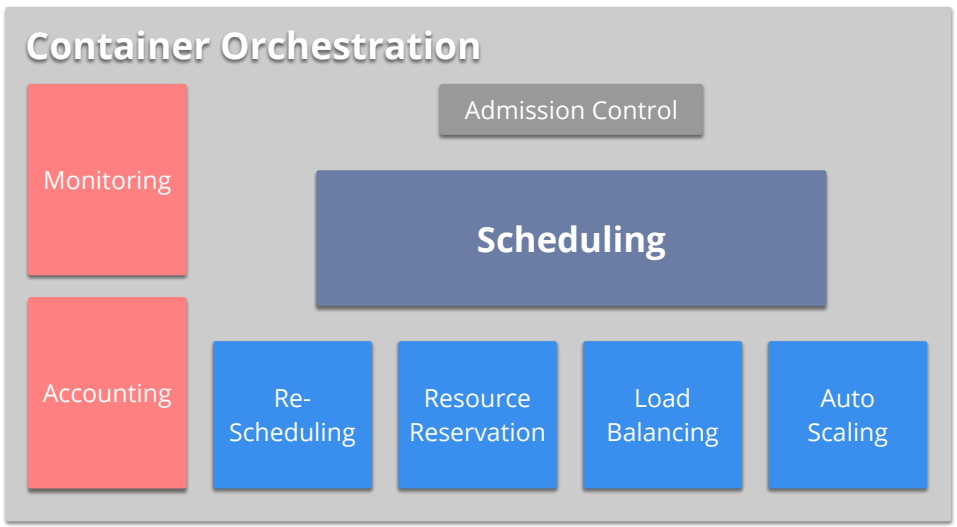
The *admission control* module limits requests to create, delete, modify objects or connect to proxy through **validating** and **mutating**.



An admission controller is a piece of code that intercepts requests to the Kubernetes API server prior to persistence of the object, but after the request is **authenticated** and **authorized**.

Components of Orchestrators

The *accounting* module monitors the available resource for a user while the *monitoring* module **keeps track** of real-time resource consumption metrics for each node and **collects metrics** related to the health of the resources to support fault tolerance systems.



Kubernetes Architecture

The ever-growing K8s ecosystem is composed, to name a few, of complementary tools for

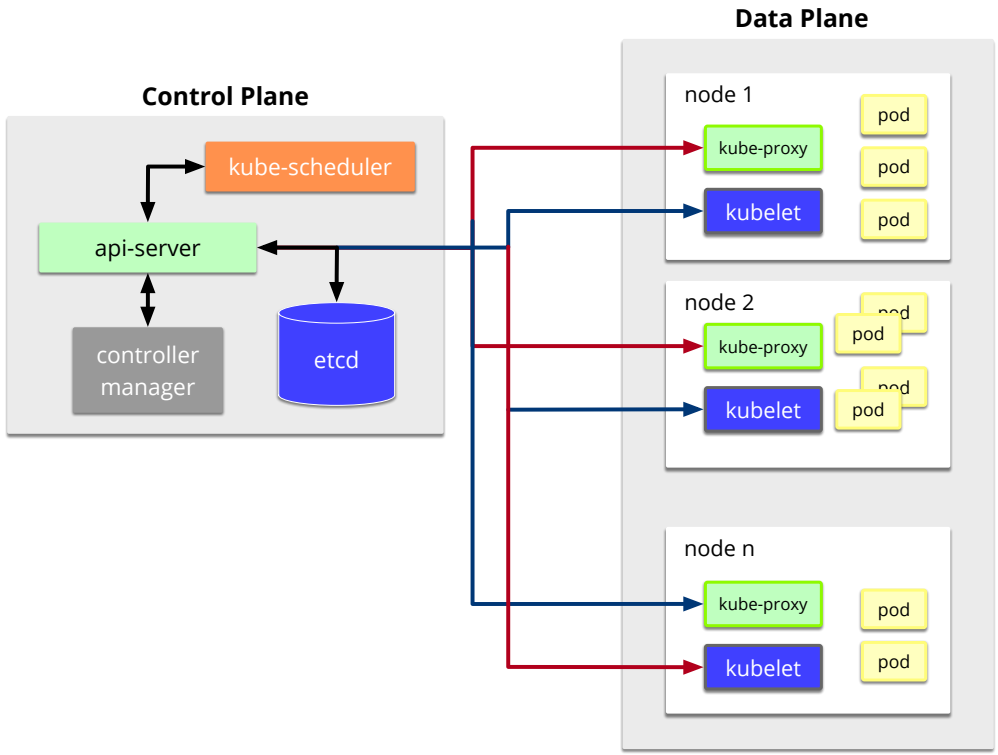
- **management deployments** (Helm, https://helm.sh/)
- **service mesh management** (Istio, http://istio.io/)
- **monitoring** (Prometheus, https://prometheus.io, Grafana, https://grafana.com)
- **logging** (Kibana, https://www.elastic.co/kibana/)

CNCF Cloud Native Interactive Landscape can be found at https://landscape.cncf.io/.

