Docker

Containers and Docker

Containers and Docker

Section overview

1 Why use containers?

1. Compare how deployments happen with and without containers
2. Understand the main benefits of using containers

2 Containers and Virtual Machines

1. Compare and highlight the differences between using containers and VMs for running workloads

3 Docker components

1. Discuss the different components from a Docker system
2. Understand how these components interact when running common operations in Docker

Docker

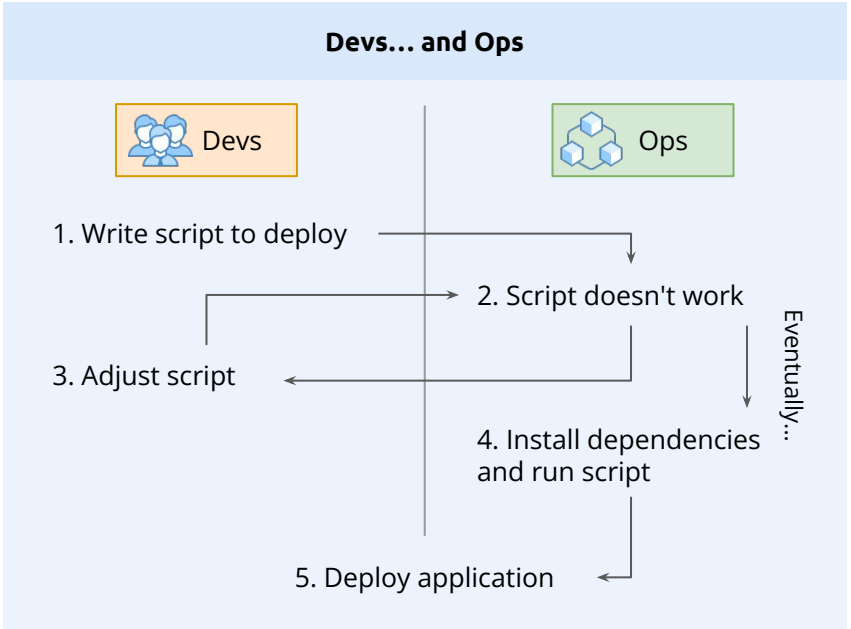
Why Containers?

Why Containers?

How did we build and deploy applications, really?

- What do we need to deploy a NodeJS app?
  - Install NodeJS dependencies
  - Install NodeJS itself
  - Install the app's dependencies
  - Run the application
- How about with other programming languages?

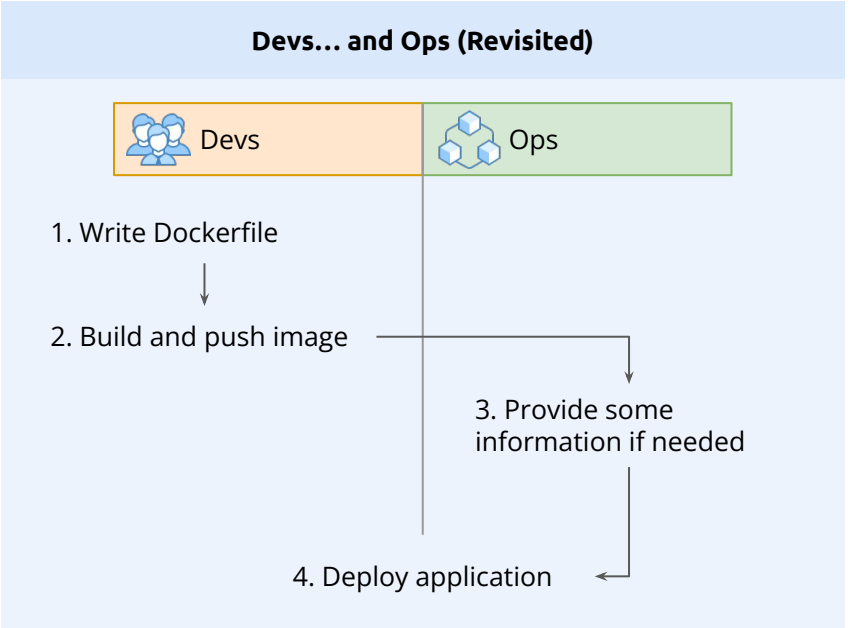
- How about with other NodeJS versions?
- Not to mention managing multiple applications running side by side...



Why Containers?

How do containers come to the rescue?

- Containers encapsulate all the dependencies and configuration necessary to run whatever application. From the outside, they all look the same and are all run (almost) the same way. This leads to, among others:
  - Simplified setup
  - Portability
  - Consistent environments
  - Isolation
  - Efficiency
  - Better resource control
  - Easily scalable applications

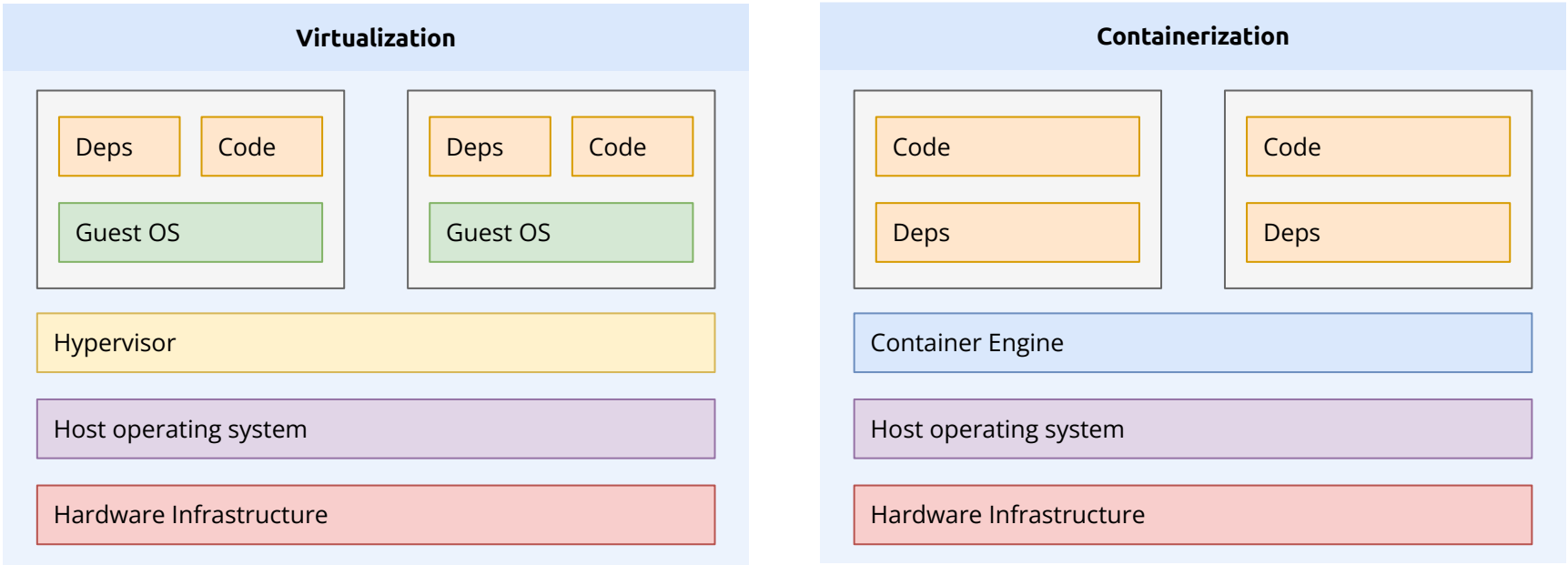


Docker

Containers and Virtual Machines

Containers and Virtual Machines

Containers are not the only solution



Containers and Virtual Machines

Containers are not the only solution

Virtualization

Containerization

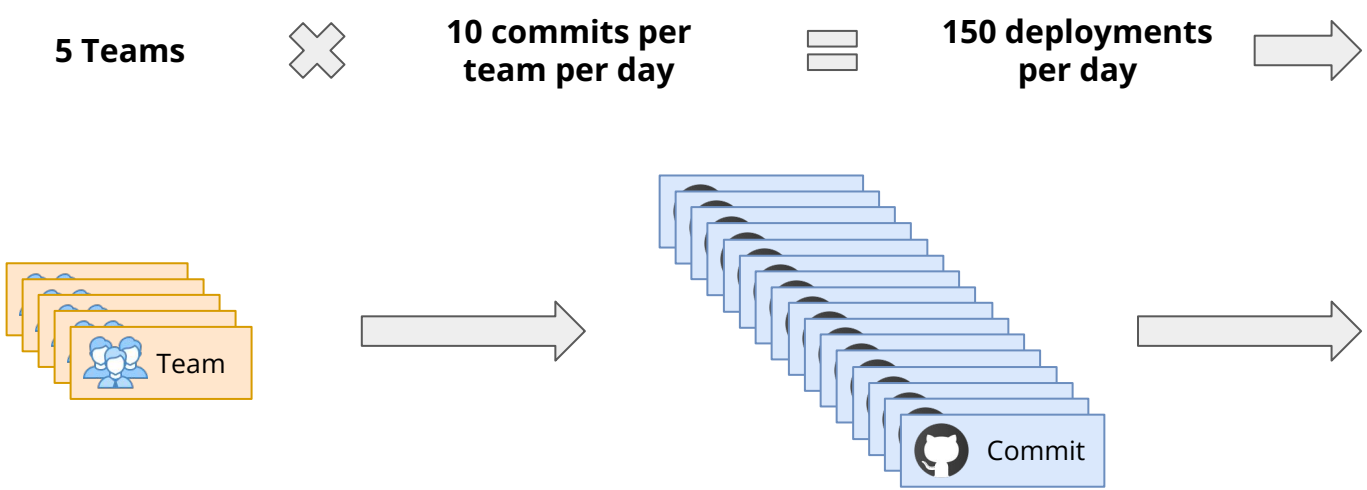
Containers and Virtual Machines

Containers are not the only solution

Feature	Virtual Machines (VMs)	Docker Containers
Isolation	**Strong isolation:** Each VM has its own OS, providing complete isolation.	**Process-level isolation:** Containers share the host OS kernel.
Size/Overhead	**Larger:** VMs have a larger footprint due to the guest OS and virtual hardware.	**Lightweight:** Containers have minimal overhead, as they share the kernel.
Portability	**Less portable:** VMs can be tied to specific hypervisors and guest OS configurations.	**Highly portable:** Containers are platform-agnostic and run consistently.
When to use		
	▪ You need strong isolation between different environments. ▪ You're dealing with legacy applications that might not be easily containerized. ▪ You want to replicate a complete system environment for testing or development.	▪ You're building modern, cloud-native applications using microservices architecture. ▪ You need to scale your applications quickly and efficiently. ▪ Portability across different environments is a top priority.

Containers and Virtual Machines

How much time can we save?



Time savings

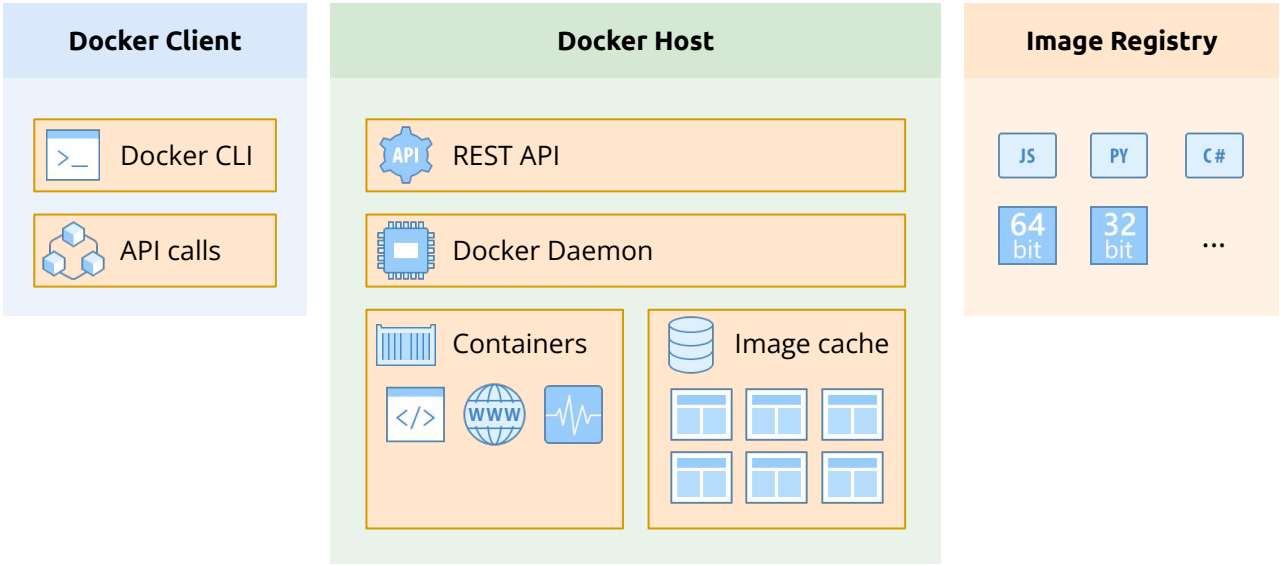
- 1-minute difference:
  150 minutes/day
  2.5 hours/day
- 2-minute difference:
  300 minutes/day
  5 hours/day
- ...

Docker

Docker Components

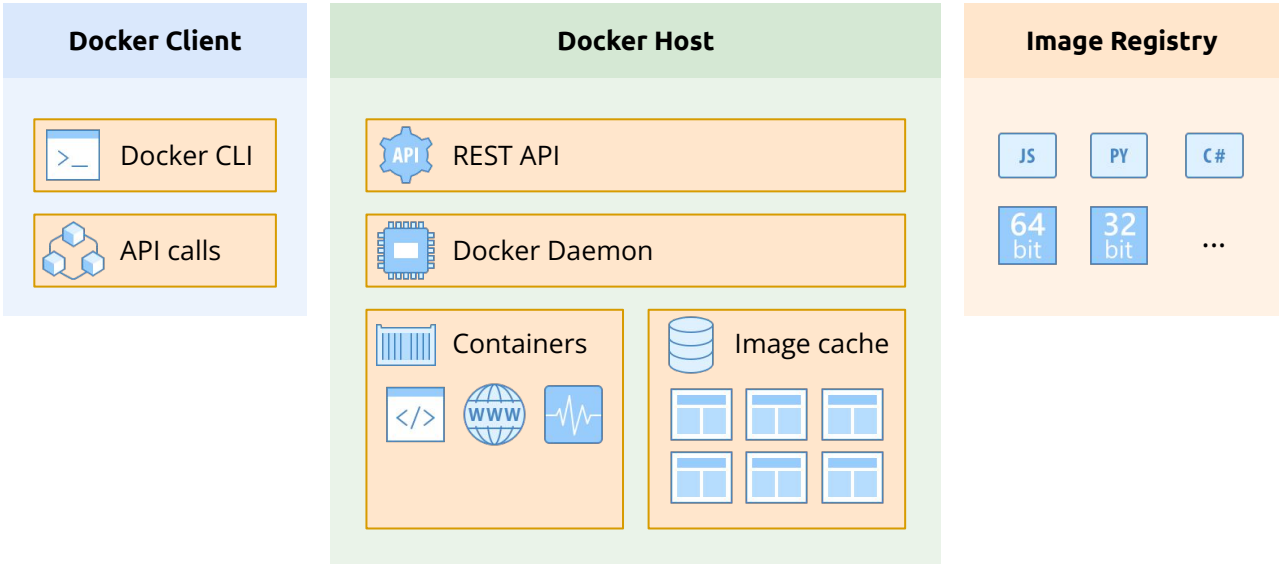
Docker Components

What are the different parts in a Docker-based system?



