Catherine Yu and Hollis Ma
COS 316 Final Project

kubernetes

Introduction

In 2013, Docker was released to the world. Docker is a tool that simplifies the creation and deployment of applications by packaging software into containers. Containers are more lightweight than virtual machines (VMs), maintaining the same properties as VMs, and providing other benefits, including efficient, continuous, and reliable deployment, decoupling of applications from infrastructure, portability, and resource isolation and utilization. However, Docker only focuses on individual containers and running them on individual machines. This is where Kubernetes (also known as K8s) comes in--Kubernetes is a system that coordinates a cluster of computers to deploy and manage large numbers of containers. Moreover, Kubernetes provides several other features to manage applications safely and efficiently, including service discovery, load balancing, automated rollouts and rollbacks, and self-healing.

Basics of Kubernetes

We begin by defining several terms and reviewing Kubernetes basic architecture and objects. When a user deploys Kubernetes, they will receive a set of machines called the **Cluster**. The machines are called **Nodes** and run containerized applications. Each Cluster has a master node and at least one worker node (for the rest of the paper, worker nodes will simply be known as Nodes). The Nodes host **Pods**, which are collections of **containers**, and the **Master Node** controls the Nodes and their Pods by communicating with Nodes via **Kubelets**, which are a part of each Node and ensures that containers are running in every Pod in the Node.

To communicate between Pods, **Services** expose an **Endpoint** that objects can send requests to, and they then redirect the request to the correct Pod to allow for informational and behavioral manipulation of Pods.

A **Deployment** manages a replicated application and describes a desired state for Pods and **ReplicaSets**, which maintains a specific number of Pods in a Node. **Volumes** are storages within Pods and act as the memory for applications. **Namespaces** are used to create virtual clusters backed by the same physical cluster. **Kubectl** is a command-line tool for running commands on Kubernetes clusters, and is used primarily by users to communicate with the Kubernetes API.

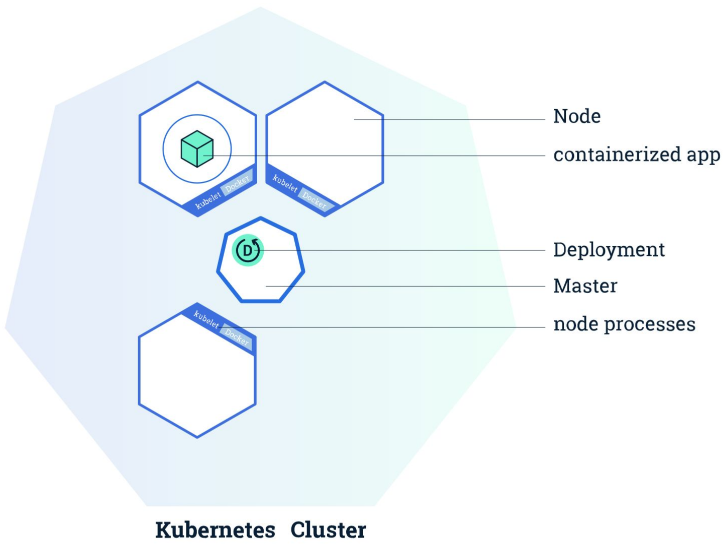

Figure 1. Cluster architecture

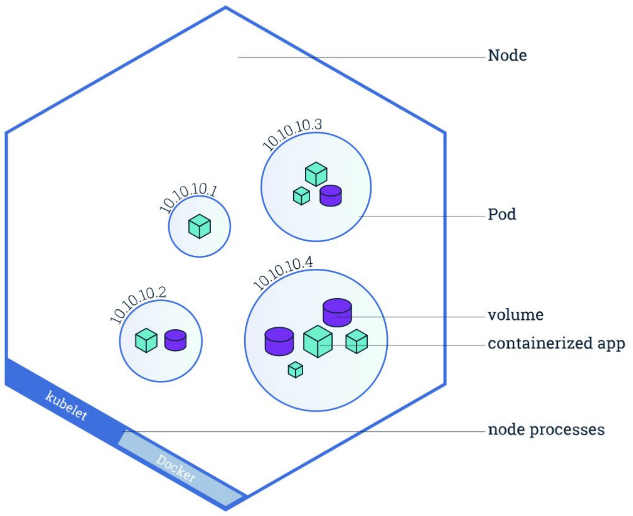

Figure 2. Node architecture

Naming

Every Kubernetes object has a **Name** and a **UID** to identify the object locally and globally. The **Name** is a client-provided string that is unique across the object's resource type, and it is stored in the metadata of the object. For example, only one Pod can have the name "1234" in the same namespace, but a Deployment can also have the name "1234". If the client does not specify a name for an object, Kubernetes will auto-generate a pseudo-random name. The pool of characters that Kubernetes randomly draws from does not include vowels and a few numbers to decrease the chance that a “bad word” might be accidently chosen. The **UID** is a Kubernetes systems-generated string to uniquely identify objects across the cluster. Every object’s UID is distinct from any other UID created for the entire lifetime of a cluster, so the UID helps to distinguish between historical occurrences of objects.

