KOENIG

step forward

Containers

Containers

- Introduction to Containers
- Containers vs. Virtual Machines (VMs)
- How container components interacts
- Basic Docker Commands

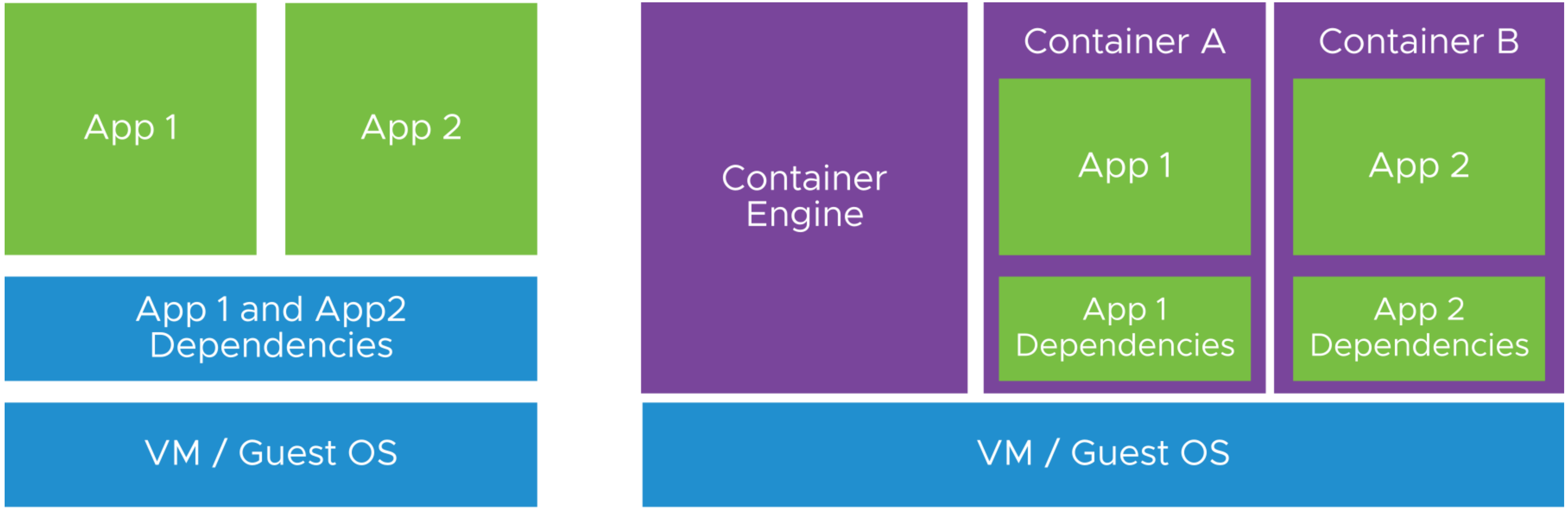
Introduction to Containers

- What Are Containers?
  - Containers are lightweight, portable, and self-sufficient environments that package applications with all their dependencies.
  - They ensure that software runs consistently across different environments.
- Key Features of Containers
  - **Lightweight**: Share the host OS kernel, reducing overhead.
  - **Portable**: Run the same container across different platforms (e.g., local, cloud, on-prem).
  - **Isolated**: Each container has its own file system, processes, and network.
  - **Fast & Scalable**: Start quickly and scale efficiently.

Virtual Machines (VMs) vs. Containers

- Each **VM** provides virtual hardware that the guest OS uses to execute applications.
- Multiple applications run on a single physical server while still being logically **separated and isolated**.
- With containers, developers take a streamlined base OS file system and layer on only the required binaries and libraries on which the application depends.