Docker Components

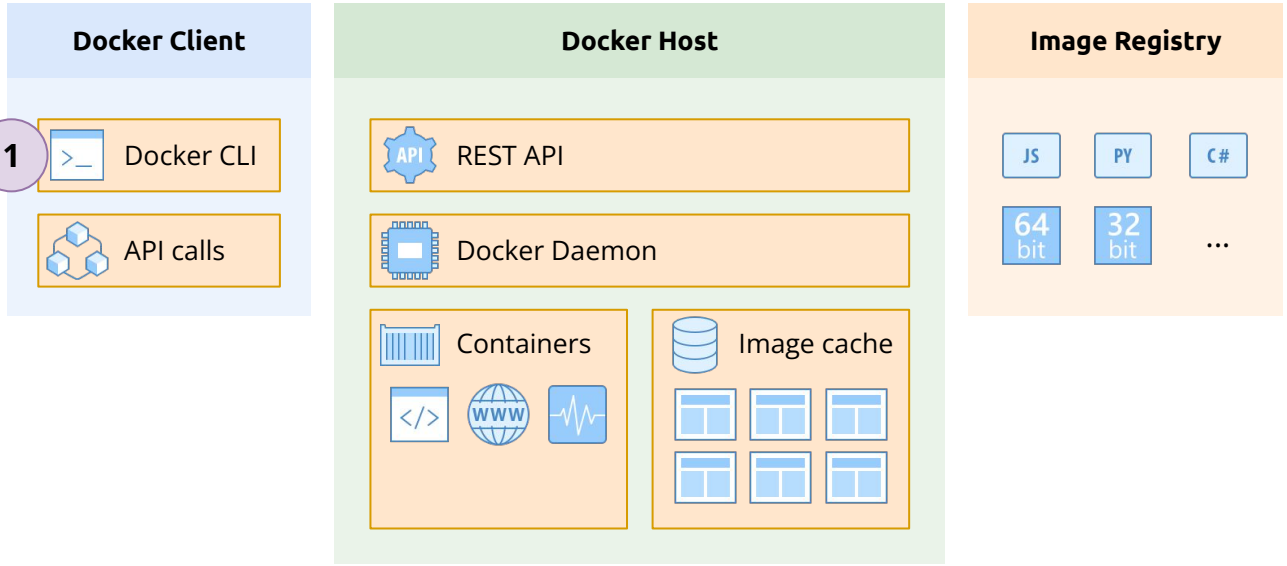
What are the different parts in a Docker-based system?



Running a container from an image

Docker Components

What are the different parts in a Docker-based system?

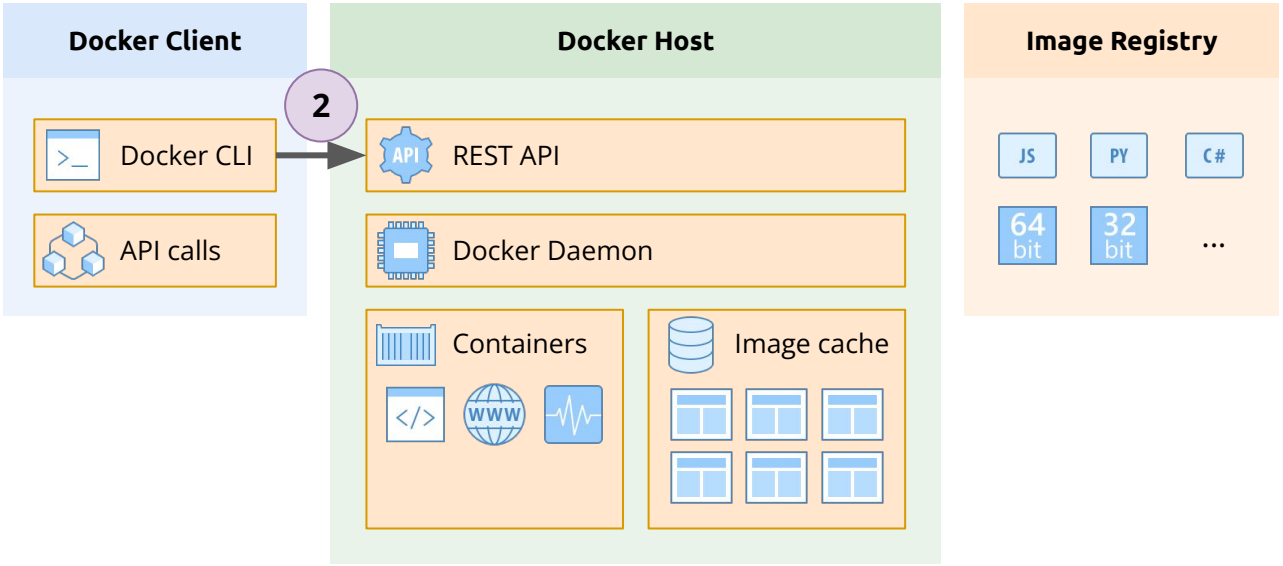


Running a container from an image

1. Issue `docker run` in the CLI.

Docker Components

What are the different parts in a Docker-based system?

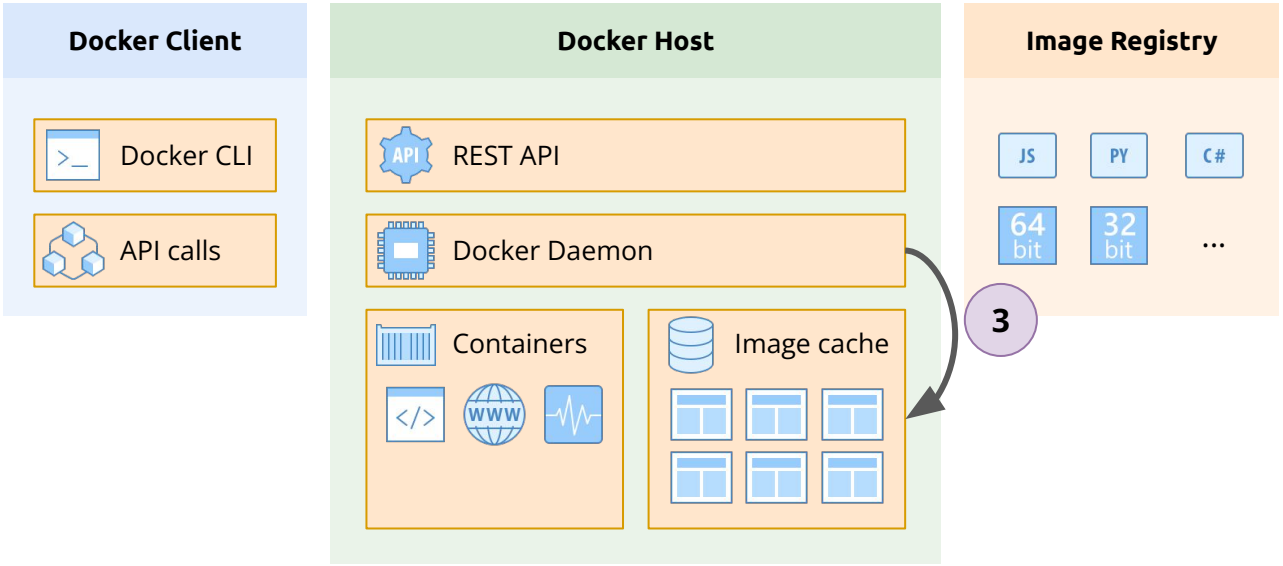


Running a container from an image

1. Issue `docker run` in the CLI.
2. CLI sends a request to the host's REST API.

Docker Components

What are the different parts in a Docker-based system?

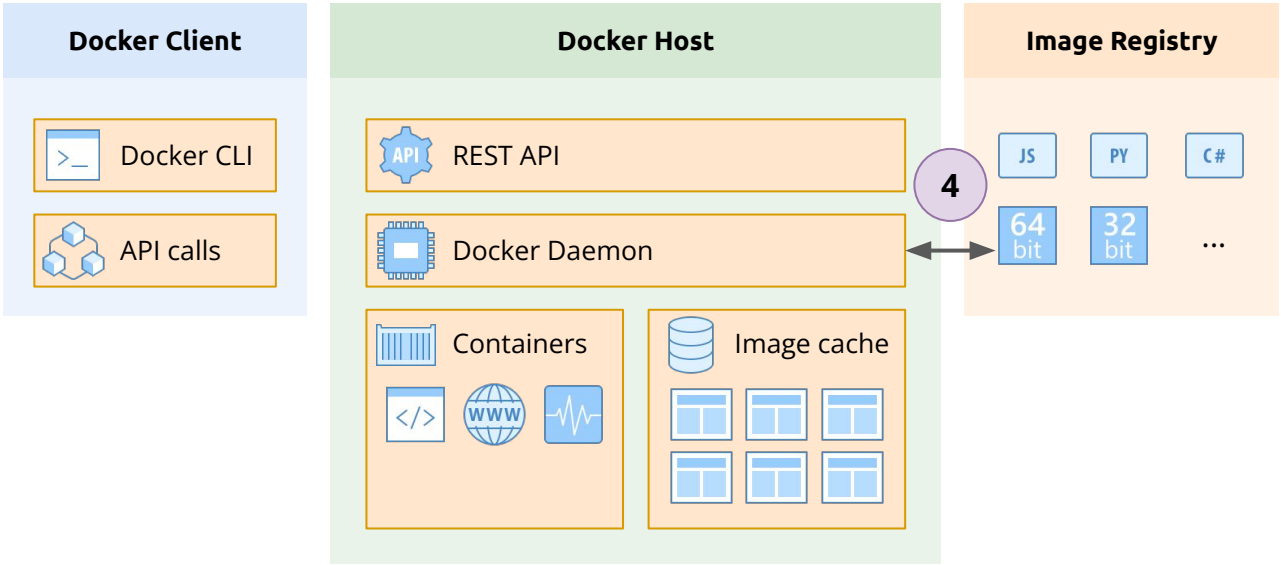


Running a container from an image

1. Issue `docker run` in the CLI.
2. CLI sends a request to the host's REST API.
3. Docker Host checks if the image is present in the local cache.

Docker Components

What are the different parts in a Docker-based system?

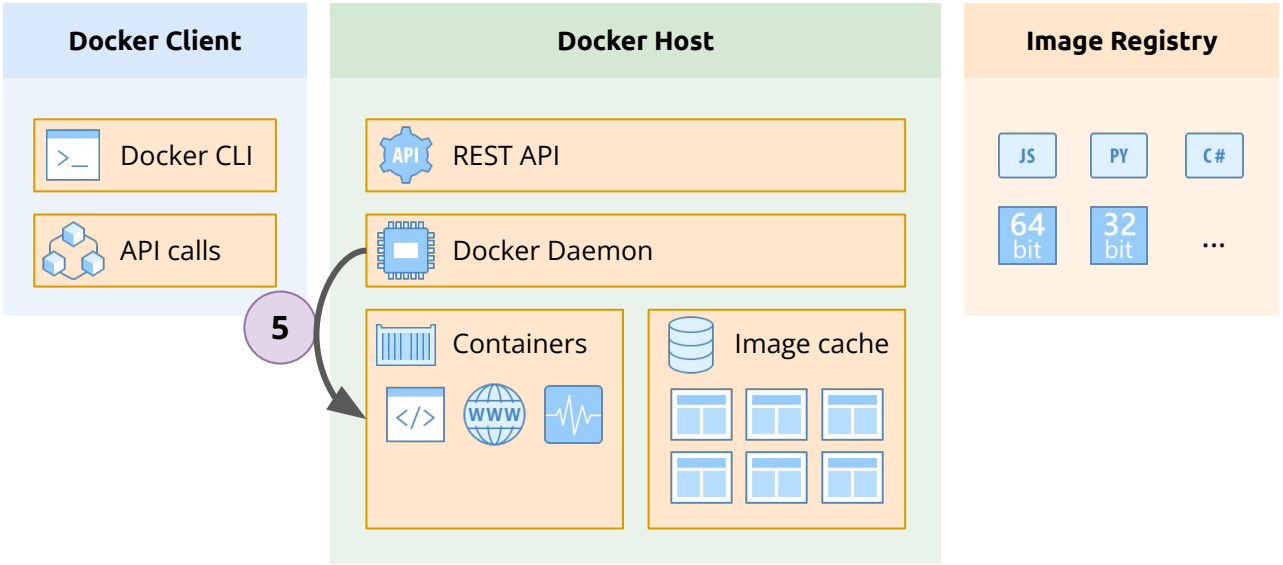


Running a container from an image

1. Issue `docker run` in the CLI.
2. CLI sends a request to the host's REST API.
3. Docker Host checks if the image is present in the local cache.
4. If not, it downloads it from the image registry.

Docker Components

What are the different parts in a Docker-based system?

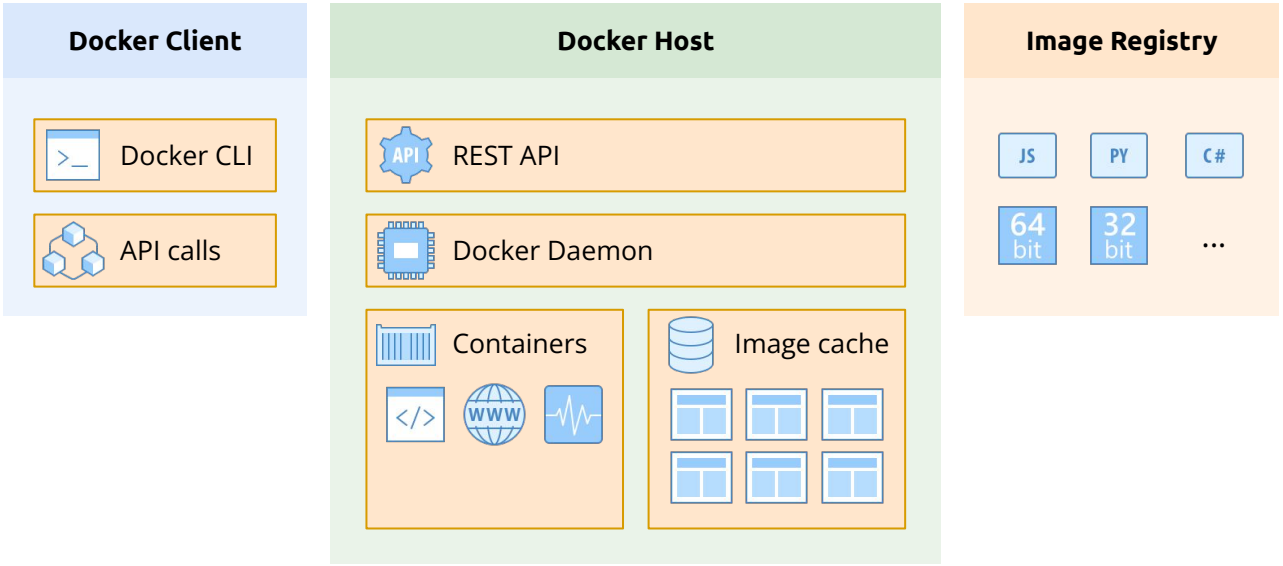


Running a container from an image

1. Issue `docker run` in the CLI.
2. CLI sends a request to the host's REST API.
3. Docker Host checks if the image is present in the local cache.
4. If not, it downloads it from the image registry.
5. Docker Host instantiates a new container based on the image.

Docker Components

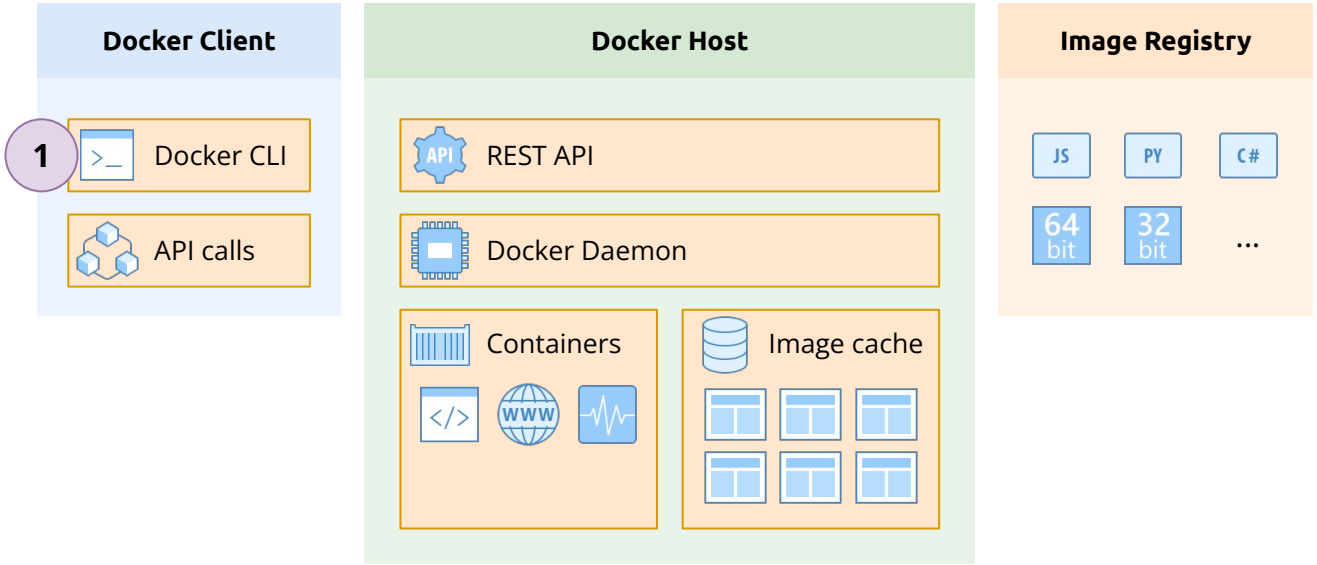
What are the different parts in a Docker-based system?