Kubernetes uses a specific **naming convention** for ReplicaSets and Pods following a Deployment. When a client creates a Deployment, they will give the Deployment a name. Kubernetes will create a ReplicaSet based on the Deployment with the name by attaching a randomly generated string after it with a hyphen, i.e. `<deployment-name>-<random>`. The ReplicaSet will create the Pods and name them by adding another randomly generated string to the end of the ReplicaSet’s name, i.e. `<deployment-name>-<random>-<random>`.

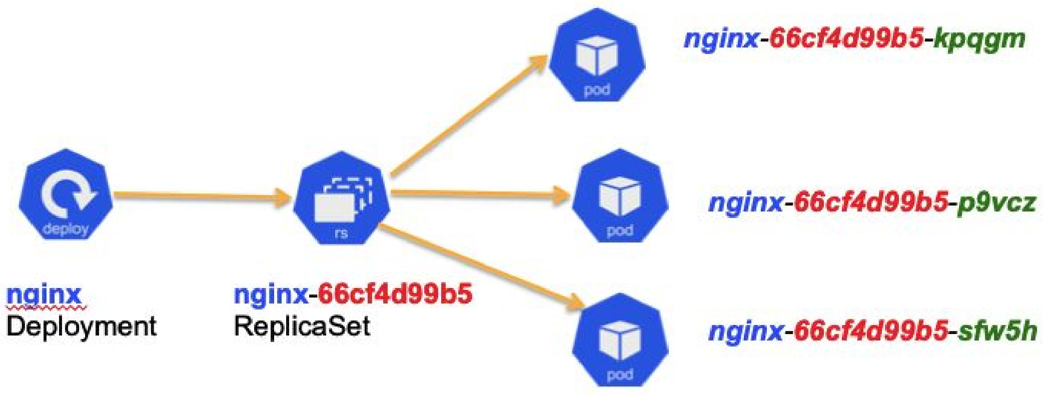

Figure 3. Kubernetes naming convention

Another way to identity and filter Kubernetes objects are **labels**, which are key-value pairs used to identify specific attributes that are relevant to users but do not imply semantics to the core system. Clients can use labels to get, retrieve, and organize objects. To allocate a label to an object, the key-value pair is stored in the configuration files of the object, and to translate a key-value pair to a set of objects, label selectors can be used. Clients can identify objects with the same label by using two types of **label selectors**, which are made up of requirements. Equality-based requirements allow filtering by label keys and values, such as `environment = production` or `tier != frontend`. Set-based requirements allow filtering keys by a set of values, such as `environment in (production, qa)` or `tier notin (frontend, backend)`.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-demo
  labels:
    environment: production
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

Example manifest for a Pod with labels

Caching

Caching is important in every large application that needs to perform actions at specific times because it drastically lowers the time needed for certain instructions. For Kubernetes, caching is used to quickly fetch objects so that other objects can use them. These caches can be external caches like Redis, or internal caches which are commonly stored in **Volumes**. If a client wants a cache to be available throughout an entire cluster, persistent volumes can be used because they are accessible from anywhere within a cluster, as opposed to volumes only being local to Nodes.

An example of caching in Kubernetes can be seen the **NodeLocal DNSCache**, which caches DNS queries by Pods to make the queries more efficient. The NodeLocal DNSCache does this by running a DNS caching agent on the Cluster Nodes as a DaemonSet--a DaemonSet ensures that all (or some specified) Nodes run a copy of a certain Pod. Each of these Pods will then be able to query the DNS caching agent, avoiding the traditional, slower usage of iptables and connection tracking.

Each Kubernetes **Master Node** will need to schedule applications to maximize CPU usage. The CPU Manager can determine how sensitive the processor should be to cache misses and can also reap benefits from sharing processor resources (e.g., data and instruction caches). Having a central scheduler also allows partitioning of CPUs among workloads, reducing interference between resources, including not just CPUs, but also cache hierarchies and memory bandwidth.