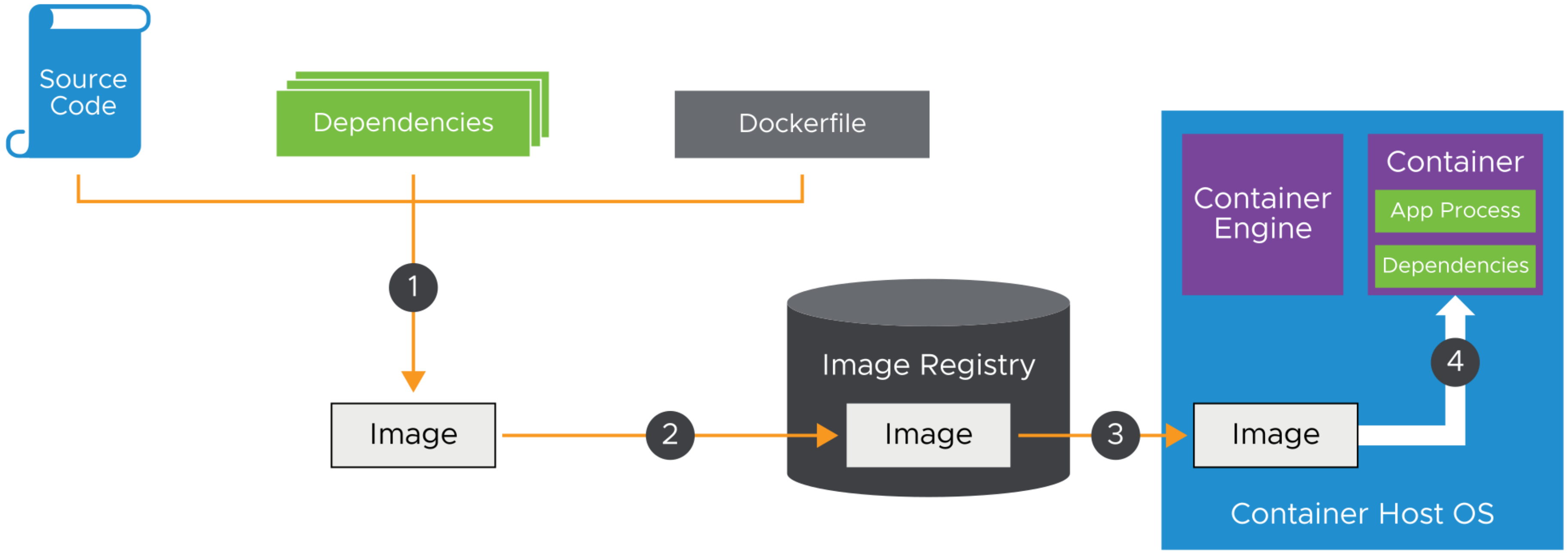
Virtual Machines (VMs) vs. Containers



Feature	Virtual Machines	Containers
OS	Each has a full OS	Share host OS
Performance	Slower, heavier	Faster, lightweight
Resource Usage	Higher	Lower
Startup Time	Minutes	Seconds