Many scheduling proposals interact with some of these tools to, for example, extract real-time information about the state of the system or predict the use of a resource.

Kubernetes Architecture

From an architectural point of view, a K8s cluster consists of a set of **nodes** (physical or virtual machines) integrated to function as a single entity.



Using software-defined overlay networks, such as *Flannel* or *Calico*, allows K8s to assign a unique IP address to each **pod** and **service**.

Kubernetes Architecture

- The *master* node coordinates the cluster
  1. **etcd** is a key-value database used to synchronize the desired state of the system
  2. **kube-scheduler** places each pod on a worker node
  3. **API server** receives commands and manipulates the data for K8s objects, which are persistent entities representing the state of the cluster. The API server exposes a RESTful HTTP API to describe an object with JSON or YAML. Users can send commands to the API server by using the cli (kubectl)
  4. **controller manager** monitors etcd and forces the system into the desired state. Typical controllers are ReplicaSet, Deployment, Job, DaemonSet, etc. Controllers monitor the status of K8s objects and perform the actions to ensure their successful execution. Scheduling will be triggered if necessary
- The *worker* nodes are in charge of running the pods
  1. **kubelet** is the node agent responsible for the lifecycle of the deployed pods and monitoring pods and node status
  2. **kube-proxy** reflects services on each node and forwards streams across a set of backends

Outline

How K8S Scheduler Works

When defining an application in a cluster, the master node receives the information via the Kubernetes API and deploys the application to the worker node it deems appropriate.

In particular, the scheduler component looks for pods that are in *pending* state, because they have just been created and do not yet have a worker node assigned and finds the best worker node to run the pod.

The placement decision in a worker node is based both on **the scheduling policy** and **the user specification**.

Scheduling in Kubernetes: User specifications

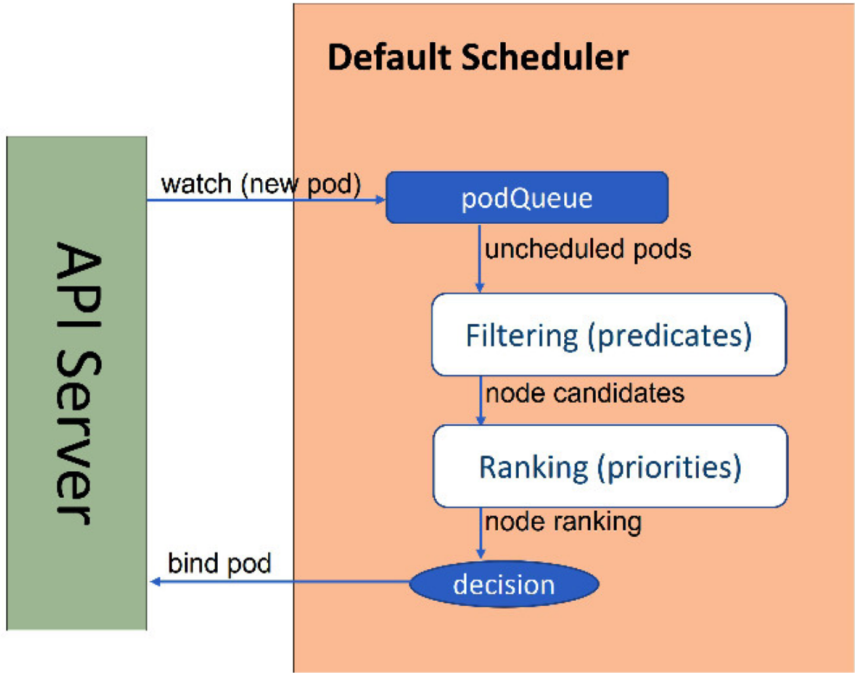
Users can configure a wide range of options to specify the conditions that the scheduler should satisfy. To this end, user specifications indicate different types of constraints that play the role of a control admission. The constraints can be **node-level**, **namespace-level**, or **pod-level**.