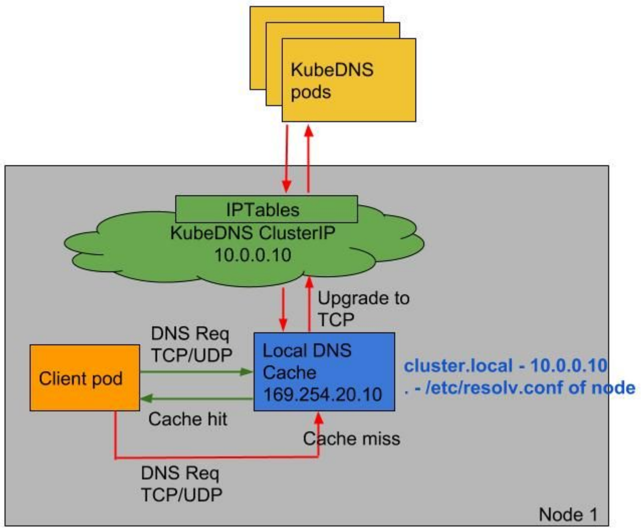

Figure 4. NodeLocal DNSCache architecture and flow

Resource Management

Kubernetes has many examples of resource management. We'll first discuss resources shared across Kubernetes objects. As defined in the Basics of Kubernetes, **Pods** contain container images. These containers are tightly coupled and use the same disk space, volume, memory, and network information (IP addresses, port number). Pods in the same **Node** share the CPU and memory with each other on that Node. In general, the total amount of compute resources available to the Node, which is made up by the CPU, memory, and maximum number of pods that can be scheduled on that node, is called the **Node Allocatable**. Some of these resources are reserved for Kubernetes system daemons and OS system daemons.

Nodes and the Pods they contain are managed by the Master Node in a cluster. The **Master Node** is responsible for maintaining the desired state of the cluster through communication with the rest of the cluster and running the scheduler and controllers, specifically through its components: `kube-apiserver`, `etcd`, `kube-scheduler`, `kube-controller-manager`, `cloud-controller-manager`. The first two components respectively provide a frontend to the cluster's shared state for the objects in the cluster and provide a backing store for all cluster data. We will discuss the latter three in the following paragraphs.

Controllers are like thermostats--they work to maintain the desired state of the cluster, and they try to move the current cluster state closer to this desired state. Each Controller tracks at least one Kubernetes resource type. Most Controllers interact with the cluster API server, although some act by direct control. Kubernetes has built-in controllers that run in the kube-controller-manager, which include a Node Controller (Nodes), Replication Controller (ReplicaSets), Deployment Controller (Deployments), and more. Many of these controllers also have cloud provider dependencies, which are taken care of by the cloud-controller-manager.

Scheduling in Kubernetes means making sure that Pods are matched to Nodes so that their containers can be run by Kubelet in the Nodes. `kube-scheduler` is the default scheduler and is in charge of watching for newly created Pods without a Node assignment. Every container has different requirements for resources, and every Pod has its own requirements for resources, so if there are no feasible Nodes, the Pod will have to wait until the scheduler can assign it somewhere. The scheduler decides where to assign new Pods based on scheduling policies. There are 2 steps: 1) **filtering** and 2) **scoring**. In filtering, the scheduler finds a set of feasible Nodes to schedule the Pod, and in scoring, the scheduler ranks the Nodes to choose the best placement--the Node with the highest ranking (randomly chosen if there is a tie) has priority. Lastly, the scheduler notifies the API server about its decision in a process called **binding**.

Filtering →	→ Scoring →	→ Binding
e.g. `PodFitsResources`: check if Node has free resources for Pod e.g. `CheckNodeMemoryPressure`: Pod won't be scheduled on a Node if it has memory pressure	e.g. `LeastRequestedPriority`: favors nodes with fewer resources e.g. `ImageLocalityPriority`: favors nodes that already have container images for Pod cached	

In large clusters, the user can define a `percentageOfNodesToScore` parameter to dictate how many Nodes they want to include in the assignment process (a percentage of the total cluster size), which improves scheduler performance. In this case, the scheduler will scan the Nodes in round-robin fashion.

We will now discuss resource requirements of containers and Pods, which were mentioned in scheduling but not clarified. When a user specifies a Pod, they can also specify how much CPU and memory (RAM) each container needs in terms of a **request** and a **limit** for each resource. The request is the amount that the system will guarantee to the container, while the limit is the maximum amount the system will allow the container to use. When the request is less than the limit, Kubernetes can oversubscribe Nodes while maintaining resource guarantees, increasing utilization of Nodes overall.

The Pod's resource requirements are calculated by totalling the resource requirements of all the containers in the Pod. However, the Pod itself also requires system resources for its infrastructure, called the **Pod Overhead**. Together, these requirements are used for scheduling the Pod on Nodes. Using CPU and memory requests and limits give Pods a better chance of being scheduled. Not setting a limit means that a Pod has no upper bound on the CPU power it uses, which may lead to starvation of other Pods and Nodes.

```
...
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    resources:
      limits:
        cpu: “1”
        memory: “200Mi”
      requests:
        cpu: “0.5”
        memory: “100Mi”
```

Example Pod with resource requirements

Lastly, CPU is managed on Nodes by **Kubelets** and the **CPU Manager**. In general, Kubelets use Completely Fair Scheduler (CFS) quota to enforce pod CPU limits, but Kubelets also allow alternative CPU management policies in cases where workload performance may be affected. This is provided by the CPU Manager, which enables better placement of workloads by allocating exclusive CPUs to certain containers and can provide better performance in different scenarios such as increased context switches. The CPU Manager can especially help workloads that are sensitive to CPU throttling, processor cache misses, context switches, cross-socket memory traffic, and hyperthreads.