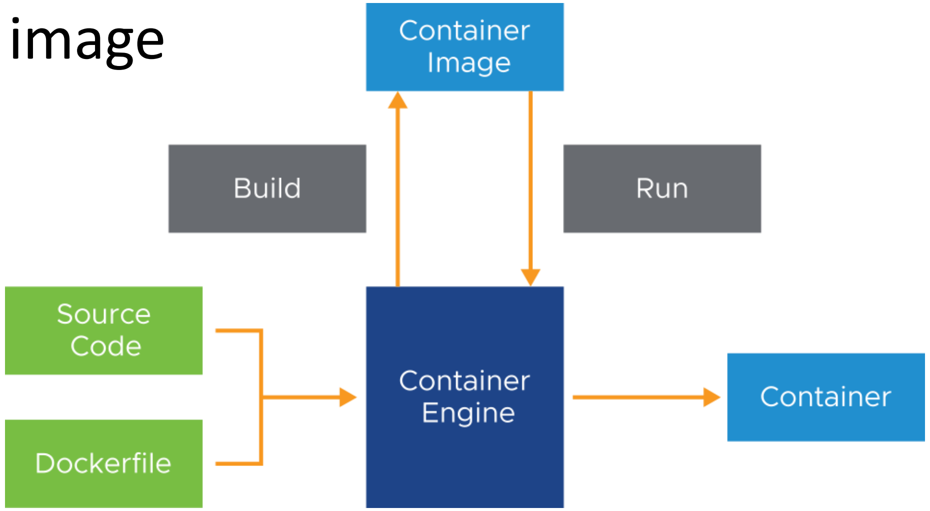
How container components interacts

- A container workflow follows these steps:
  1. Build an image from the source code and dependencies.
  2. Push the image to the image registry.
  3. Pull the image from the image registry.
  4. Run the image as a container.



Container Components: Container Engines

- A container engine (also called a container runtime) is a control plane that is installed on each container host.
- The control plane manages the containers on that host.
- Container engines perform the following functions:
  - Build container images from source code (for example, Dockerfile) or load container images from a repository
  - Create running containers based on a container image
  - Commit a running container to an image
  - Save an image and push it to a repository
  - Stop and remove containers
  - Suspend and restart containers
  - Report container status



- Docker is the most common container engine.

Container Components: Dockerfile

- Dockerfile is a plain text file that declares how to create an image.
- Dockerfile is similar to the source code of an image.
- You can use the BUILD command to create an image from Dockerfile.