Building and pushing an Image

Docker Components

What are the different parts in a Docker-based system?

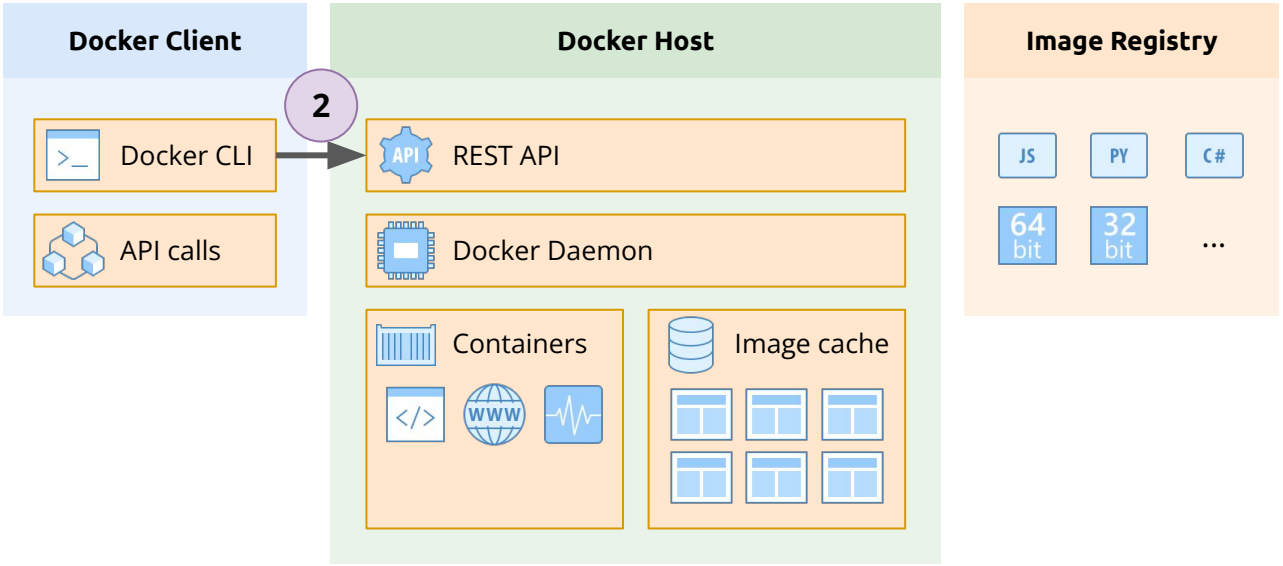


Building and pushing an Image

1. Issue `docker build` in the CLI.

Docker Components

What are the different parts in a Docker-based system?

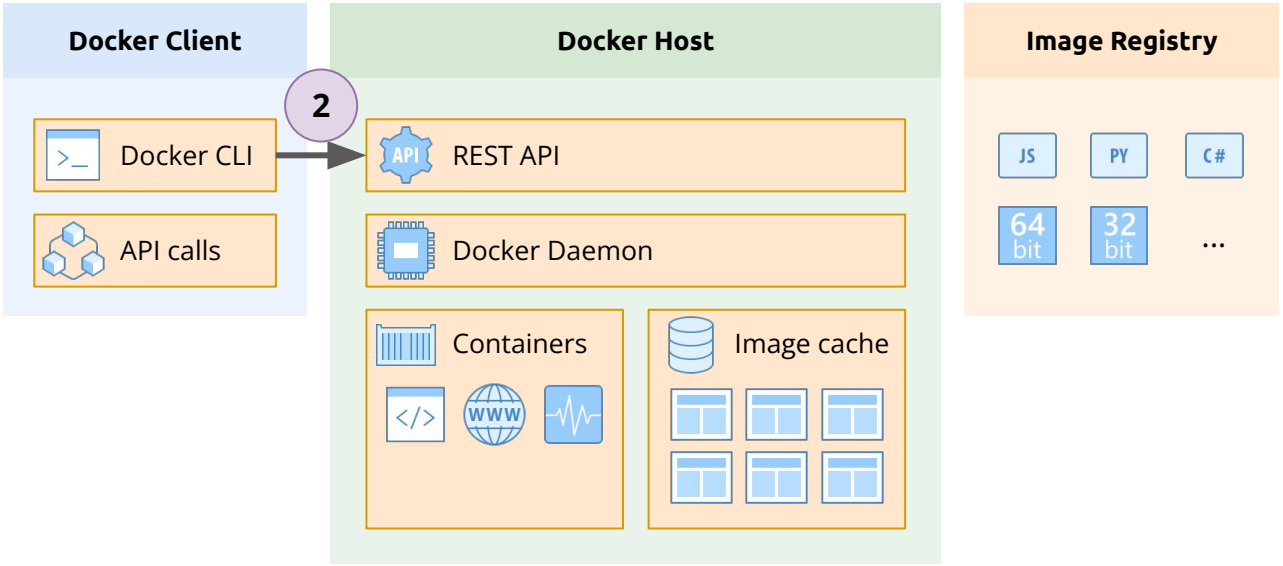


Building and pushing an Image

1. Issue `docker build` in the CLI.
2. CLI sends a request to the host's REST API.

Docker Components

What are the different parts in a Docker-based system?

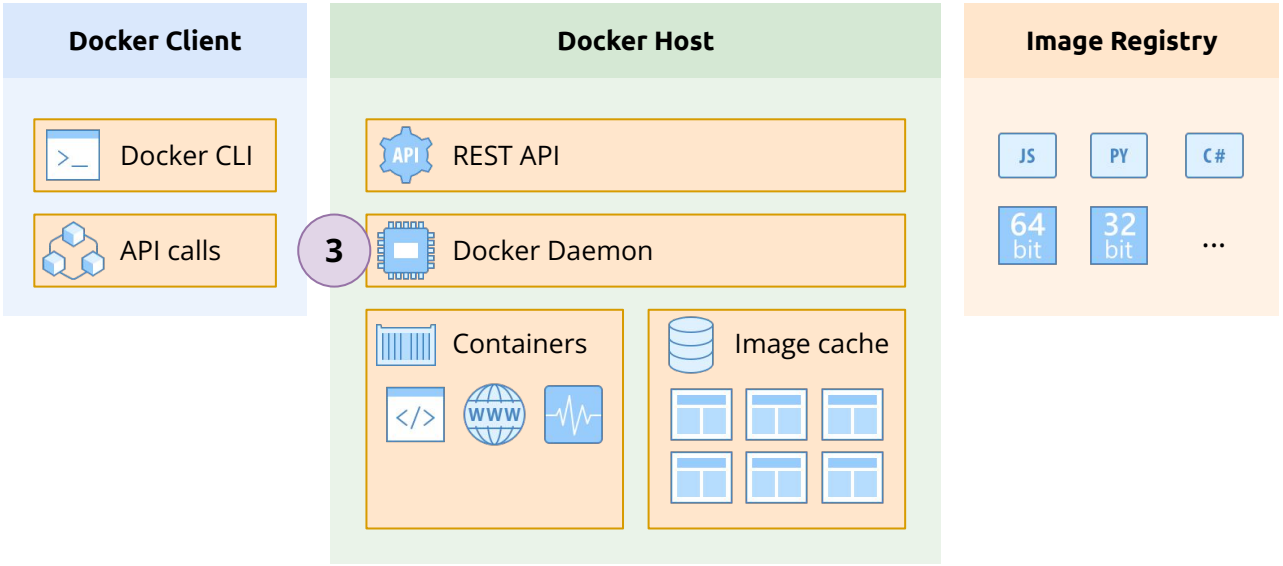


Building and pushing an Image

1. Issue `docker build` in the CLI.
2. CLI sends a request to the host's REST API.
   a. This also includes the respective Dockerfile and context.

Docker Components

What are the different parts in a Docker-based system?

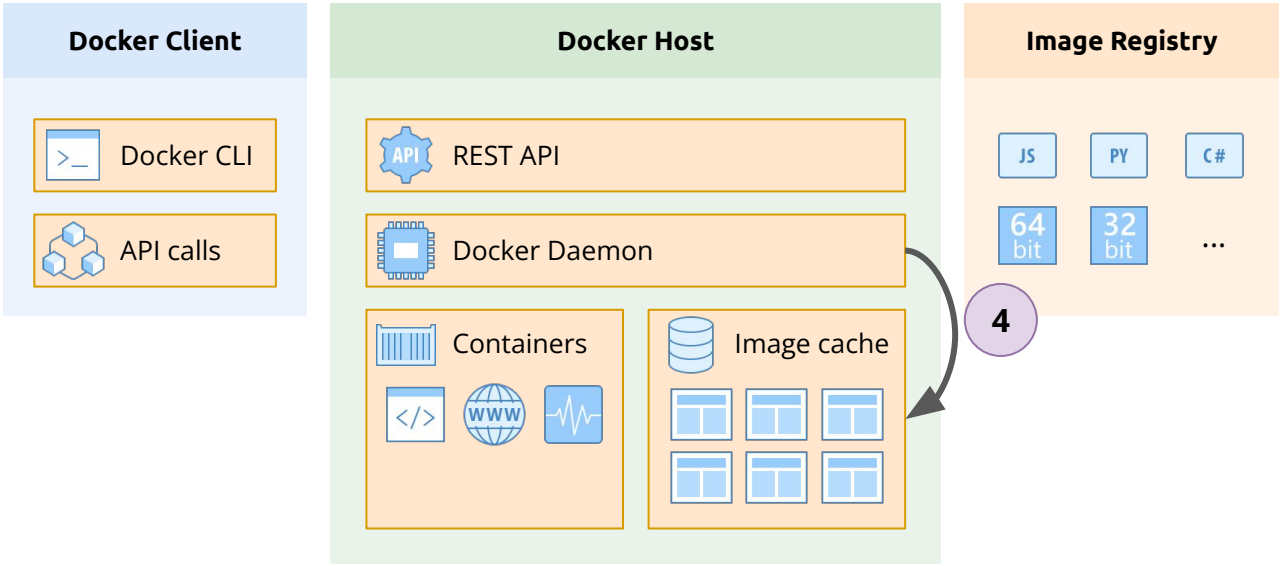


Building and pushing an Image

1. Issue `docker build` in the CLI.
2. CLI sends a request to the host's REST API.
   a. This also includes the respective Dockerfile and context.
3. Docker Host builds the image according to the Dockerfile.

Docker Components

What are the different parts in a Docker-based system?

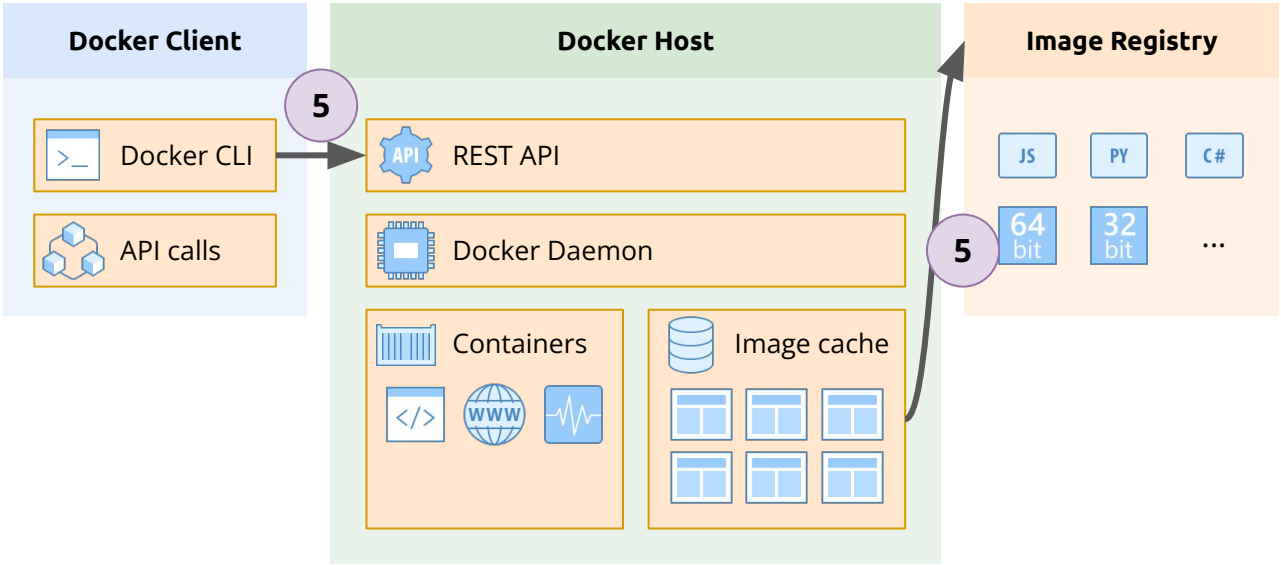


Building and pushing an Image

1. Issue `docker build` in the CLI.
2. CLI sends a request to the host's REST API.
   a. This also includes the respective Dockerfile and context.
3. Docker Host builds the image according to the Dockerfile.
4. Docker Host tags the image and stores it locally.

Docker Components

What are the different parts in a Docker-based system?



Building and pushing an Image

1. Issue `docker build` in the CLI.
2. CLI sends a request to the host's REST API.
   a. This also includes the respective Dockerfile and context.
3. Docker Host builds the image according to the Dockerfile.
4. Docker Host tags the image and stores it locally.
5. Issue `docker push` command on the CLI.