Another resource managed by Kubernetes deals with **networks** and includes **network bandwidth** and **endpoints**. Pods need to communicate with other Pods to be able to query or send information and manipulate the state or behavior of other Pods. To identity which Pod is being queried to or from, Pods use their unique IP addresses, which are dependent on which machine they live on. However, a problem surfaces when Pods die and the applications running in that Pod are relocated. The IP addresses of where those applications are running change, and normally an iptable is used to reflect these changes, but iptables have high overhead and can become slow in high traffic networks.

Kubernetes solves this communication problem by using **Services**, which are an abstract way to expose an application running on a set of Pods as a network service. This microservice is a REST object that receives requests to **Endpoints** that get updated whenever the set of Pods in a Service change, and forwards them to the relevant set of Pod. By abstracting away the process of querying a specific Pod, Services allow for efficient management of Endpoints and network bandwidth due to the decrease in complexity and overhead of determining how to route a request.

Kubernetes has recently introduced another endpoint management feature in the form of **Endpoint Slices**, which allow for distributing network endpoints across multiple resources. This dissemination ensures that no single Endpoint gets overloaded and also spreads out endpoint traffic to improve performance.

There are many alternatives to Services and Endpoints, but the policies these methods use to manage resources lead to inefficiencies. For example, instead of proxying to forward inbound traffic to backends, Kubernetes could configure DNS records and use **round-robin name resolution**. The problem with this is that some apps do not respect TTLs and cache results after they should have expired or do DNS lookups only once then cache the results indefinitely.

Another alternative that still uses proxying is the **user space proxy mode**, where a kube-proxy opens or closes a port on a local Node whenever a Service or Endpoint is added or removed. A limitation of this mode is that it obscures the source IP address of packets accessing a Service, so certain kinds of network filtering like firewalling become impossible.

Finally, we will discuss management of storage. Each Pod has **Volumes**, which provide storage for containers in a Pod, are accessible to all containers in a Pod, and preserve container files even if the container restarts. Volumes cease to exist when the Pod ceases to exist. A Volume can be shared for multiple uses by specifying a subPath.

```
...
spec:
  containers:
  - name: mysql
    image: mysql
    volumeMounts:
    - mountPath: /var/lib/mysql
      name: site-data
      subPath: mysql
  ...
  volumes:
  - name: site-data
```

Example Pod with a LAMP stack using a single, shared volume.
The databases are stored in the mysql folder, an example of subPath.

Kubernetes also allows for a cluster-wide storage system composed of two resources. A **Persistent Volume (PV)** is a piece of storage that remains unchanged even when Nodes and Pods die. PVs are a level of abstraction above that of the actual implementation of the storage, which may be NFS, CSI, or other supported types. A **PersistentVolumeClaim (PVC)** is a request for storage by a user; similar to a Pod consuming Node resources, PVCs consume PV resources, and similar to a Pod having specific requirements, PVCs can request specific sizes and access modes.

Virtualization

There are various instances of virtualization throughout the Kubernetes system where Kubernetes presents resources to multiple users/objects as though they each have exclusive access to those resources. This virtual sharing of resources happens at all levels in the Kubernetes ecosystem--from Container images and memory, to clusters and Kubelets. We analyze virtualization in all the levels going from the bottom up.

The lowest level of virtualization that Kubernetes brings into play is the relationship between **containers** and the **applications** they run. Each application is presented with a virtual OS supplied by a container, and it appears to the application that they are the only user of that OS. This virtualization allows a large number of applications to run in parallel on fewer computers than if each application had its own physical OS, and also allows for more modularity in terms of OS compatibility.

Containers can be grouped together to form a **Pod**. Pods use virtualization to increase modularity and reusability of containers, facilitate intra-pod communication, provides ease of management and flexibility for application architectures, and allows containers to share volumes, network, CPU, and more. For example, if we are running an app server pod that contains the app server itself, a logging adapter, and a monitoring adapter, Pods virtualization makes it easier to connect these containers and also allows the logging and monitoring containers to be used in other Pods.

Each **Pod** is contained in a **Node**, which allow Pods to run as if they are the only application running on their virtual or physical machine. Pods can read and write to memory, handle I/O, create their own filesystems, and do whatever they want--all as if they are the only user of their machine. This is possible because of how the Nodes virtualize machines by handling the containers' memory use, CPU usage, etc.