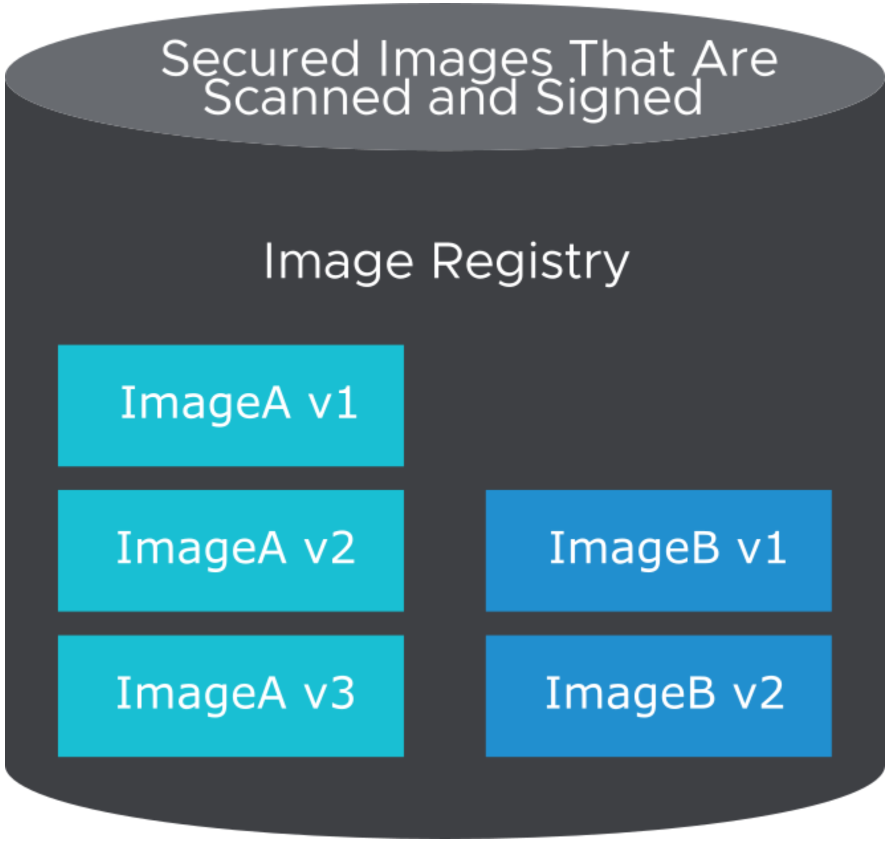
```
FROM nginx:alpine
COPY html/index.html /usr/share/nginx/html/index.html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Container Components: Container Images

- Container images are comparable to VM templates.
- They have the following characteristics:
  - Container images contain application code and application dependencies.
  - Container images are built using a layered file system and can consist of one or more layers.
  - Docker images are built using a Dockerfile, and each line in a Dockerfile represents a layer in a container image.

Container Components: Image Registry

- Container images are stored in a central image registry:
  - The image registry maintains multiple versions of container images.
  - Container engines can take images from an image registry to run them.
  - Docker Hub is the most common public registry.



Docker Images Vs Containers

- A Docker image is the result of a build.
- A container is a running instance from an image.

Basic Docker Commands

- Command to create an image from Dockerfile.
  - $ docker build
- Command to create and start a container.
  - $ docker run
- Command to stop a running container
  - $ docker stop
- Command to delete a stopped container
  - $ docker rm
- Command to delete a running container
  - $ docker rm –f

Basic Docker Commands

- Command to list all images
  - $ docker images
- Command to list all running containers
  - $ docker ps
- Command to list all containers (running or paused or stopped)
  - $ docker ps -a
- Command to display logs
  - $ docker logs
- Command to run a command within a container
  - $ docker exec

Kubernetes, Performance, and Scalability

Kubernetes, Performance, and Scalability

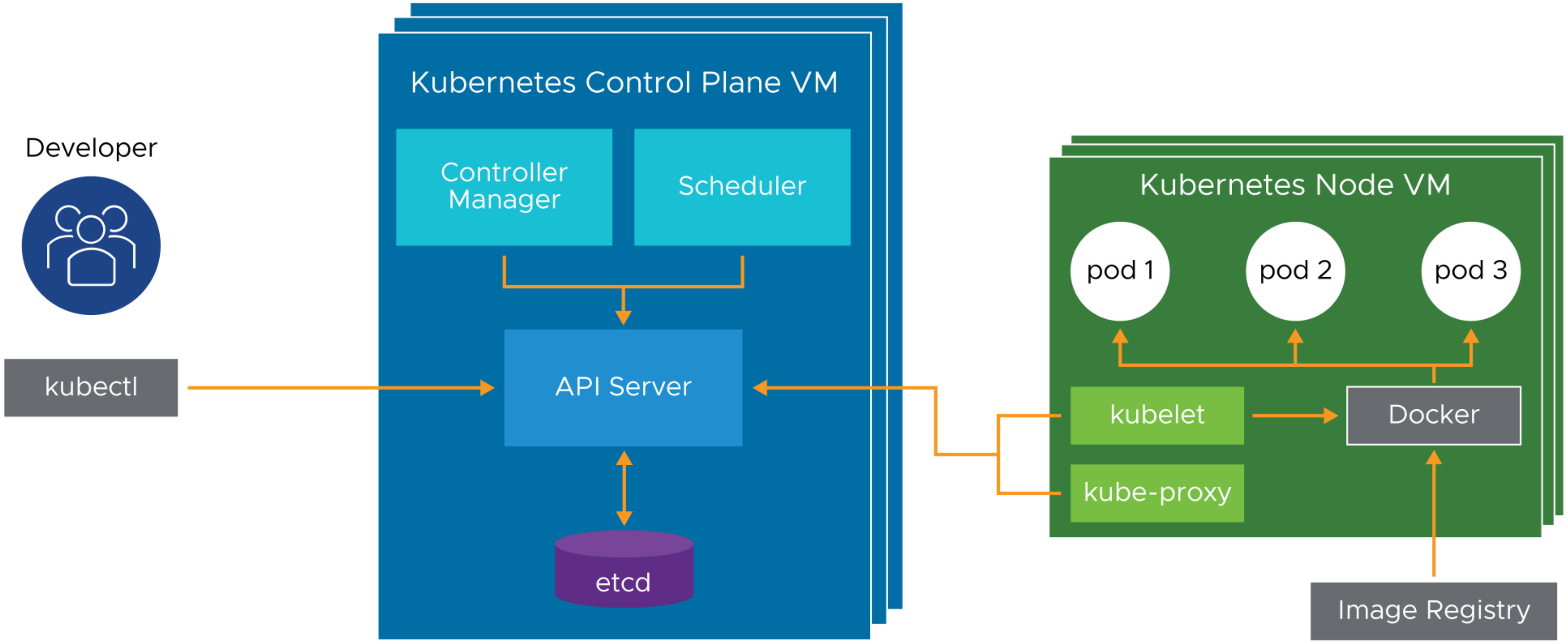
- Importance of Kubernetes
- Introduction to Kubernetes
- Basic Architecture of Kubernetes
- Basic Kubernetes Workflow
- Basic kubectl Commands
- Performance Optimization
- Scalability – Auto Scaling

Importance of Kubernetes