Docker

Installing Tools

Installing Tools

Section overview

1. Install Docker on Windows
    1. Setting up Windows Subsystem for Linux (WSL)
    2. Installing and configuring Docker Desktop
2. Install Docker on macOS
    1. Installing and configuring Docker Desktop
3. Install Docker on Linux
4. Explore online playgrounds
5. Install Node and Postman

Docker

Running Containers

Running Containers

Section overview

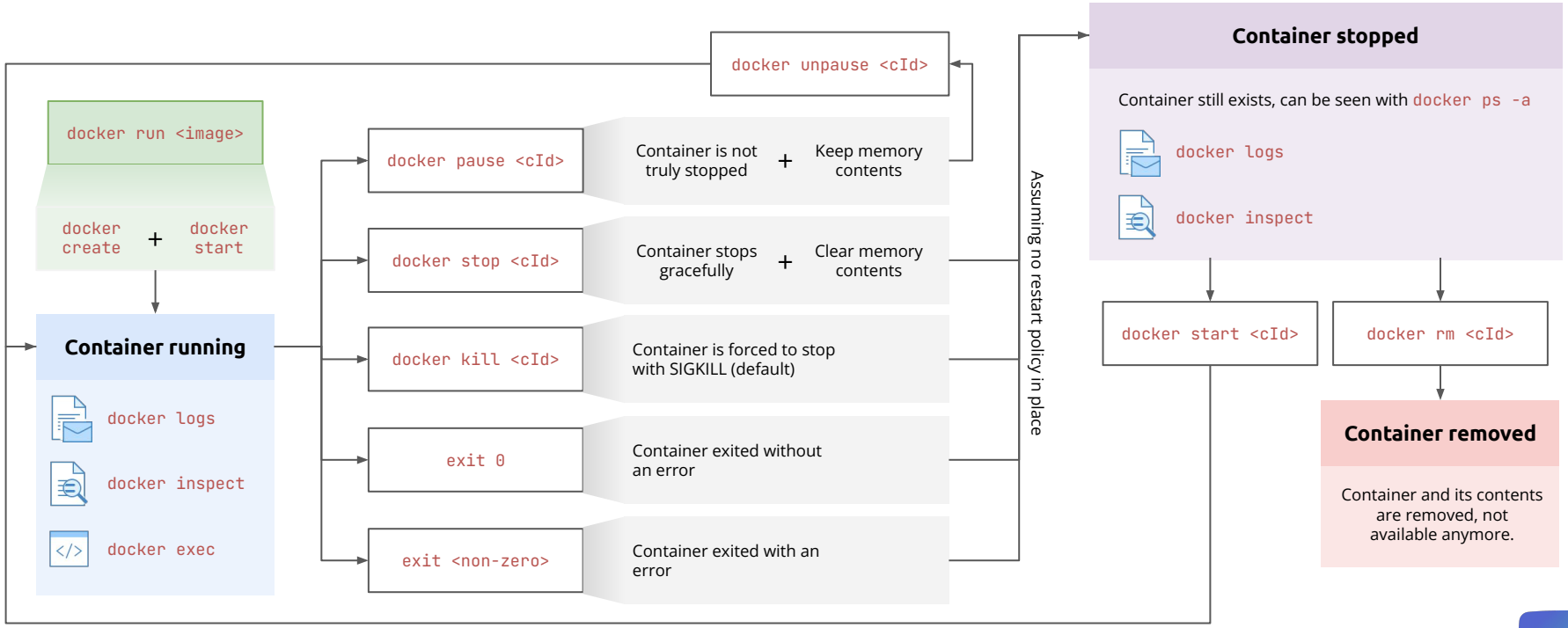
1. Run your first container
2. Understand the container lifecycle
    1. Discuss the different states a container can be in
    2. Understand which CLI commands result in which states
3. Explore the Docker CLI
    1. Demonstrate essential Docker CLI commands for managing containers and images.
    2. Explain container behavior and Docker's management of short-lived vs. persistent containers.
    3. Demonstrate commands for logging, running commands within containers, and executing shells.
    4. Introduce the Docker build process by creating a simple Dockerfile and building a custom Docker image.
4. Get help with the CLI

Docker

Container Lifecycle

Container Lifecycle

Understand what happens with containers throughout their lifetime



Docker

Working with Images

Working with Images

Section overview

1. Understand Docker images and their purpose
2. Explore container registries
    1. Understand the purpose of container registries, and the different options available
    2. Learn how to browse images in Docker Hub
3. Manage images with the CLI
4. Create Dockerfiles and use them to build images
5. Clarify the difference between images and containers

Docker

Docker Images

Docker Images

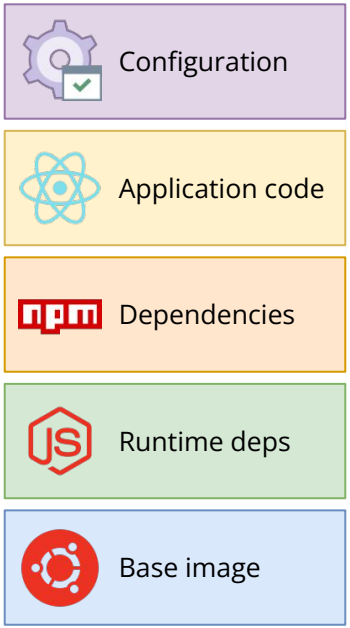
Understanding the DNA of our containers

Images are a self-contained, read-only template that encapsulates everything needed to run your application:

- **The base layer:** often a minimal Linux distribution like Alpine, or a more full-fledged one like Ubuntu.
- **Runtime Environment:** Specific software (e.g., Python, Node.js) required by your application.
- **Libraries & Dependencies:** All the external code your application relies on.
- **Application Code:** Your own source code or compiled binaries.
- **Configuration:** Settings for your application and its environment.

A Docker image is like a snapshot of your application and its complete runtime environment, frozen in time and ready to be brought to life as a container. Images can be sourced from multiple locations:

- **Docker Hub:** This is the official Docker image repository, offering a vast collection of images for various purposes.
- **Private Registries:** For organizations or individuals with proprietary software, private registries provide secure storage for custom images.
- **Building Your Own Images:** This is where the real power of Docker shines. By writing Dockerfiles, you can create images that perfectly match your application's requirements.



Docker

Container Registries

Container Registries

Storing and managing Docker images

- Container registries offer a multitude of benefits:
  - **Collaboration:** Share your images with teammates, clients, or the wider community.
  - **Versioning:** Track different versions of your images for easy rollback and updates.
  - **Security:** Private registries provide a secure environment for storing sensitive images.
  - **Automation:** Automate image building and deployment as part of your CI/CD pipeline.
- Types of Container Registries:
  - **Public Registries:** Open to everyone and host a vast collection of images from various sources. Docker Hub is the most prominent example.
  - **Private Registries:** Used for storing proprietary or sensitive images and offer granular access control.

Container Registries

What to consider when selecting Container Registries

Hosting Type	**Public**	Ideal for open-source projects, community collaboration, and public distribution.
	Private	Essential for sensitive data, proprietary code, and internal projects.
	Self-Hosted	Offers maximum control and flexibility, but requires infrastructure management.
	Cloud-Hosted	Convenient, scalable, and often integrates well with other cloud services.
Security Features	**Basic (Authentication)**	Sufficient for less sensitive projects or public images.
	Advanced (RBAC, Scanning, Signing)	Crucial for sensitive data, compliance requirements, and secure deployments.
Integrations	**Limited**	Suitable for standalone projects or simple workflows.
	Extensive (API, Webhooks, CI/CD)	Enables automation, complex workflows, and integration with other tools.
Cost Model	**Free Tier (with limitations)**	Great for experimentation, small projects, or public images.
	Usage-Based	Flexible, but costs can scale with storage and bandwidth needs.
	Fixed Subscription	Predictable costs, but may not be cost-effective for low usage.
	Open-Source	Free to use, but requires infrastructure and maintenance investment.

Docker

Docker

Images Deep Dive

Images Deep Dive

Section overview

1. Explore Dockerfile and Docker's Layered Architecture
2. Learn about build contexts and `.dockerignore`
3. Understand how to pass environment variables to containers
4. Differentiate between `CMD` and `ENTRYPOINT`
5. Explore Distroless images and understand their benefit

Images Deep Dive

Section overview

6 Master multistage builds

7 Explore strategies to optimize image size and build time

Docker

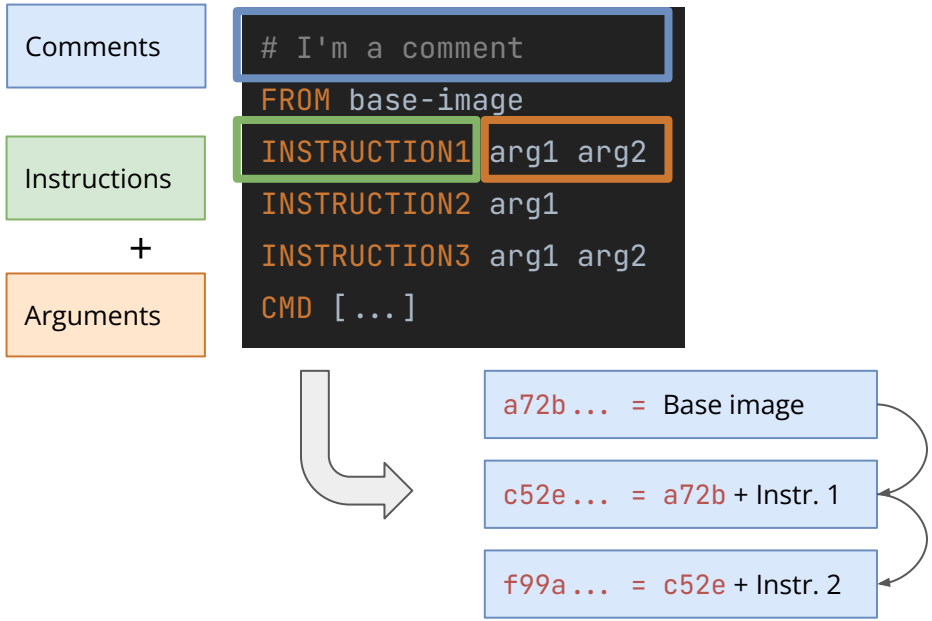
Dockerfile

Dockerfile

Programmatically define steps for the creation of Docker images

Dockerfiles give you the power to create custom images that perfectly match your requirements. They bring many advantages, among which the main ones are:

- **Reproducibility:** By defining all the steps necessary to build your Docker images, you ensure that anyone with the Dockerfile can recreate the exact same image.
- **Automation:** Dockerfiles automate the build process for your applications. This removes manual steps and reduce the risk of human error.
- **Transparency and documentation:** Dockerfiles should act as good documentation for how your image is built by providing a clear overview of all the steps involved in building it.
- **Optimization:** Since Dockerfiles provide you full control over the build steps, you can tackle into optimizations to improve security and speed up build time and image size.



Docker

Distroless Images

Distroless Images

Smaller, more secure alternatives to run your applications

- Distroless images are minimal Docker images that contain only the necessary runtime dependencies of applications. Unlike traditional images that include an operating system, shell utilities, and other binaries, distroless images exclude these components, resulting in a much smaller and more secure image.
  - Distroless images exclude OS distributions and related components, such as shell utilities and additional binaries.
  - As a result, Distroless images are smaller and more secure, but also harder to work with!

Advantages	Challenges
✅ **Enhanced Security:** Fewer components in the image mean fewer potential vulnerabilities and a smaller attack surface.	⭕ **No Debugging Tools:** Without shell utilities or debugging tools, troubleshooting issues inside a distroless container is challenging.
✅ **Reduced Image Size:** Smaller image size means faster image pulls and reduced storage requirements.	⭕ **Dependency Management:** Managing dependencies is more complex, as you need to ensure all required libraries and binaries are included in the build process.
✅ **Improved Performance:** The smaller size and fewer components can result in quicker container startup times and lower resource consumption.	⭕ **Increased Build Complexity:** Creating distroless images often involves more complex Dockerfiles and build processes.
✅ **Simplified Compliance and Auditability:** With fewer components, distroless images are easier to audit and verify.	⭕ **Learning Curve:** The knowledge from traditional images is not immediately applicable due to the lack of familiar tools and a different debugging workflow.

Docker

Volumes

Volumes

Section overview

1. Understand how data persistence works in Docker
2. Learn how volumes work
3. Differentiate between bind mounts and volumes
4. Manage volumes effectively with the CLI

Docker

Volumes

Volumes

Persisting data beyond the container's lifecycle

Docker volumes enable you to persist data outside the container lifecycle. They act as managed directories or files, separate from the container’s file system, ensuring your data remains safe and accessible.

Benefits of using volumes:

- **Data Persistence:** Persist your data even if the container stops or is removed.
- **Data Sharing:** Share data between multiple containers by sharing the same volume.
- **Backup and Recovery:** Easily back up and restore your valuable data.
- **Decoupling Data from Containers:** Gain flexibility in managing and deploying containers by separating data from application runtime.

Types of Docker volumes

Type	Description	Use-case
Bind Mounts	Directly link host system directories or files to your container.	Ideal for real-time development updates.
Named Volumes	Created by the user and reusable across containers.	Best match for perfect for persistent data.
Anonymous Volumes	Automatically created, but without a name that enables reusability.	Not often used. Temporary data that doesn't need to persist.

Docker

Advanced Topics

Advanced Topics

Section overview

1. Deep dive into setting resource limits for containers
    1. Setting CPU limits for pods
    2. Setting memory limits for pods
    3. Discuss what happens when limits are exceeded
2. Understand how different restart policies work
3. Explore Docker networking
    1. Work with default Docker networks
    2. Leverage user-defined networks for improved functionality

Docker

Networking

Networking

Leverage Docker's networking capabilities to isolate and connect containers

Docker networking provides the infrastructure to make inter-container and external communication possible while maintaining the security and isolation benefits of containers.

- Containers in a Docker network receive a unique IP address. This address allows other containers on the same network to communicate with it.
- We can also assign **names** to containers and use them as aliases. This is best practice, since IP addresses may change on container recreation, and names are more stable.
- Containers in bridge networks can be made reachable from the host network by exposing ports.
- Containers can be connected to multiple networks, and will receive a unique IP address per network.

Main Network Drivers

Driver	Description	Use-case
Bridge (Default)	Creates a private network on the host machine where your containers can communicate with each other.	Suitable for most single-host scenarios.
Host	Removes network isolation between the container and the host.	Useful for maximum performance when strict isolation is not required.
None	Disables all networking for a container.	Rarely used, but can be helpful for specific isolation requirements.
Overlay	Designed for multi-host networking, allowing containers on different hosts to communicate directly.	Essential for creating swarm clusters or distributed applications.

*** Macvlan:** assigns a MAC address to the containers, making them appear as a physical device.

Docker

Docker Compose

Docker Compose

Section overview

1. Explore the benefits of Docker Compose
2. Learn how to use Compose to manage Docker resources
    1. Manage application lifecycle with Compose
    2. Handle environment variables
    3. Create and manage volumes
    4. Create and manage networks
    5. Define and manage service dependencies
3. Import and merge multiple Compose configurations

Docker

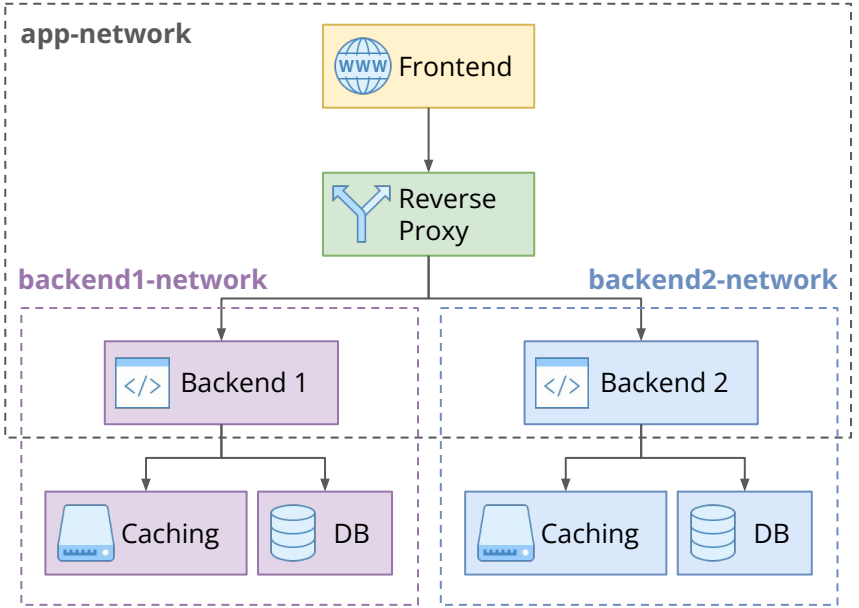
Docker Compose

Docker Compose

More robust multi-container management for development workflows

Managing multi-container applications only using Docker can be very challenging:

- Manually starting and linking multiple containers is error-prone.
- Ensuring containers start in the correct order can be challenging.
- It's difficult to ensure consistency across environments, leading to bugs and integration issues.
- Managing networks for inter-container communication and volumes for persistent storage across container restarts is no simple task.
- Managing configuration via environment variables can be complex and require checking multiple places to get a consolidated view of the configuration.



Docker Compose

More robust multi-container management for development workflows

When to use Docker Compose?

- **Local Development:** Create consistent development environments easily.
- **Testing and Staging Environments:** Replicate production environments for accurate testing.
- **CI/CD Pipelines:** Automate environment setup during build and deployment.

```
docker-compose up -d
./tests
docker-compose down
```

- **Single-host production deployments:** Ideal for small-scale production setups.

```yaml
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
    networks:
      - my_network
    depends_on:
      - db

  db:
    image: mongo:latest
    volumes:
      - db-data:/data/db
    networks:
      - my_network

volumes:
  db-data:

networks:
  my_network:
```

Docker

Project - Customizing NGINX

Project - Customizing NGINX

Project overview

Welcome to this foundational project on Docker and the basics of interacting with containers! Let's see what we will achieve with this project:

- Run a container based on the NGINX image, using the specific tag 1.27.0.
- Get an interactive shell running inside of the container.
- Install the text editor vim.
- Modify the index.html from the NGINX server to deliver custom content.