To figure out which applications a **Node** needs to run, **Kubelets** handle communication with the Master Node and the other Nodes in a cluster. The Kubelets use virtualization by being the sole bridge between Nodes and the Master, such that each Node appears to be the only Node receiving instructions on which apps to run, increasing flexibility and reusability of Nodes while decreasing chances of collisions and confusion.

Another example of virtualization within **Nodes** come in the form of **Services**, which create a query-able endpoint for outside entities to manipulate and communicate with Nodes. Services allow for one-to-one communication between Nodes and other objects--each Node receives queries that are meant exclusively for them and each queryer is presented with exclusive access to Services and abstracts away the specifics of which IP address to query. This virtualization allows for faster requests and responses and a cleaner, clearer method of communication.

The final instance of virtualization we consider are **Namespaces**, which support creating different virtual clusters that reside on the same physical cluster. Each of these virtual clusters is unaware of the other virtual clusters and uses the physical cluster as if it were the only virtual cluster. This virtualization allows for users of different teams/projects to be separate but still share resources.

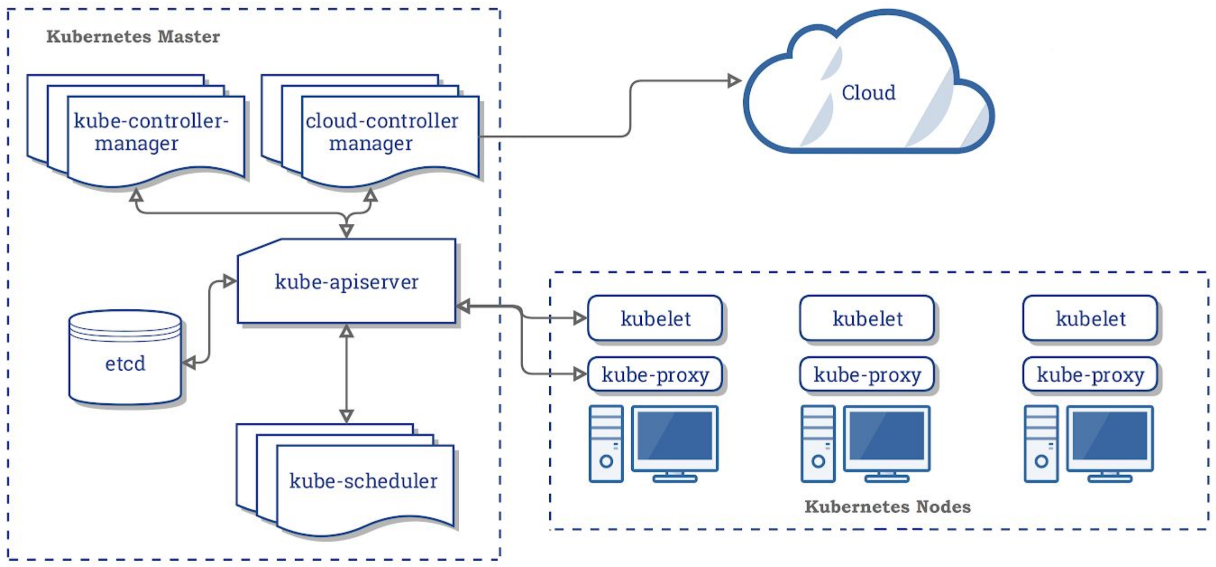

Figure 5. Another diagram of cluster architecture

Access Control

There are two types of users in Kubernetes: **normal users**, which are for humans, and Kubernetes **service accounts**, which are for processes in Pods. Normal users are managed by an outside, independent service, such as Google Accounts, or even a file with usernames and passwords. Service accounts are managed by the Kubernetes API, and they are either created automatically by the API server or manually by API calls. Their differences are compared and listed in the table on the following page.

Normal users	Kubernetes service accounts
• Not represented by any user account object in Kubernetes • Intended to be global, names unique across all namespaces • Typically synced from corporate database - new user account creation requires privileges, tied to business procedures	• Tied to credentials stored as Secrets, which are in Pods • Bound to namespaces • Allows processes to talk to API • Lightweight creation - cluster users can create accounts for specific tasks, based on the principle of least privilege (min. access rights)

Both normal users and service accounts can communicate with the system through the Kubernetes API using kubectl, client libraries, or REST requests. Any requests made by a user must follow a 4-step process to be successful.