- **Node level**: *affinity* and *taint*
- **Pod level**: We can specify how much of each resource a container needs with *requests*/*limit* quota and *priority*
- **Namespace level**: *LimitRange* and *ResourceQuota*

Scheduling in Kubernetes: Internal workflow

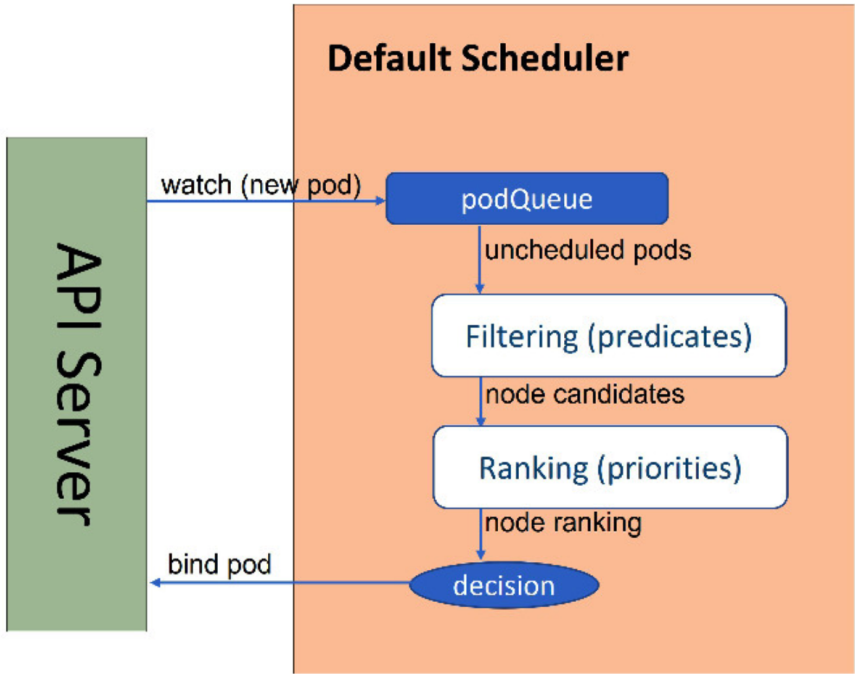
The lifecycle of the kube-scheduler works as follows:

- **a)** The scheduler maintains a queue of pods called *podQueue* that keeps listening to the *API Server*
- **b)** When a pod is created, the pod *metadata* is first written to etcd through the *API Server*



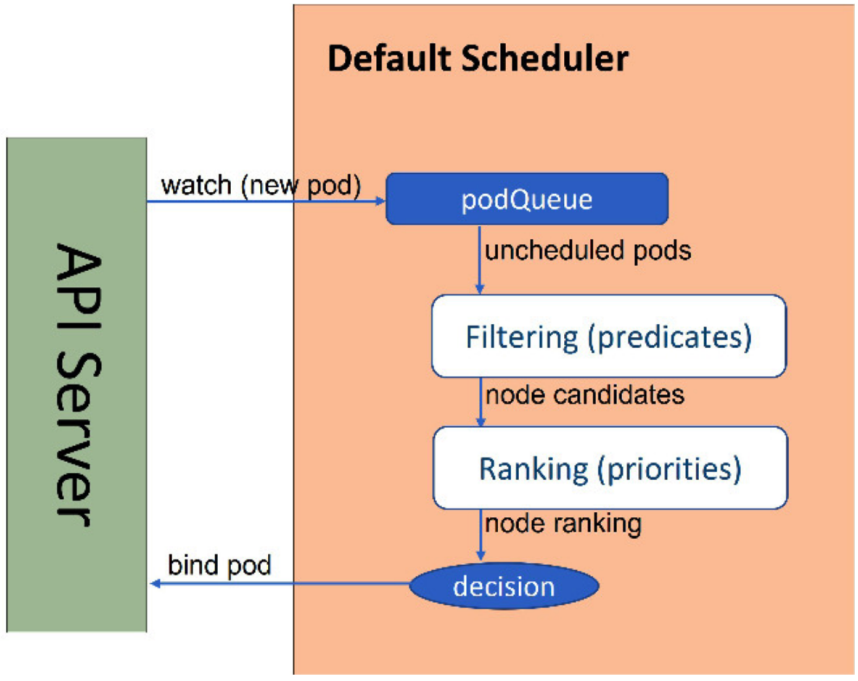
Scheduling in Kubernetes: Internal workflow