This is a fairly simple project, but it will help us get familiar with a few key aspects of working with Docker. And if you are looking for more challenging ones, don't worry! We will also tackle more complex topics in upcoming projects :)

Project - Customizing NGINX

Project summary, limitations, and next steps

Executing commands inside our container is a great first step towards understanding Docker, but there is still so much more to learn! Let's have a look at some limitations of our project:

- We ran many commands manually, and this is definitely not a scalable approach. If our container dies, and we need to recreate it, then we need to run all these commands again. There is a lot of room for error!
  - The same reasoning goes for creating copies of our container: we would have to run the commands manually on every new container created based on the NGINX image.
- We edited files within the container, but if that container disappears, we lose all the changes we made. Containers' storage is, by definition, ephemeral, so we have to have a better way of handling persistent storage.
- The commands we run are not written anywhere! What if we forget one of them, or what if we run a different one instead?

Docker

Project - Containerize an Express App

Project - Containerize an Express App

Project overview

Welcome to this project on Docker and the basics of building Docker images with Dockerfiles! Let's see what we will achieve with this project:

- Code a backend API using NodeJS and Express
- Write a Dockerfile to build an image for our application
- Create containers based on the built image
- Get a solid understanding of the lifecycle for coding, building, and running applications with Docker

Docker

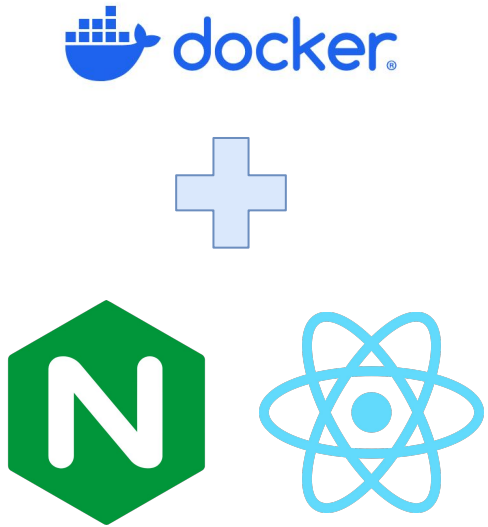
Project - Containerize a React App

Project - Containerize a React App

Project overview

Welcome to this project on Docker and working with multistage builds! Let's see what we will achieve with this project:

- Spin up a React App using create-react-app
- Write a multistage Dockerfile to build the React application
- Extend the Dockerfile to serve static files through an Nginx server
- Get familiar with different use-cases for using multistage builds, in addition to what was discussed during the previous lectures.



Docker

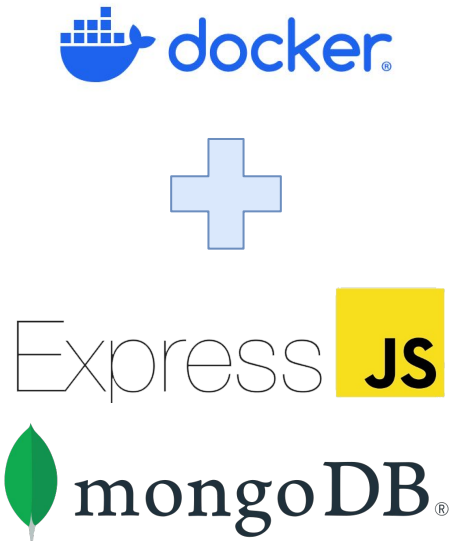
Project - Build a key-value REST API

Project - Build a key-value REST API

Project overview

Welcome to this project on Docker and working with multiple containers! Let's see what we will achieve with this project:

- Code a key-value API to store information under specific keys
- Run multiple containers for the application and database
- Leverage volumes to persist data from the database container
- Connect application and database containers by leveraging user-defined networks
- Use shell scripts to manage the created containers, networks, and volumes



Project - Build a key-value REST API

Project overview

API specification	
API method and path	**Expected behavior**
`POST /store`	■ Expects request body to contain both `key` and `value` ■ Returns `400` if this is not the case ■ If the key already exists in DB, returns `400` ■ If the key does not exist, stores key and value, and returns `201`
`GET /store/:key`	■ If the key does not exist, return `404` ■ If the key exists, returns `200` and the key and value
`PUT /store/:key`	■ Expects request body to contain `value` ■ Returns `400` if this is not the case ■ If the key does not exist, returns `404` ■ If the key exists, updates the value and returns `200`
`DELETE /store/:key`	■ If the key does not exist, return `404` ■ If the key exists, deletes the key and returns `204`
`GET /health`	■ Returns `200` and the text `up`

Docker

Docker

Project - Build a notes REST API

Project - Build a notes REST API

Project overview

Welcome to this project on Docker and working with multiple containers and services! Let's see what we will achieve with this project:

- Code a notes REST API with two services, notes and notebooks, and a reverse proxy
- Create and manage multiple Compose projects
- Leverage Compose merge functionality to integrate individual projects and manage them simultaneously
- Explore how to manage service outage and improve application resiliency

Project - Build a notes REST API

Project overview

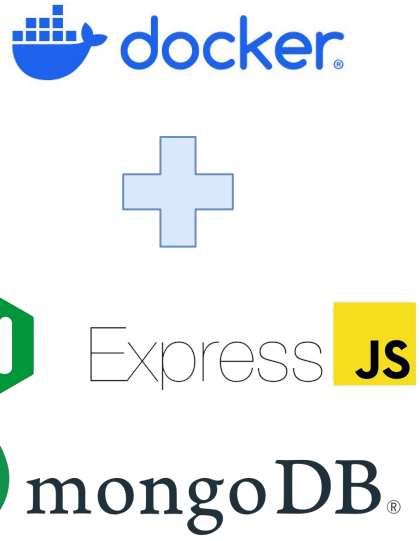
Notebooks REST API specification	
API method and path	**Expected behavior**
`POST /api/notebooks`	■ Expects request body to contain `name`, returns `400` otherwise ■ Optionally receives `description` information in body ■ Saves notebook information and return `201`
`GET /api/notebooks`	■ Returns all notebooks
`GET /api/notebooks/:id`	■ If a notebook with provided `id` does not exist, returns `404` ■ Returns notebook information
`PUT /api/notebooks/:id`	■ If a notebook with provided `id` does not exist, returns `404` ■ If the `id` exists, updates the notebook information and returns `200`
`DELETE /api/notebooks/:id`	■ If the `id` does not exist, returns `404` ■ If the `id` exists, deletes the notebook and returns `204`
`GET /health`	■ Returns `200` and the text `up`

Docker

Project - Build a notes REST API

Project overview

Notes REST API specification	
API method and path	**Expected behavior**
`POST /api/notes`	■ Expects request body to contain `title` and `content`, returns `400` otherwise ■ Optionally receives `notebookId` information in body ■ If `notebookId` does not exist, returns `404`. ■ If `notebooks` service is down, stores note with provided `notebookId` ■ Saves notes information and return `201`
`GET /api/notes`	■ Returns all notes
`GET /api/notes/:id`	■ If a note with provided `id` does not exist, returns `404` ■ Returns note information
`PUT /api/notes/:id`	■ If a note with provided `id` does not exist, returns `404` ■ If the `id` exists, updates the note information and return `200`
`DELETE /api/notes/:id`	■ If the `id` does not exist, returns `404` ■ If the `id` exists, deletes the note and return `204`
`GET /health`	■ Returns `200` and the text `up`



Docker

Kubernetes

10000-Foot Overview

10000-Foot Overview

Section overview

1. What is Kubernetes? And why use it?
2. Discuss Kubernetes' architecture
    1. Control plane vs. data plane
    2. Nodes and Kubernetes objects
3. Introduce the Kubernetes CLI: `kubectl`
    1. Differentiate `kubectl` and the Kubernetes cluster
    2. Understand how `kubectl` communicates with the cluster
    3. Discuss the structure of `kubectl` commands

Kubernetes

Managing containers at scale

Managing containers at scale

Can't we get by with just Docker?

- Docker and Docker Compose alone are enough for running individual containers or creating development environments, but they do not meet the requirements of production workloads.

Challenge	Docker	Kubernetes
Container Scheduling	⭕ Manual assignment of containers to servers ⭕ Time consuming and error-prone	✅ Declarative definition of criteria for node (server) assignment ✅ Handled automatically by Kubernetes based on resource definitions
Load Balancing	⭕ No built-in load balancing	✅ Services provide built-in load balancing across Pods
Scaling Applications	⭕ Lacks automation for horizontal scaling ⭕ Requires low-level adjustments for directing load to newly created containers	✅ Horizontal Pod Autoscaler can scale pods horizontally and automatically through real-time metrics
Self-Healing	⭕ Lacks automation for detecting and restarting unhealthy workloads	✅ Health checks and continuous monitoring allow for considerably more robust self-healing mechanisms

Managing containers at scale

Can't we get by with just Docker?

- Docker and Docker Compose alone are enough for running individual containers or creating development environments, but they do not meet the requirements of production workloads.

Challenge	Docker	Kubernetes
Service Discovery	⭕ No built-in solution for service discovery at scale	✅ Internal DNS allows for robust service discovery mechanisms through Services
Configuration Management	⭕ Requires low-level configuration and adjustments, and can become cumbersome for larger applications and multiple environments	✅ Configuration maps and secrets allow decoupling and scaling of application configuration

Kubernetes

What is Kubernetes?

What is Kubernetes?

Understanding the goal, features, and architecture of Kubernetes

- **Open-source** tool, which has become the **de facto standard for container orchestration**.
  - Kubernetes is designed to be easily extensible, and as a result the K8s ecosystem has grown to thousands of different tools.
- **Goal:** address the complexities of **running containers in production**. As such, the tool is designed around:

Automation	Declarative	Scalability	Resilience
Reduces the need for manual intervention in deploying and managing applications.	Declare the final, desired state and let Kubernetes figure out how to get there.	Effortlessly scales applications horizontally to meet demand.	Ensures that applications are always available, even in the face of failures.

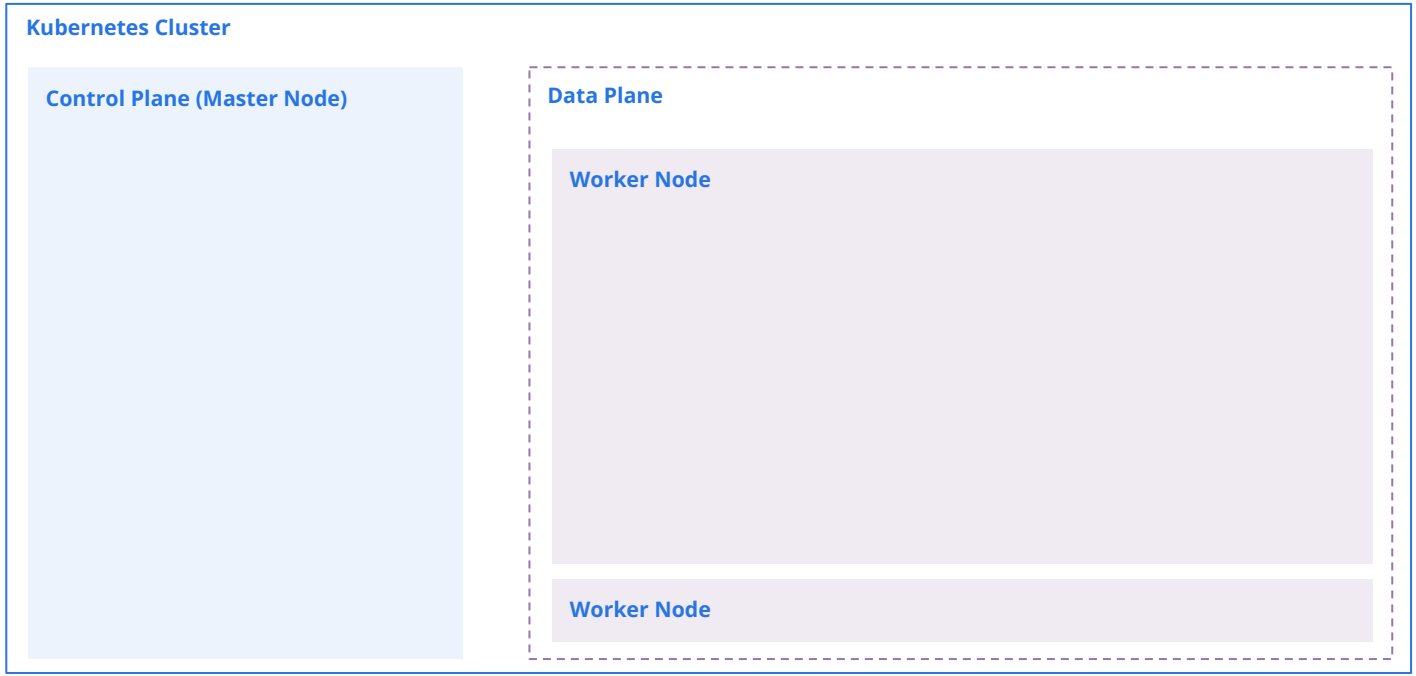
- A Kubernetes cluster has many components, and they are either placed in the control plane or in worker nodes:
  - **Control plane:** it's the brain of the cluster, containing the components responsible for managing the entire system.
  - **Worker nodes:** they are the machines where your applications actually run.