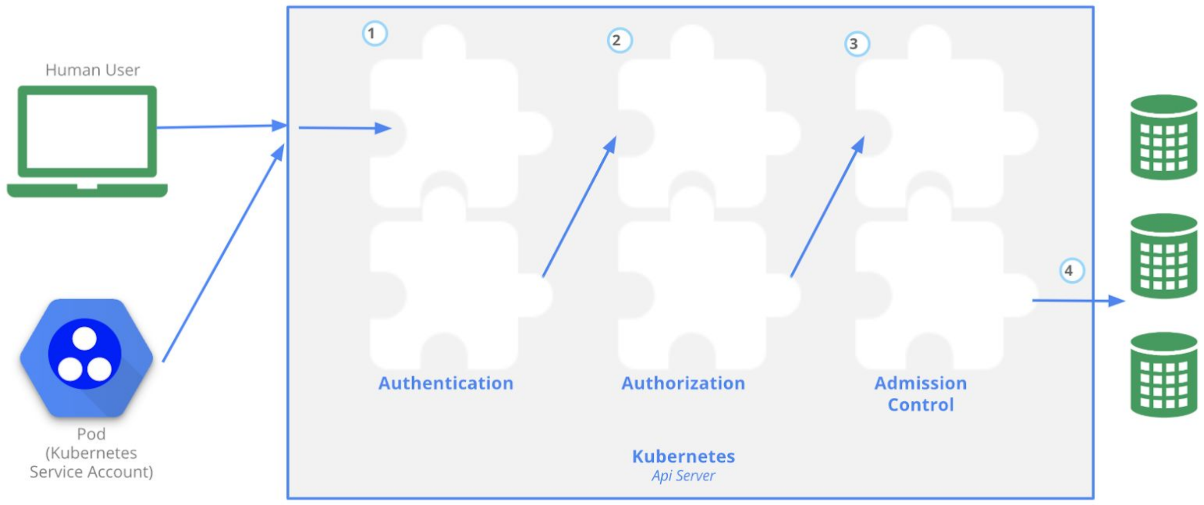

Figure 6. Kubernetes access control process

The first step is **Authentication**. **Authenticator modules** are run by the cluster admin or a creation script, which include client certificates, bearer tokens, HTTP basic auth, or authenticating proxy. Although the input is an entire HTTP request, the modules usually examine the headers and/or client certificate and attempt to associate a username, UID, groups, and extra fields with the request. At least two methods should be used--service account tokens for service accounts, and another method for user authentication (e.g. authenticating proxy). When multiple methods are enabled, the first one to successfully authenticate short circuits the rest. If the request cannot be authenticated, it is rejected with the HTTP status code 401. If it is successful, the user is authenticated as a specific `username`, and some authenticators also provide group memberships. Again, Kubernetes **does not** store “user” objects or information, and simply uses the `username` for access control.

The second step is **Authorization**. A request must include the username, requested action, and affected object in addition to other attributes like group, API request verb, and HTTP request verb; it is authorized if the existing authorization policy declares that the user has permissions for the action. If any module authorizes the request, it passes, but if a request is denied by all the modules, it is rejected with HTTP status code 403.

Kubernetes supports multiple **authorization modules**, which can be configured using the flag `--authorization-mode=<module>`, for example, `--authorization-mode=ABAC` or `--authorization-mode=Node`.
First, we have **role-based access control (RBAC)**. RBAC regulates access based on roles of individual users--a `Role` contains rules that represent a set of permissions in one namespace, while a `ClusterRole` applies cluster-wide. Next, we have **attribute-based access control (ABAC)**, where access rights are granted to users through the use of policies which combine attributes together. Request attributes correspond to properties of a “policy object”. Third, we have **Node authorization**, which authorizes API requests made by Kubelets. Kubelet can perform API operations: reads, writes, and authorization-related operations. Lastly, **Webhook mode** is a HTTP callback: an HTTP POST will be sent when something happens.

```
apiVersion: abac.auth.k8s.io
kind: Policy
spec:
  user: “bob”,
  namespace: “projCaribou”,
  resource: “pods”,
  readonly: true
```

```
apiVersion: auth.k8s.io
kind: SubjectAccessReview
spec:
  resourceAttributes:
    namespace: “projCaribou”,
    verb: “get”
    resource: “pods”
```

Example of a user policy (left) and a request (right)
If Bob has the policy on the left, his request will be authorized
because he can read objects in the `projCaribou` namespace.
If Bob makes a request to write (create, get), his request will be denied.
If Bob makes a request to objects in a different namespace, it will be denied.

The third step is **Admission Control**. **Admission Control Modules** are software modules that can modify or reject requests as they can access the contents of an object being created or updated. Control modules can be “validating” modules, which cannot modify objects, “mutating” modules, which can modify objects, or both. Mutating controllers check requests first, then validating controllers. Modules can be set using the flag `--admission-control=<list-of-modules>`, for example, `--admission-control=NamespaceLifecycle,ResourceQuota`. If any module rejects a request, it is immediately rejected.

The last step is validation. The request is validated using validation routines for their corresponding API object (Pod, Node, etc.) and then written to the object store.

Performance

When dealing with the deployment and maintenance of large scale applications, Kubernetes needs to be wary of performance issues and scalability to provide clients with a consistent and reliable container-orchestration system. A common technique to test performance of a cluster is to **aggregate runtime data** across Nodes and containers in the cluster, then compare them across times or against other similar clusters to see how the target cluster compares. Basic monitoring tools like Heapster, Prometheus, Grafana, InfluxDB, and CAdvisor can also report on ongoing operational behaviors and detect dangerous situations.