- **c)** kube-scheduler, as a controller, follows the watch state, takes action, and updates the state pattern. It watches the unbound pods from the *etcd* and each time an unbound pod is read from the etcd, the pod is added to the *podQueue*



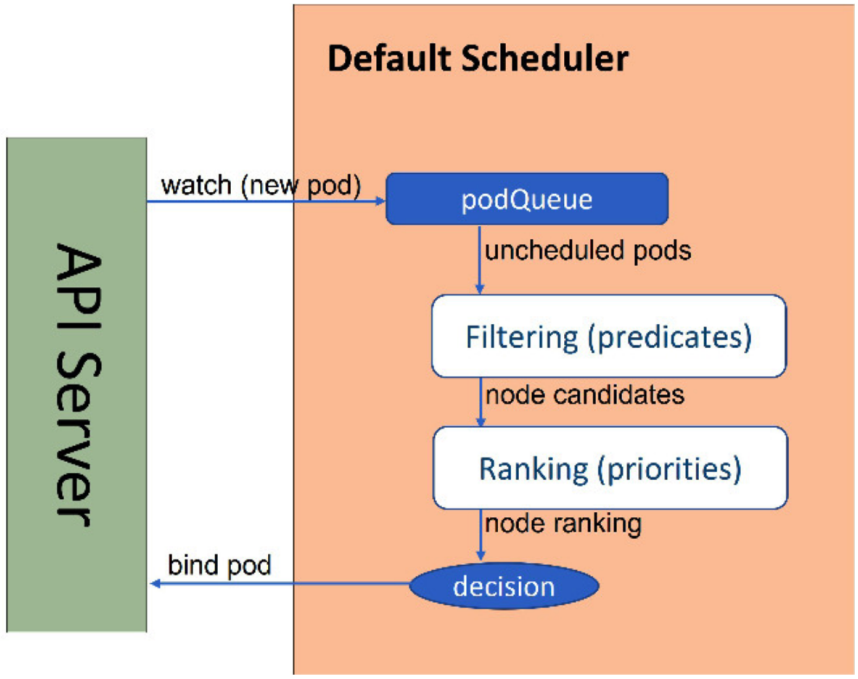
Scheduling in Kubernetes: Internal workflow

- **d)** The main process continuously extracts pods from the *podQueue* and assigns them to the most suitable nodes for running those pods



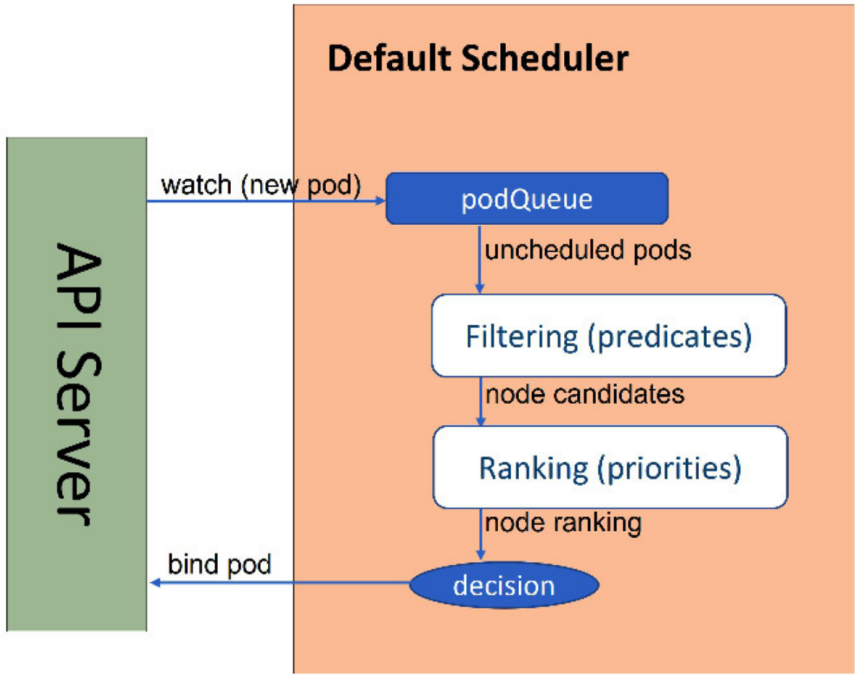
Scheduling in Kubernetes: Internal workflow

- **e)** The scheduler updates the pod-node binding in the *etcd* in order to be notified to the *kubelet* on the worker nodes



Scheduling in Kubernetes: Internal workflow

- **f)** The kubelet component running in the selected worker node, which monitors the object store for assigned pods, is notified that a new pod is in pending execution and it executes the pod. Finally, the pod starts running on the node



Scheduling in Kubernetes: Internal workflow

The logic of the main process iterates over the nodes in a round-robin fashion and performs per each unbinding pod the **filtering** and **ranking** substeps.