Kubernetes

Kubernetes Architecture

Kubernetes Architecture

Understanding the main components of a Kubernetes Cluster

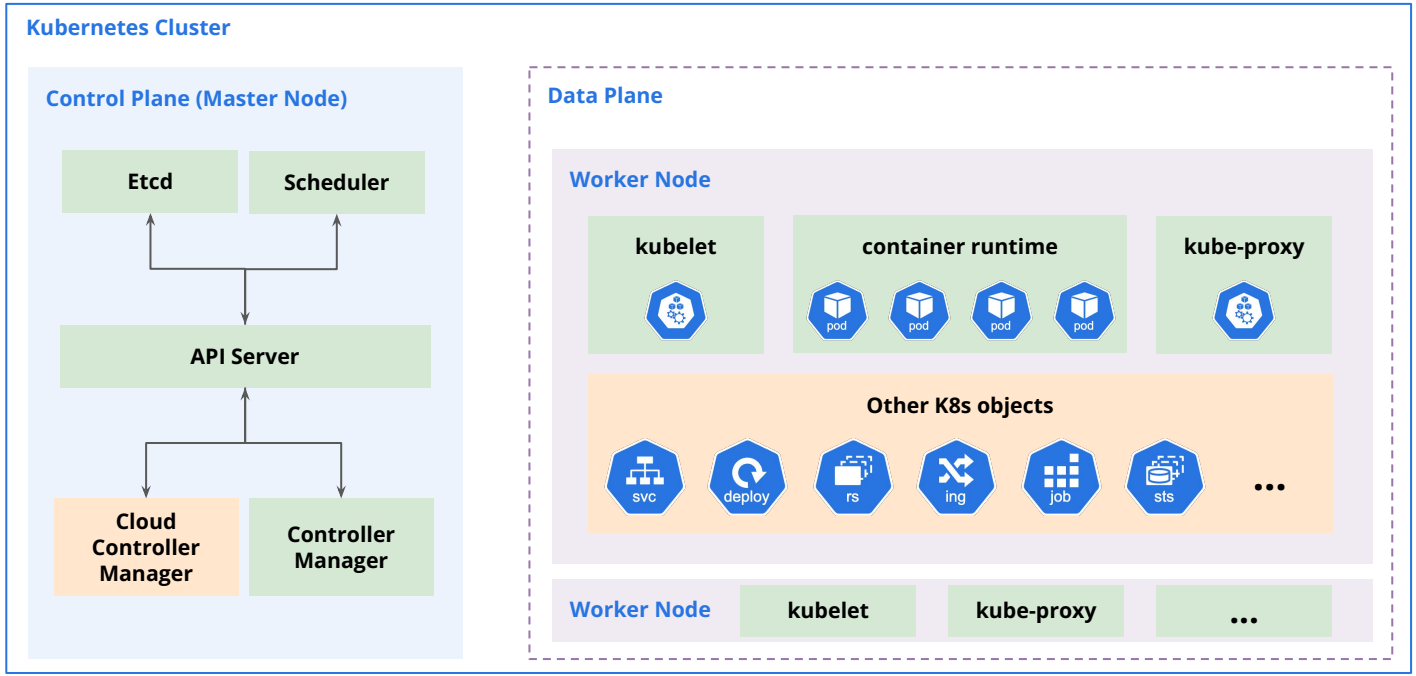


Kubernetes Architecture

Understanding the main components of a Kubernetes Cluster

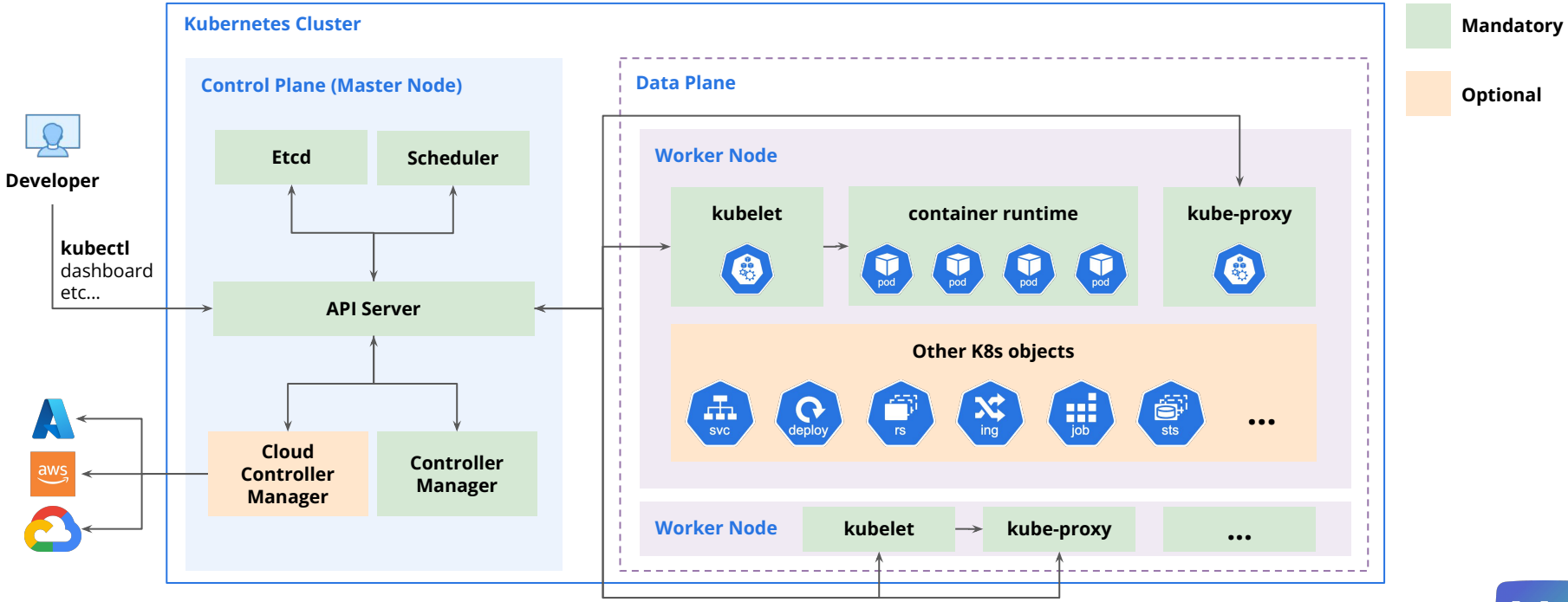
Kubernetes Architecture

Understanding the main components of a Kubernetes Cluster



Kubernetes Architecture

Understanding the main components of a Kubernetes Cluster



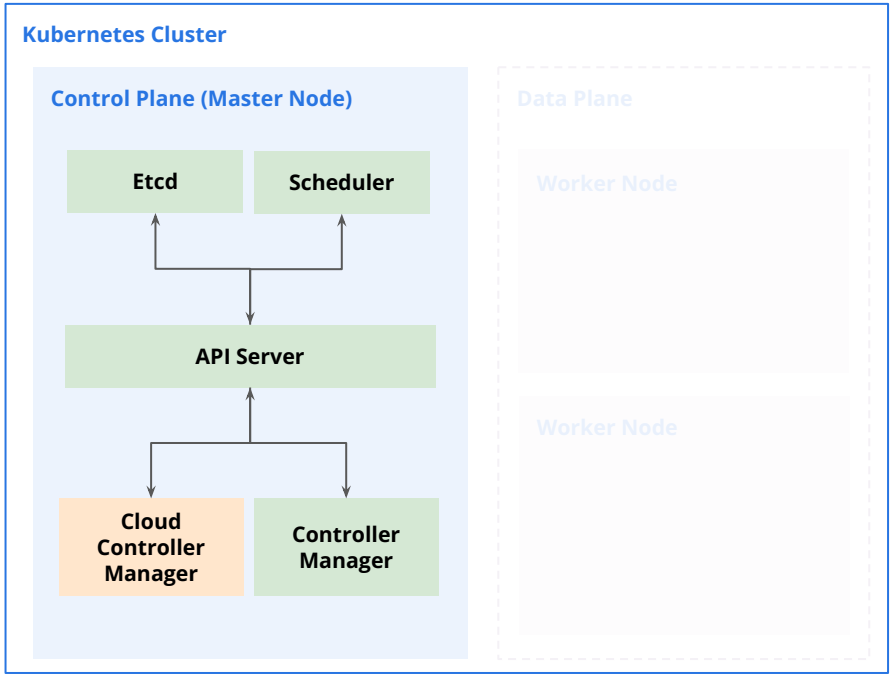
Kubernetes

The Control Plane

The Control Plane

Taking a closer look at the brains of Kubernetes

- **API Server:** exposes the Kubernetes API, which is the main entry point for all administrative tasks (kubectl, dashboard, etc.).
- **Scheduler:** responsible for placing pods on the most suitable nodes. It selects nodes based on resource availability and other constraints.
- **Controller Manager:** responsible for running controller processes that handle routine tasks (e.g. ReplicaSet controller, node controller, job controller, etc.).
- **Etcd:** distributed key-value store that stores all cluster data, including the configuration and state of the cluster. Acts as the single source of truth for the cluster's state.
- **Cloud Controller Manager:** enables Kubernetes to interact with the underlying cloud infrastructure. It handles tasks such as managing cloud-based load balancers, persistent storage, and node management.



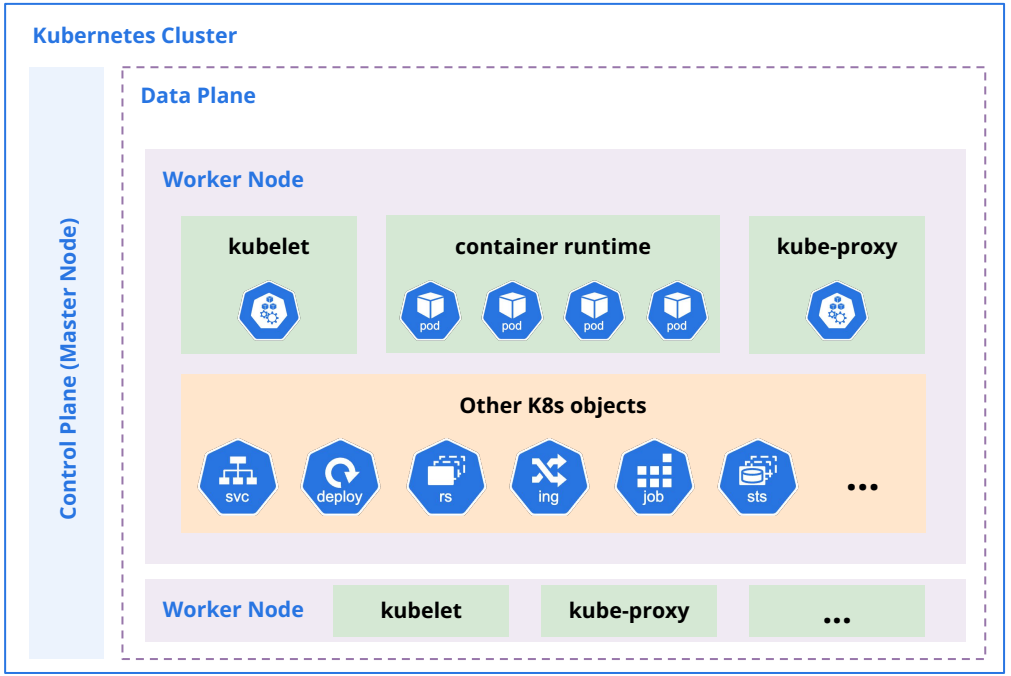
Kubernetes

The Worker Nodes

The Worker Nodes

Taking a closer look at where containers are actually run

- **kubelet:** agent that runs on each worker node and communicates with the control plane. It ensures that containers are running in the pods as defined by the pod specifications.
- **Container Runtime:** responsible for running the containers on each worker node. Kubernetes supports several runtimes, such as Docker, containerd, or CRI-O.
- **kube-proxy:** manages network rules on each worker node. It maintains network connectivity and load balancing for services, ensuring that requests are routed to the appropriate pods.
- **Other K8s Objects:** higher-level Kubernetes objects such as ReplicaSets, Deployments, Jobs, Services, StatefulSets, among others.



Kubernetes

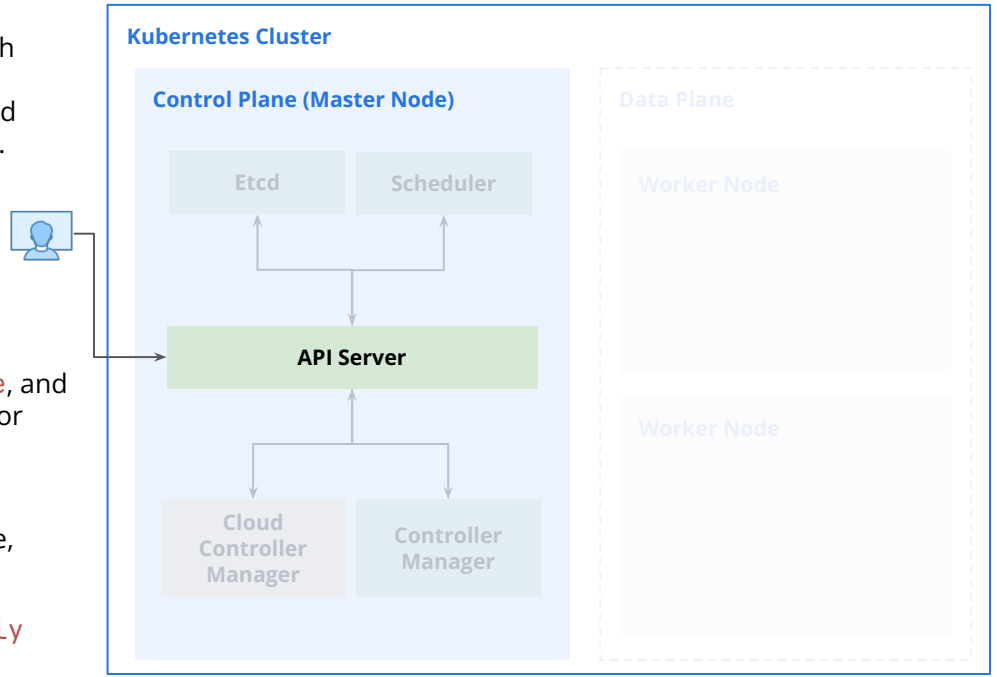
The kubectl CLI

The kubectl CLI

Interacting with Kubernetes clusters via the CLI

- `kubectl` is the primary command-line utility for interacting with a Kubernetes cluster. It allows you to manage and control Kubernetes resources, such as pods, services, deployments, and more, by sending API requests to the Kubernetes control plane.

```
kubectl [command] [res. type] [name] [flags]
```



Main types of commands / use-cases

- **Inspect cluster resources:** Commands such as `get`, `describe`, and `logs` can be used to describe the state of individual resources or groups of resources.
- **Imperative resource management:** Commands such as `run`, `scale`, `create`, and `delete` can be used to imperatively create, update, and delete resources.
- **Declarative resource management:** Commands such as `apply` and `diff` can be used to declaratively manage resources.

Kubernetes

Installing K8s Tooling

Installing K8s Tooling

Section overview

1. Discuss Pods and their role in Kubernetes
2. Manage Pods in Kubernetes
    1. Create Pods to run containers
    2. Retrieve Pod information and delete Pods
3. Expose Pods via Services
    1. Create services to facilitate communication with Pods
    2. Understand why services are helpful
4. Understand how to go from Dockerfile to Pods

Kubernetes

Running Containers in K8s

Running Containers in K8s

Section overview

1. Discuss Pods and their role in Kubernetes
2. Manage Pods in Kubernetes
    1. Create Pods to run containers
    2. Retrieve Pod information and delete Pods
3. Expose Pods via Services
    1. Create services to facilitate communication with Pods
    2. Understand why services are helpful
4. Understand how to go from Dockerfile to Pods

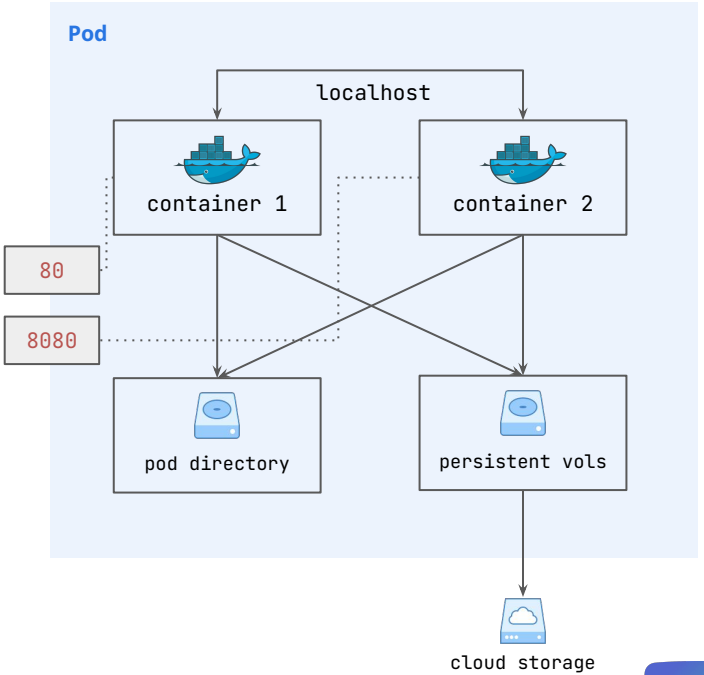
Kubernetes

Pods

Pods

Meet the smallest and simplest unit that you create to run containers.

- Pods represent a single instance of a running process in your cluster. They:
  - Encapsulate one or more containers.
  - Allow containers to share storage and network resources.
  - Provides multiple options to configure how containers are run (ports, environment variables, volumes, security configuration, among others).
- Pods provide a higher level of abstraction than working directly with containers, allowing multiple containers to work together if necessary.
  - Containers running in the same Pod can communicate with each other via `localhost`.
  - They can also read/write to the same volumes.
- Pods can all communicate with each other by default in Kubernetes.
- We can set health probes in each container so that they are restarted or stop receiving traffic if considered unhealthy.