Aside from facilitating cluster performance monitoring, Kubernetes also boasts a strong and helpful community and team that have solved many problems in the past few years. Recently, a team at Airbnb encountered a performance issue while using Kubernetes. Since Kubernetes uses **multitenancy** (single instance serving multiple applications) for using the CPU, performance problems can arise, such as the Noisy Neighbors Problem, where a CPU-intensive process hogs CPU time and starves other applications from using the CPU. Kubernetes solves this by using the Linux kernel's CFS bandwidth control, which allocates CPU time to pre-defined groups, but this sometimes makes a Node appear slow even if nothing else is happening on the CPU. Since then, the Kubernetes community has implemented changes to combat these problems to make CRFS quota periods configurable or just disabling CPU quotas entirely.

The Kubernetes team also **monitors the performance** of clusters and detects potential regressions by creating a 100-node cluster every couple hours and running scalability tests on it. They focus on the full cluster case, where each Node has a full 30 Pods running, to test the most performance demanding scenario. The tests they run mimic what a user might do: populating Pods and replication controllers to fill the cluster, generating a load and recording performance measures, stopping all running Pods and replication controllers, and scraping metrics and checking if they match their expectations. From these tests, Kubernetes has identified hindrances to performance and implemented fixes.

A few years ago, the team set a goal of being able to run tests on 1000-node clusters instead of 100-node clusters and listed several possible improvements that included moving events out from etcd, using better json parsers, and rewriting the scheduler. They have since then met the 1000-node cluster goal by implementing several changes, the main one being creating a read cache at the API server level. The team also has weekly meetings for the Kubernetes Scale Special Interest Group where they discuss ongoing issues and plans for performance tracking and improvements.

Scale

An important reason why people use Kubernetes is because it scales efficiently and effectively. There are three main ways to scale an application in Kubernetes: **Horizontal Pod Autoscaler**, **Vertical Pod Autoscaler**, and **Cluster Autoscaler**. Each of these processes involves automatically detecting when an application needs more or less resources and then implements changes to counteract the imbalance of resources.

Horizontal Pod Autoscaling is usually triggered when the CPU or memory meets a certain threshold, and in response HPA changes the number of Pod replicas to scale the number of Pods in the cluster. **Vertical Pod Autoscaling** allocates more or less CPU or memory to existing Pods, which then takes effect when the Pods restart. Kubernetes automatically calculates suggested CPU and memory values based on historic measures. **Cluster Autoscaler** scales cluster Nodes based on pending Pods, which are Pods that are waiting to be assigned to a Node. If there are one or more Pods in the pending state due to lack of available resources on cluster, then CA attempts to provision one or more additional Nodes. When the Node is granted by the cloud provider (e.g., AWS, Azure, Google Cloud), the Node joins the cluster and becomes ready to serve the pending Pods.

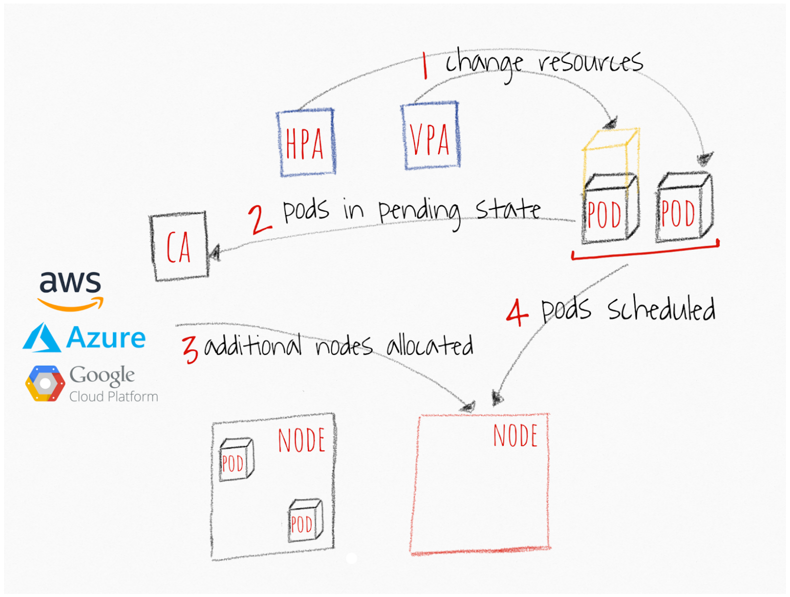

Figure 6. HPA, VPA, CA overview

Once an application becomes large, updating it becomes difficult. Users expect applications to be up and running all the time and developers need to deploy new versions of the application, sometimes a few times a day. Kubernetes facilitates a smooth updating process by using **rolling updates**, allowing a Deployment's update to occur with zero downtime by incrementally updating Pods. Each update is versioned to support reverting Deployments in case something goes wrong, also known as a **Rollback**. Rolling updates and scaling allow for easy promotion of an app from one environment to another, rollbacks to previous versions, and continuous integration and delivery of applications with zero downtime.