1. **Filtering**: Node filtering is based on predicates (Boolean functions that indicate whether a pod fits a worker node)
   - *PodFitsHostPorts*
   - *PodFitsResources*
   - *PodFitsHost*
   - *CheckNodeCondition*
   - ...
2. **Ranking**: Assigns a score to the remaining candidates depending on certain configurations and metrics
   - *LeastRequestPriority*
   - *BalanceResourceAllocation*
   - *SelectorSpreadPriority*
   - ...

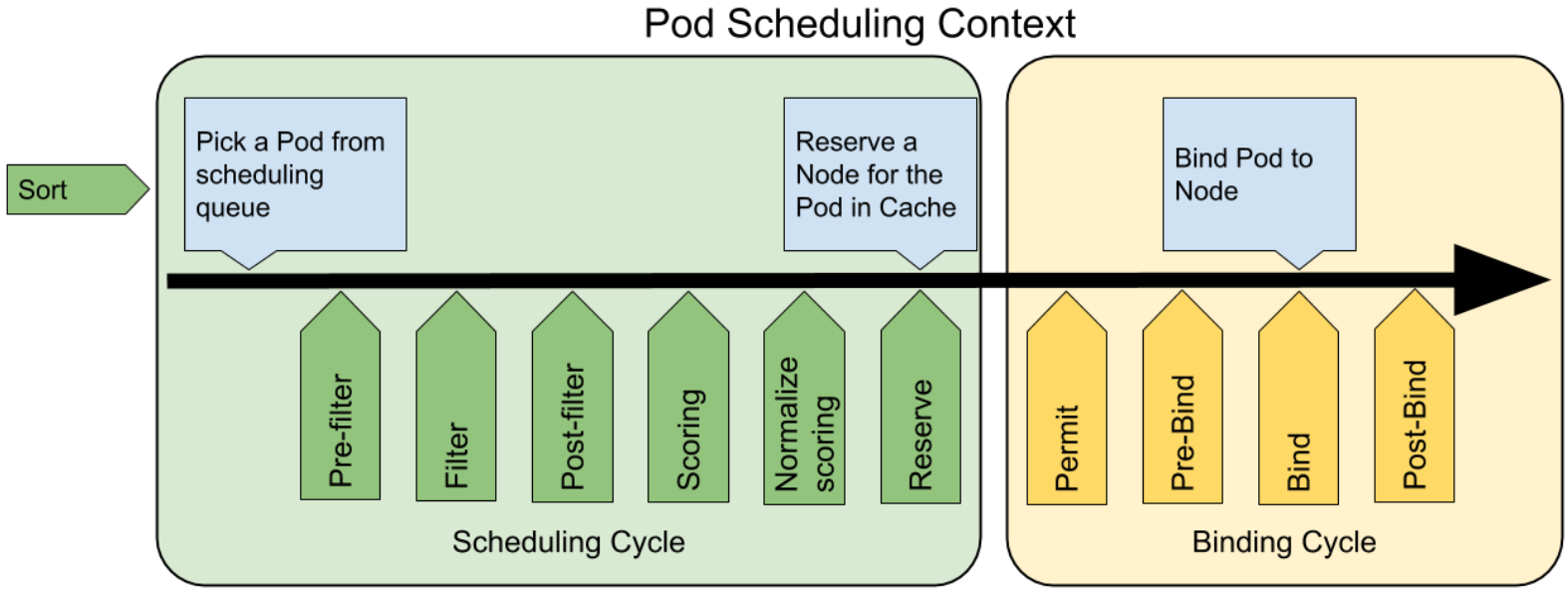
Extending K8S Scheduler

Kubernetes scheduler supports several ways to extend its functionality.