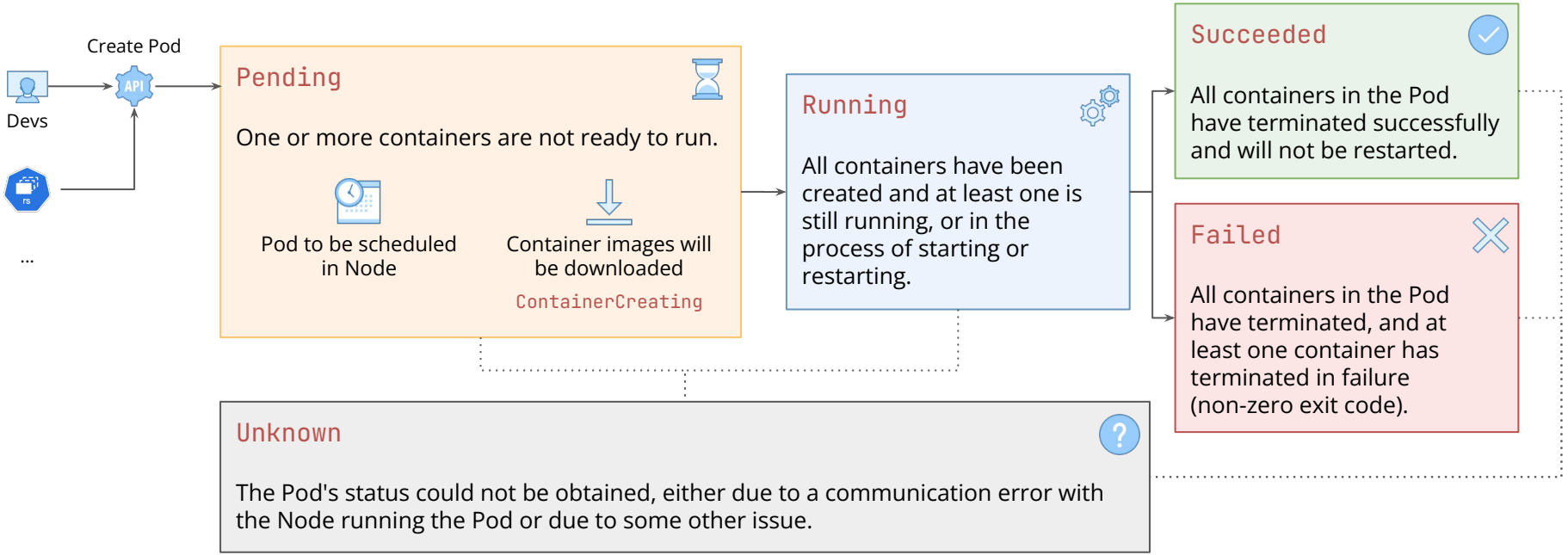


Kubernetes

Pod lifecycle

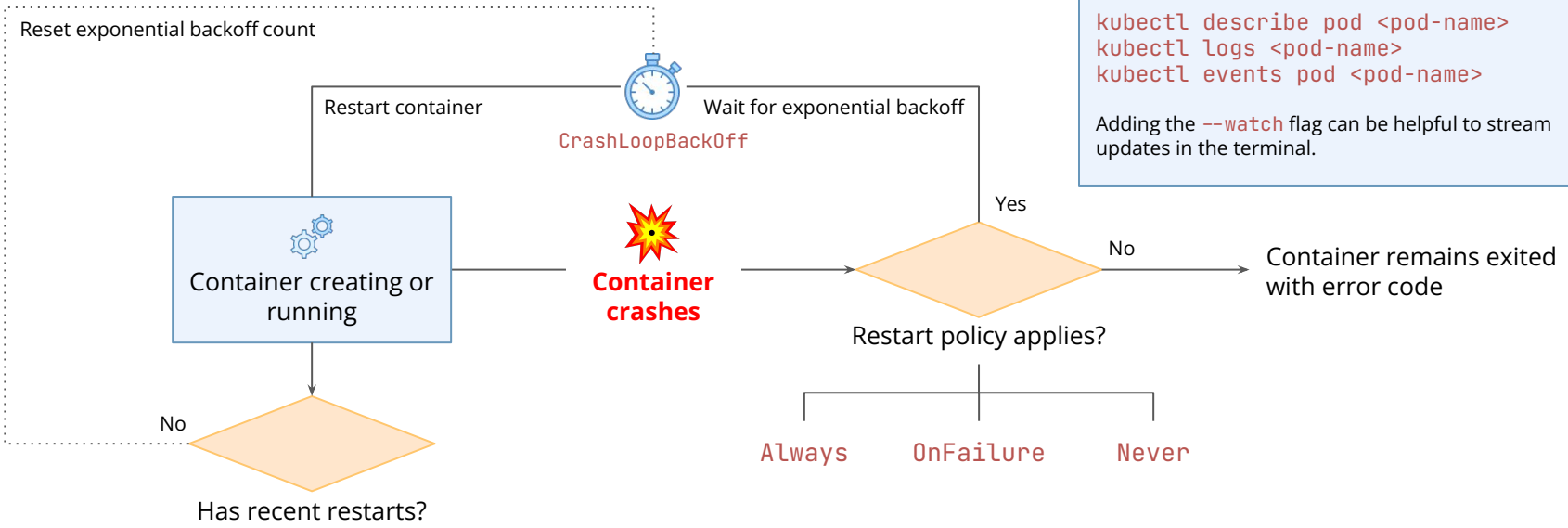
Pod lifecycle

Understand the different phases of a Pod's lifecycle



Pod lifecycle

Understand how Pods handle container errors



Helpful debugging commands

```
kubectl describe pod <pod-name>
kubectl logs <pod-name>
kubectl events pod <pod-name>
```

Adding the `--watch` flag can be helpful to stream updates in the terminal.

Kubernetes

Object Management & YAML Files

Object Management & YAML Files

Section overview

1. Discuss the different approaches to managing objects in Kubernetes
2. Explore Kubernetes YAML Manifests / Configuration Files
3. Deep dive into imperative and declarative object management
    1. Understand the shortcomings of imperative management
    2. Discuss the challenges of declarative management
    3. Explore different use-cases for each approach
    4. Go through how to migrate from imperative to declarative management
4. Explore declaring multiple objects in a single manifest

Kubernetes

Object Management

Object Management

Understanding the multiple approaches available to manage k8s objects

	Imperative mgmt. with kubectl	Imperative mgmt. with config files	Declarative mgmt. with config files

Main commands	`kubectl create <resource> [config]`
	`kubectl delete <resource>`
	`kubectl expose <resource> <name>`
	`...`
	`kubectl [get | describe | logs ] ...`
		`kubectl create -f <filename>`
		`kubectl delete -f <filename>`
		`kubectl replace -f <filename>`
		
		`kubectl [get | describe | logs ] ...`
			`kubectl apply -f <filename>`
			`kubectl diff -f <filename>`
			`kubectl delete -f <filename>`
			
			`kubectl [get | describe | logs ] ...`

Pros	✅ Lowest learning curve ✅ Commands transparently communicate changes via single word actions ✅ Single step to make changes to the cluster	✅ Configuration files can be committed, reviewed, and audited ✅ Files provide a template for creating new objects ✅ Simpler than declarative management	✅ Persists updates made to live objects even if not reflected in the configuration files ✅ Better support for automatically identifying necessary operations for each object
Cons	⭕ Not possible to save templates for creating new objects ⭕ No change review nor audit trail possible ⭕ No records of what has been created / deleted (only what is in the cluster)	⭕ More suitable for single files, rather than directories ⭕ Requires familiarity with the object schemas for each object being managed ⭕ Does not persist updates made outside the configuration files	⭕ Highest learning curve ⭕ Partial updates are more complex to understand and debug ⭕ Live objects' state might not be entirely reflected in the configuration files

Kubernetes

K8s Manifest Files

K8s Manifest Files

Understanding the structure of K8s YAML manifests

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: my-nginx-pod
 labels:
   app: nginx
spec:
 containers:
   - name: nginx
     image: nginx
     ports:
       - containerPort: 80
```

K8s Manifest Files

Understanding the structure of K8s YAML manifests

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: my-nginx-pod
 labels:
   app: nginx
spec:
 containers:
   - name: nginx
     image: nginx
     ports:
       - containerPort: 80
```

apiVersion

The `apiVersion` field defines which API group and the respective version of the API is being used to create the object.

kind

The `kind` field defines which exact object or resource is being managed by the configuration file. It must be supported by the specified `apiVersion`.

metadata

The `metadata` field defines data that is used to uniquely identify the object, such as name, namespace, labels, and annotations. K8s may also add information to the `metadata` section.

spec

The `spec` field defines the actual configuration for the objects being managed by the configuration file. The exact shape of the configuration varies according to both the `apiVersion` and the `kind` fields.

K8s Manifest Files

Understanding the structure of K8s YAML manifests

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: my-nginx-pod
 labels:
   app: nginx
spec:
 containers:
   - name: nginx
     image: nginx
     ports:
       - containerPort: 80
```

`apiVersion`

The `apiVersion` field defines which API group and the respective version of the API is being used to create the object.

`kind`

The `kind` field defines which exact object or resource is being managed by the configuration file. It must be supported by the specified `apiVersion`.

`metadata`

The `metadata` field defines data that is used to uniquely identify the object, such as name, namespace, labels, and annotations. K8s may also add information to the `metadata` section.

`spec`

The `spec` field defines the actual configuration for the objects being managed by the configuration file. The exact shape of the configuration varies according to both the `apiVersion` and the `kind` fields.

K8s Manifest Files

Understanding the structure of K8s YAML manifests

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: my-nginx-pod
 labels:
   app: nginx
spec:
 containers:
   - name: nginx
     image: nginx
     ports:
       - containerPort: 80
```

`apiVersion`

The `apiVersion` field defines which API group and the respective version of the API is being used to create the object.

`kind`

The `kind` field defines which exact object or resource is being managed by the configuration file. It must be supported by the specified `apiVersion`.

`metadata`

The `metadata` field defines data that is used to uniquely identify the object, such as name, namespace, labels, and annotations. K8s may also add information to the `metadata` section.

`spec`

The `spec` field defines the actual configuration for the objects being managed by the configuration file. The exact shape of the configuration varies according to both the `apiVersion` and the `kind` fields.

K8s Manifest Files

Understanding the structure of K8s YAML manifests

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: my-nginx-pod
 labels:
   app: nginx
spec:
 containers:
   - name: nginx
     image: nginx
     ports:
       - containerPort: 80
```

`apiVersion`

The `apiVersion` field defines which API group and the respective version of the API is being used to create the object.

`kind`

The `kind` field defines which exact object or resource is being managed by the configuration file. It must be supported by the specified `apiVersion`.

`metadata`

The `metadata` field defines data that is used to uniquely identify the object, such as name, namespace, labels, and annotations. K8s may also add information to the `metadata` section.

`spec`

The `spec` field defines the actual configuration for the objects being managed by the configuration file. The exact shape of the configuration varies according to both the `apiVersion` and the `kind` fields.

Kubernetes

ReplicaSets and Deployments

ReplicaSets and Deployments

Section overview

1 Explore ReplicaSets

1. Understand how they relate to Pods
2. Discuss advantages and limitations of ReplicaSets

2 Explore Deployments

1. Understand how deployments address the shortcomings of ReplicaSets
2. Deep dive into how to create and manage Deployments
3. Explore rollouts and discuss how to debug failed rollouts
4. Explore scaling deployments

Kubernetes

ReplicaSets

ReplicaSets

Ensure that a certain number of pods is running at any given time

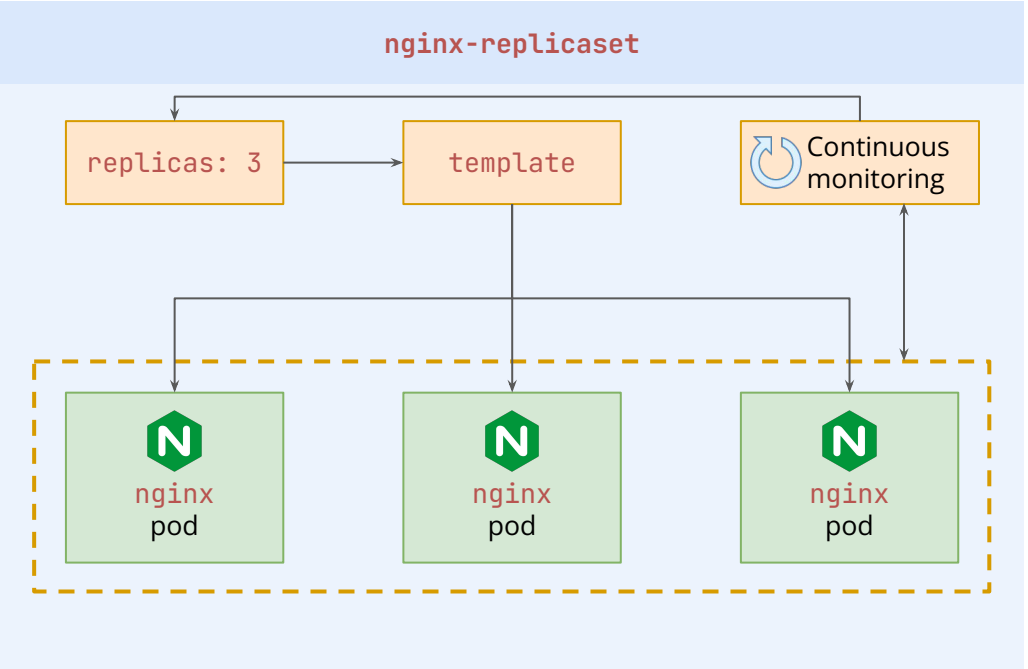
- So far, we ran pods manually and individually. If a pod breaks or enters into an unhealthy state, we need to manually intervene.
- ReplicaSets address the issue of replacing exited or unhealthy pods and making sure that a stable number of identical pods are running at any point in time.
  - Ensures high availability and fault tolerance by automatically replacing failed or terminated pods.
- ReplicaSets work by identifying pods via `selectors`, and continuously checking whether the number of running pods matches the desired number of pods specified in the configuration.
- ReplicaSets will recreate pods automatically based on the `template` section in their `spec`.

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.0
```


ReplicaSets

Ensure that a certain number of pods is running at any given time

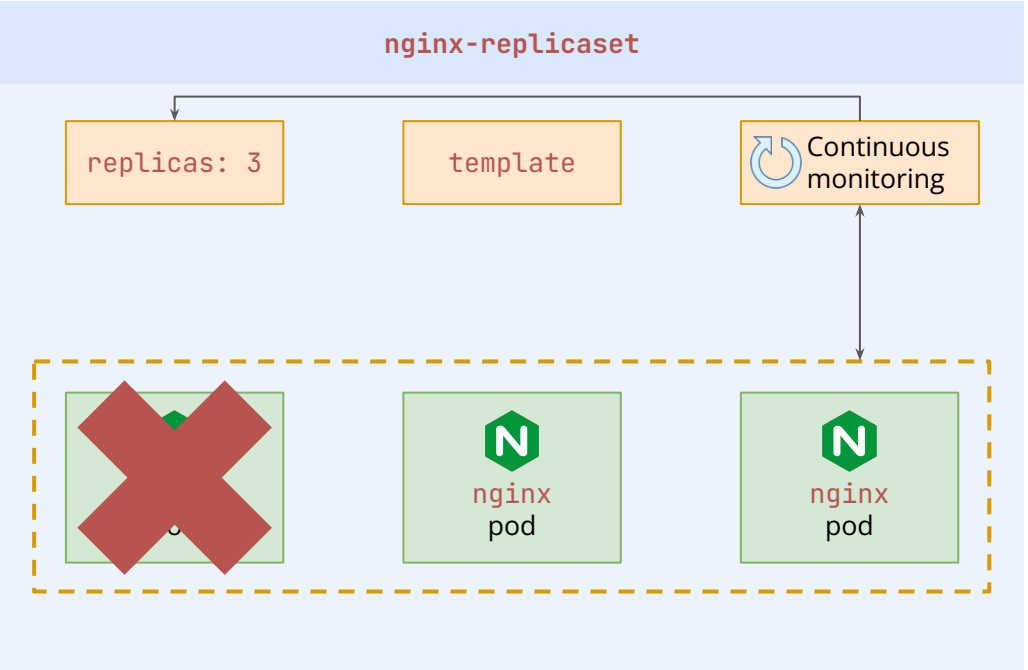
```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.0
```



ReplicaSets

Ensure that a certain number of pods is running at any given time

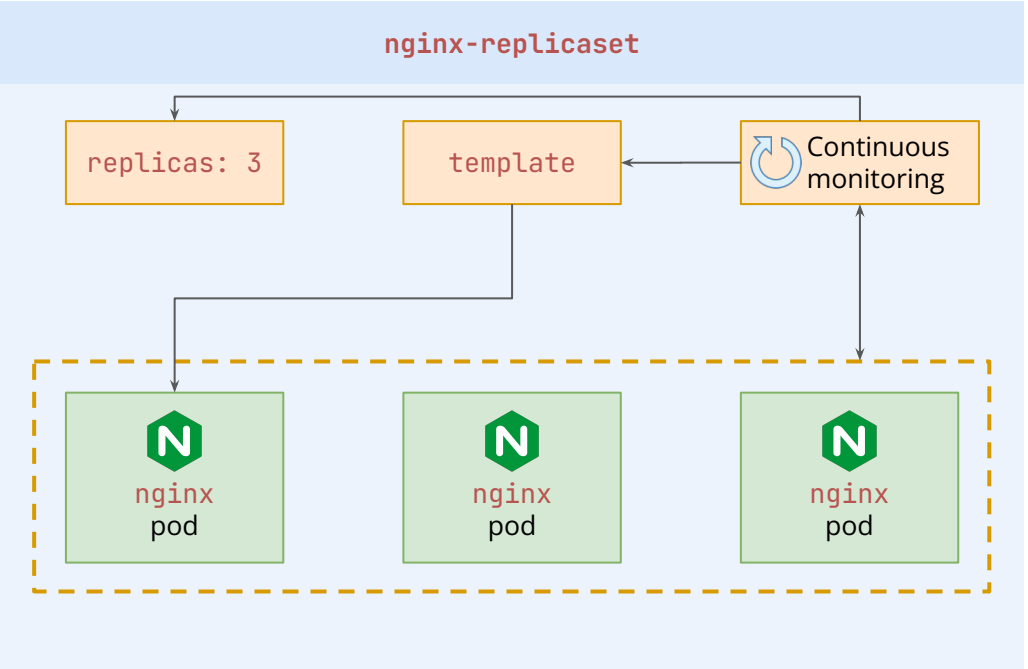
```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.0
```