- With Docker, containers are managed on a single container host.
- Managing multiple containers across multiple container hosts creates many problems:
  - Managing large numbers of containers
  - Restarting failed containers
  - Scaling containers to meet capacity
  - Networking and load balancing
- Kubernetes provides an orchestration layer to solve these issues.
- Kubernetes (K8s) is an open-source container orchestration platform for automating the deployment, scaling, and management of containerized applications.

Basic Architecture of Kubernetes

- The Kubernetes architecture consists of several components.



Kubernetes - Core Concepts

- Cluster Components
  - Master Node (Control Plane): Manages the cluster.
    - API Server: The front-end for Kubernetes.
    - Scheduler: Assigns workloads to worker nodes.
    - Controller Manager: Ensures desired state.
    - etcd: A distributed key-value store for cluster data.
  - Worker Nodes: Run the applications.
    - Kubelet: Ensures containers are running.
    - Kube Proxy: Manages networking.
    - Container Runtime: Runs containers (e.g., Docker, containerd).

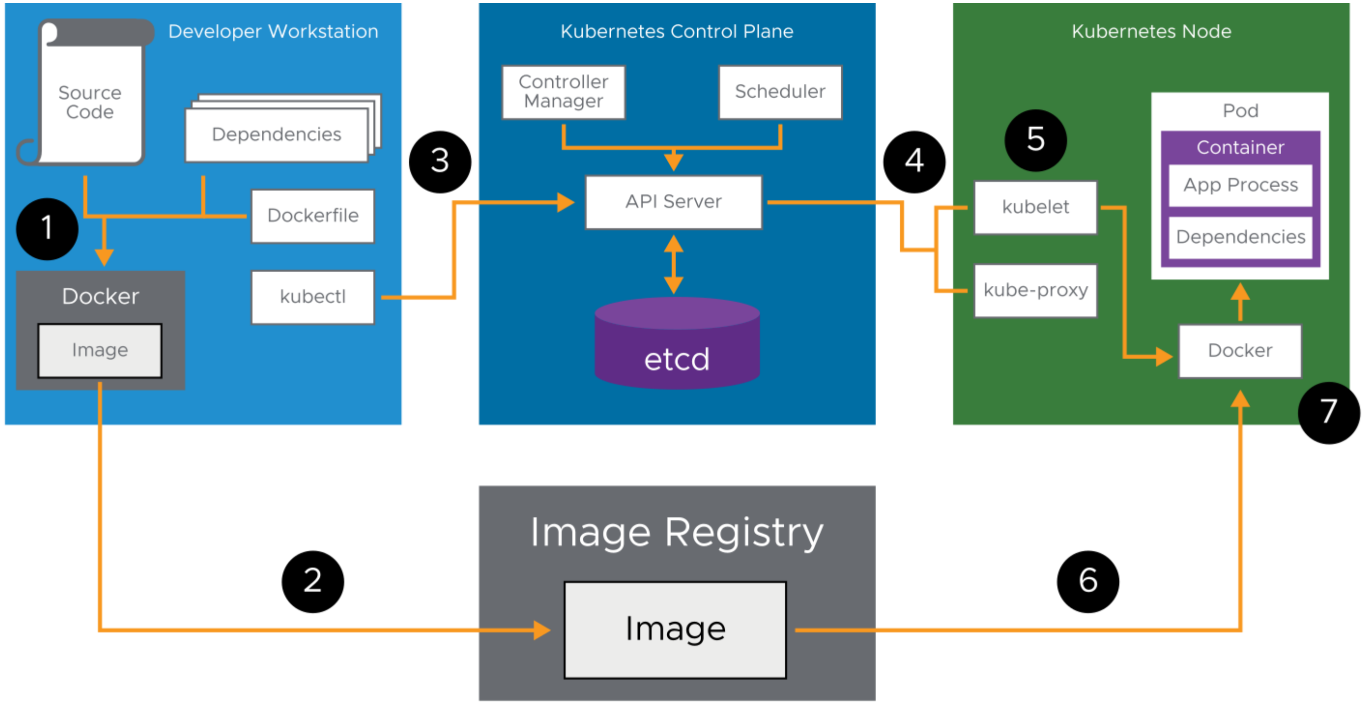
Kubernetes – Key Objects

- Pod
  - The smallest deployable unit.
  - Contains one or more containers sharing storage and networking.
- Deployment
  - Manages replicas of Pods.
  - Supports rolling updates.
- Service
  - Exposes a set of Pods as a network service.
  - Types: ClusterIP (internal), NodePort, and LoadBalancer.

Basic Kubernetes Workflow

Example workflow steps:

1. Build an image from source code and dependencies.
2. Send the image to the image registry.
3. Instruct Kubernetes to use the image to run a pod.
4. The scheduler assigns the pod to a node.
5. The kubelet accepts the pod.
6. The container engine (for example, Docker) takes the image from the image registry.
7. The container engine starts the container process inside a pod.



Kubernetes Basic Commands

- Check Kubernetes Version
  $ kubectl version