- **a) Clone the code, add new predicates and/or priorities** to the default scheduler and recompile it
- **b) Implement an extender process** that the default scheduler invokes as the final step
  - The scheduler extender is a configurable webhook that contains filter and priority endpoints that correspond to the two main phases of the scheduling cycle (filtering and ranking)
  - The state of the entire cluster stored in the cache of the default scheduler is not shared with the scheduler extender and data communication is done through serial HTTP communication with the associated communication costs

Extending K8S Scheduler

- **c) Use the scheduling framework**, which is officially recommended
  - It defines new *extension points* (queue sort, pre-filter, filter, score, reserve, etc.) that are integrated as plugins into the existing scheduler at compile time
  - We can write new scheduling functions in the form of plugins, and implement a custom scheduler process that can run instead of or alongside the default scheduler



Outline

Infrastructure Domain

- **Physical Layer:**
  - The K8s scheduler must manage *worker nodes with low resource capacities* as well as different processor architectures. These hardware-constrained nodes impose difficult limitations on the scheduling system and the K8s scheduler must assign resources efficiently, for example, to prevent SBCs from running out of memory when an application is running
  - *Device-aware scheduling* is also important (GPUs, TPUs, FPGAs, etc.)
- **Virtual Layer:**
  - *Network-aware scheduling* in K8s is particularly important, especially for the cloud-edge infrastructure (scalability and geo-distributed environments)
  - The default K8s scheduler is not aware of *hardware resource fragmentation* and *container interference*, which leads to poor isolation of multi-tenant applications

Cluster Domain

- **The Component Subdomain** comprises design parameters related to the scheduler architecture and multi-cluster or federation scheduler, as well as resource management components closely linked to the scheduler in K8s
  - Centralized scheduler $\Rightarrow$ Schedulers with modular, two-level, or distributed architectures
  - Single K8s cluster $\Rightarrow$ Multi-cluster scheduler (KubeFed)
- **The Environment Subdomain** refers to the computational technologies *where* K8s can be deployed
  - Scheduling in *edge/fog computing architectures* requires fully dynamic, network-aware, and fault tolerance solutions

Scheduling Domain

- Mathematical models
  - Mathematical modeling (ILP)
  - Heuristic (FirstFit, FIFO, DRF, etc.)
  - Meta-heuristic (GA, PSO, etc.)
  - Machine learning (DNNs, RL)
  - Gang scheduling (dependency-aware task gang scheduling)
- Metrics
  - **Infrastructure**: resource utilization (computation, storage, network), failure rate, interference, energy, etc.
  - **Cluster**: location, mobility, stability, reliability, availability, etc.
  - **Application**: response time, completion time, makespan, resource demand, etc.

Application Domain

- Application Type
  - HPC, machine learning, batch, web server, IoT, serverless applications, etc.
- Application Architecture
  - Monolithic
  - Distributed inter-dependent tasks (SFCs, DAGs)
  - Workflows that have loops
- Workloads
  - Workloads of different service types co-locate (low resource utilization, resource reservation, etc.)

Performance Domain

- Evaluation Tools
  - Simulations without standard rules
  - Small testbeds with simple benchmark policies
- Performance Metrics
  - From *resource providers'* perspective: cost (resource utilization, etc.)
  - From *end users'* perspective: QoS (JCT, makespan, total processing time, etc.)

Outline

Future Directions and Open Issues

- **Infrastructure Domain:** Resource virtualization and management of new physical devices (GPUs, TPUs, FPGAs, etc.)
- **Cluster Domain:**
  - *Distributed control plane.* The centralized nature of the K8s orchestration system does not align well with the needs of the distributed fog/edge computing environments
  - *Collaboration among clusters.* Kubernetes Cluster Federation (KubeFed, https://github.com/kubernetes-sigs/kubefed) is working on this
- **Scheduling Domain:**
  - *Advance context-aware scheduling algorithm*
  - *Scalability.* The scheduling should support resource management on dynamically scalable hardware architectures

Future Directions and Open Issues

- **Application Domain:**
  - *Workflows and microservices.* The scheduler should provide native support to the efficient scheduling of workflows
  - *Model workloads.* Understanding the characteristics and patterns of workloads running on K8s clusters is a critical task for improving K8s scheduling
- **Performance Domain:**
  - *Comparability and usability.* Current scheduling algorithms are isolated with little interaction between them
  - *No widely accepted simulation tools.* The fact that the evaluation environments are developed in an ad-hoc manner does not allow for efficient and effective analysis of the diferent algorithms
  - *Security issues*