In addition to rolling updates, another feature that Kubernetes uses to facilitate the scaling process is **canary testing**, or **canary releases**. Updates to an application often contain bugs or performance issues, and detecting these bugs and performing damage recovery quickly makes canary testing an essential tool for Kubernetes users. Kubernetes uses canary testing to slowly roll out changes to a small subset of users before rolling it out to the entire platform. These tests can be automated and can send alerts so that any negative changes can be quickly reversed by routing traffic away from the canary or by rolling back the canary update.

Why use Kubernetes?

"Kubernetes solves a fundamental problem in distributed computing: how to keep large-scale, containerized applications that are subject to continuous change, both in terms of features and resource requirements, running reliably on the internet. Kubernetes solves the problem by using the concept of a predefined state to create, run and maintain container-driven applications over a cluster of physical or virtual machines." - Bob Reselman

Kubernetes is a largely supported computer system that is constantly being updated and worked on by a strong open source community. Kubernetes has its own **KubeCon** convention specifically for developers and users of Kubernetes with over 8000 people in attendance at the most recent event. Kubernetes also has its own Slack and Stack Overflow community and GitHub community repository.

Kubernetes is easier to use and has more users and growth than its alternatives, with many tutorials and comprehensive documentation to help users learn. Supported by major computing engines and cloud software including Google Cloud, Azure, and AWS, Kubernetes has also won the **Most Impact Award** at OSCON, and is **#1** in terms of activity on GitHub.

Sources

General

- Documentation Home: https://kubernetes.io/docs/home/
- Basic Tutorial: https://kubernetes.io/docs/tutorials/kubernetes-basics/
- GitHub Repo: https://github.com/kubernetes/kubernetes
    - https://godoc.org/k8s.io/kubernetes
- Community Repo: https://github.com/kubernetes/community
- Building Kubernetes: https://github.com/kubernetes/kubernetes/tree/master/build

Articles

- https://kubernetes.io/blog/2018/07/20/the-history-of-kubernetes-the-community-behind-it/
- https://kubernetes.io/docs/setup/learning-environment/minikube/
- https://thenewstack.io/kubernetes-an-overview/
- https://coreos.com/kubernetes/docs/latest/pods.html

Basics

Figure 1:
https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/
Figure 2: https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/

Naming

Figure 3: https://medium.com/faun/kubernetes-pod-naming-convention-78272fcc53ed

- https://kubernetes.io/docs/concepts/overview/working-with-objects/names/
- https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/
- https://stackoverflow.com/questions/46204504/kubernetes-pod-naming-convention
- https://github.com/kubernetes/community/blob/master/contributors/design-proposals/architecture/identifiers.md
- https://github.com/kubernetes/kubernetes/blob/2183a84feb564582489d56c6570917959f003726/staging/src/k8s.io/apimachinery/pkg/util/rand/rand.go#L80-L90

Caching

Figure 4: https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns/

- https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/dns/nodelocaldns

Resource Management

- https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/#node-allocatable
- https://kubernetes.io/docs/concepts/overview/components/#master-components
- https://kubernetes.io/docs/concepts/architecture/controller/
- https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/
- https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/
- https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/
- https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/
- https://github.com/kubernetes/community/blob/master/contributors/design-proposals/node/resource-qos.md
- https://kubernetes.io/blog/2018/07/24/feature-highlight-cpu-manager/
- https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/
- https://kubernetes.io/docs/concepts/services-networking/endpoint-slices/
- https://kubernetes.io/docs/concepts/services-networking/service/
- https://kubernetes.io/docs/concepts/storage/volumes/
- https://kubernetes.io/docs/concepts/storage/persistent-volumes/

Virtualization

Figure 5: https://kubernetes.io/docs/concepts/overview/components/

Access Control

- https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/
- https://kubernetes.io/docs/reference/access-authn-authz/authorization/
- https://github.com/kubernetes/kubernetes/tree/master/plugin/pkg/auth
- https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/
- https://kubernetes.io/docs/reference/access-authn-authz/authentication/

Performance and Scale

Figure 6:
https://medium.com/magalix/kubernetes-autoscaling-101-cluster-autoscaler-horizontal-pod-autoscaler-and-vertical-pod-2a441d9ad231

- https://thenewstack.io/kubernetes-performance-troublespots-airbnbs-take/
- https://www.getambassador.io/docs/dev-guide/canary-release-concepts/
- https://blog.gurock.com/kubernetes-performance-testing/
- https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#canary-deployments