- Check Cluster & Node Status
  $ kubectl get nodes
- List All Running Pods
  $ kubectl get pods
- Describe Pod Details
  $ kubectl describe pod <pod-name>
- Apply a Configuration File
  $ kubectl apply -f <file-name.yaml>

Kubernetes Basic Commands

- Create a Pod (Imperative Method)

  $ kubectl run <pod-name> --image=<image-name>

- Delete Resources

  $ kubectl delete <resource-type> <resource-name>

- View Logs of a Pod

  $ kubectl logs <pod-name>

Performance

Performance

- To ensure Kubernetes clusters run efficiently, you need to optimize resource usage, networking, and scheduling.
- Optimize Resource Requests and Limits
  - Each container should have properly set resource requests and limits for CPU and memory.
- Enable Monitoring & Logging
  - Use Prometheus + Grafana for monitoring.
  - Use Fluentd, Loki, or ELK Stack for logging.
  - Enable Kubernetes Metrics Server for real-time resource metrics.

Performance

- Use Efficient Storage
  - Choose the right Persistent Volume (PV) type based on workload (SSD for high IOPS).
  - Enable ReadWriteMany (RWX) volumes for parallel access.
  - Use local storage for fast data access.
- Reduce API Server Load
  - Enable API request throttling to prevent overload.
  - Use Kubernetes Event Rate Limits to reduce logging overhead.
  - Optimize controllers to avoid excessive API calls.

Scalability - Auto Scaling

Scalability - Auto Scaling

- Scaling in Kubernetes ensures applications handle traffic spikes efficiently while optimizing resource utilization.
- Kubernetes provides **Horizontal Pod Autoscaler (HPA)** and **Vertical Pod Autoscaler (VPA)** for scaling Pods, and **Cluster Autoscaler** for scaling worker nodes.
- Horizontal Pod Autoscaler (HPA)
  - HPA automatically scales Pods based on CPU or memory utilization.
- Vertical Pod Autoscaler (VPA)
  - VPA automatically adjusts CPU and memory requests of Pods.
- Cluster Autoscaler
  - Cluster Autoscaler adds or removes worker nodes in cloud environments (AWS, GCP, Azure).

Happy Learning :)