ReplicaSets

Ensure that a certain number of pods is running at any given time

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.0
```



Kubernetes

Deployments

Deployments

Manage and update applications at scale

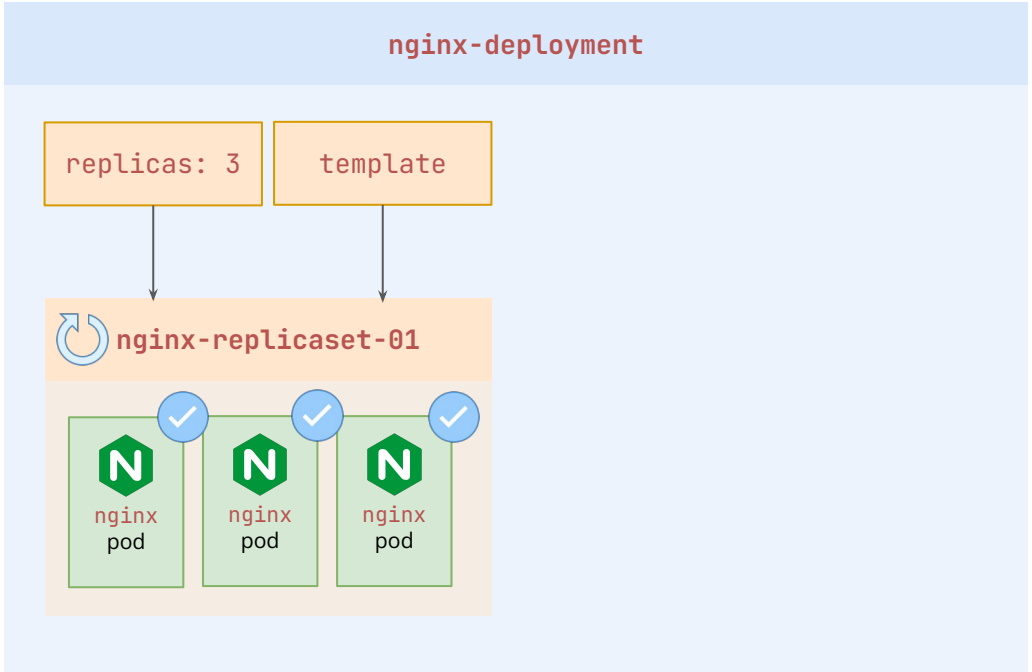
- ReplicaSets offer a great way to ensure a pre-defined number of pods are running at any given time. However, they do not provide all the needed functionality for managing applications at scale.
- Deployments offer a higher level of abstraction on top of ReplicaSets, and add the following features:
  - **Rolling updates:** Gradually replace old pods with new ones without downtime, ensuring the application remains available.
  - **Rollbacks:** roll back to a previous version in case something goes wrong during an update. This feature is crucial for maintaining application stability and quickly recovering from errors.
  - **Declarative updates** of replicas over time.
  - **History and revision control:** keep track of the history of all changes made, allowing you to view and revert to previous versions.
  - **Advanced rollouts:** limit risk by leveraging controlled rollouts of new versions, ensuring that updates are applied gradually and safely.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 labels:
   app: nginx
spec:
 replicas: 3
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers: ...
 strategy:
   type: RollingUpdate
   rollingUpdate: ...
```

Deployments

Manage and update applications at scale

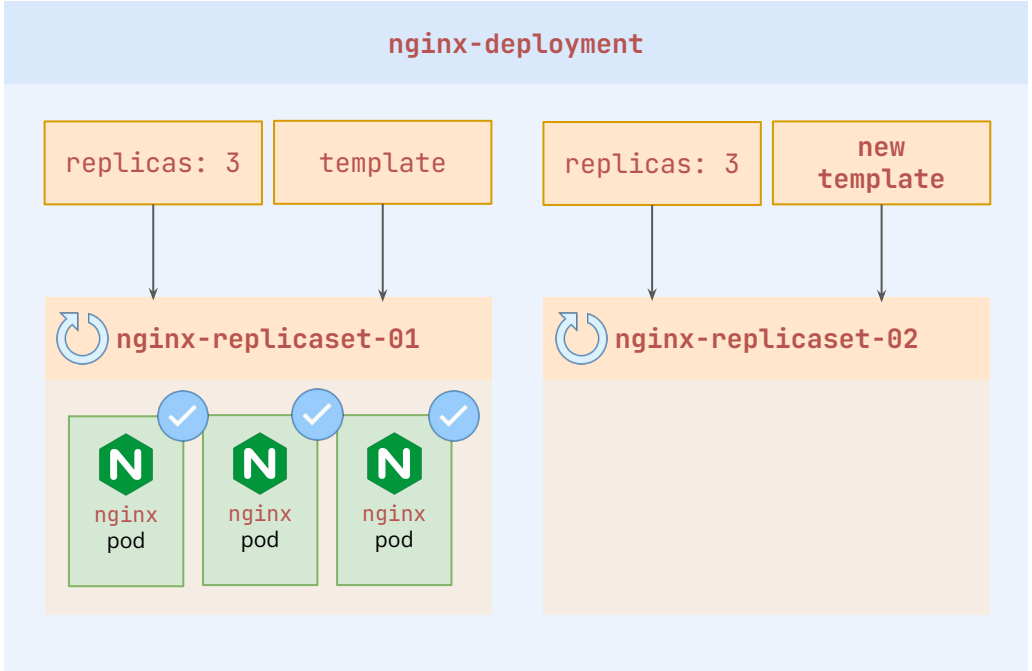
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 labels:
   app: nginx
spec:
 replicas: 3
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers:
       - ...
 strategy:
   type: RollingUpdate
```



Deployments

Manage and update applications at scale

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 labels:
   app: nginx
spec:
 replicas: 3
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers:
       - ...
 strategy:
   type: RollingUpdate
```

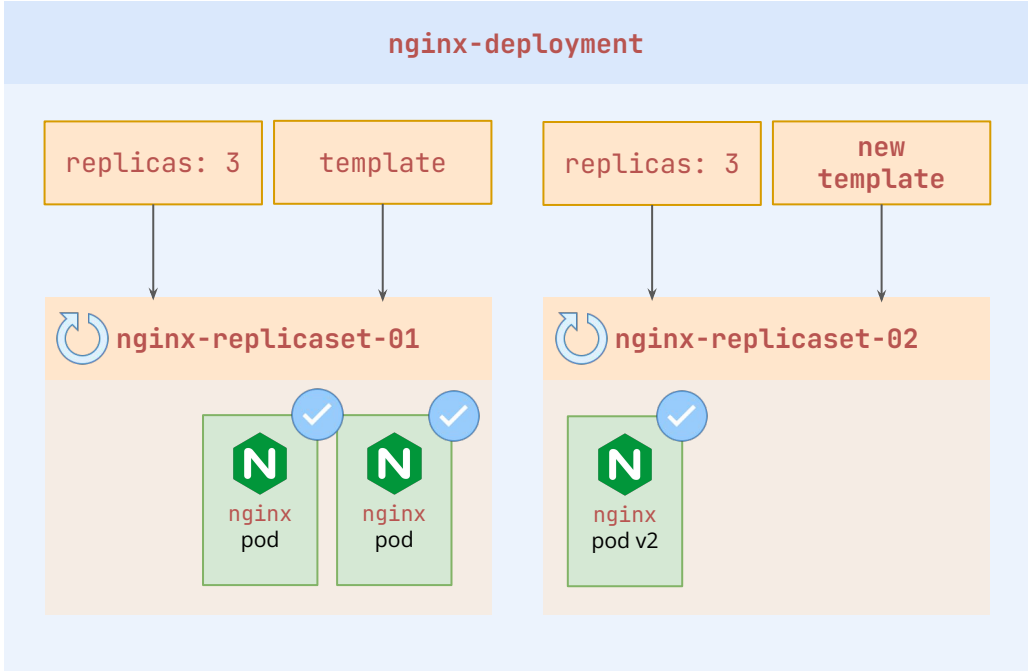


Kubernetes

Deployments

Manage and update applications at scale

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 labels:
   app: nginx
spec:
 replicas: 3
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers:
       - ...
 strategy:
   type: RollingUpdate
```

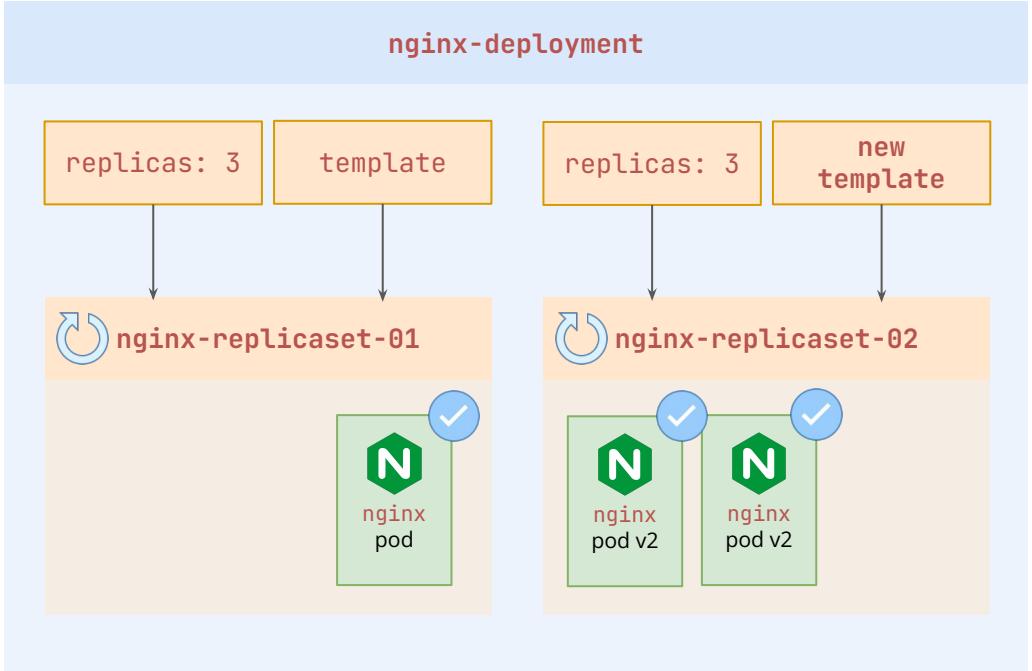


Kubernetes

Deployments

Manage and update applications at scale

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 labels:
   app: nginx
spec:
 replicas: 3
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers:
       - ...
 strategy:
   type: RollingUpdate
```

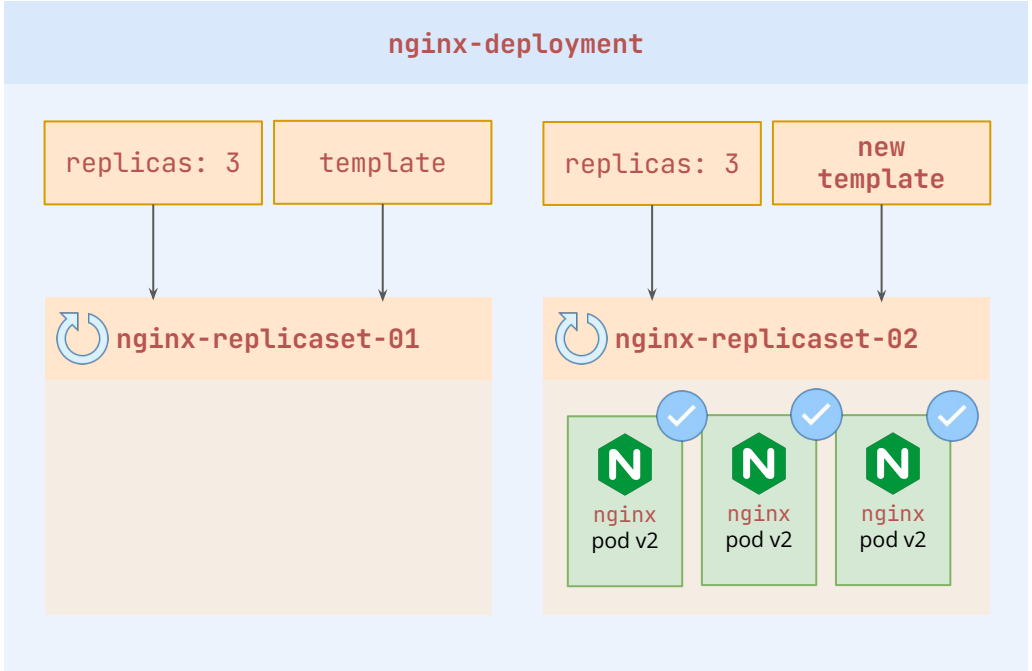


Kubernetes

Deployments

Manage and update applications at scale

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 labels:
   app: nginx
spec:
 replicas: 3
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers:
       - ...
 strategy:
   type: RollingUpdate
```

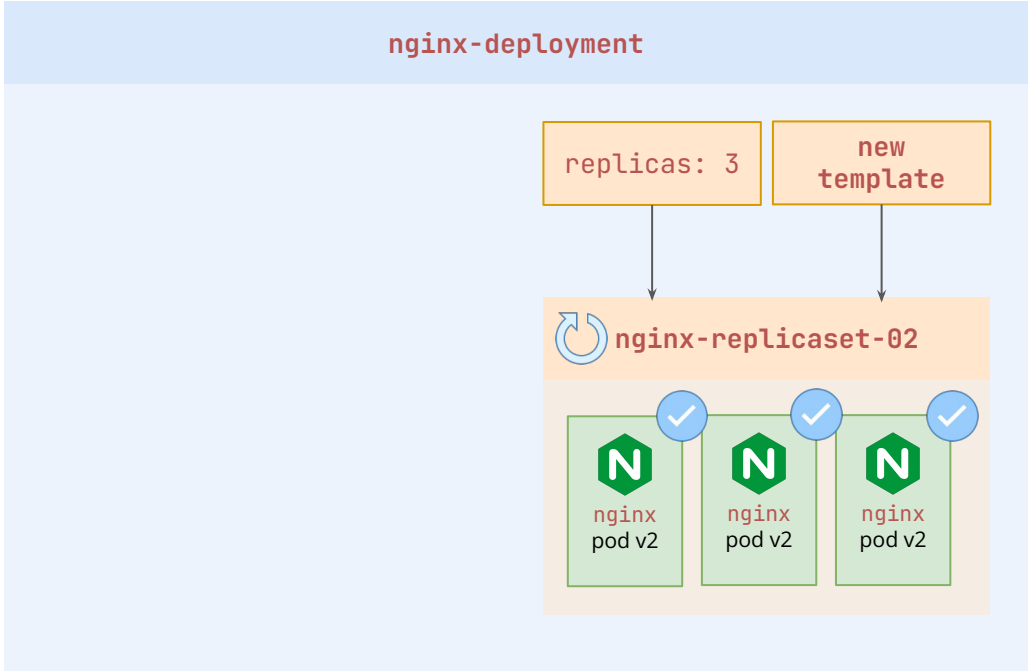


Kubernetes

Deployments

Manage and update applications at scale

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 labels:
   app: nginx
spec:
 replicas: 3
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers:
       - ...
 strategy:
   type: RollingUpdate
```

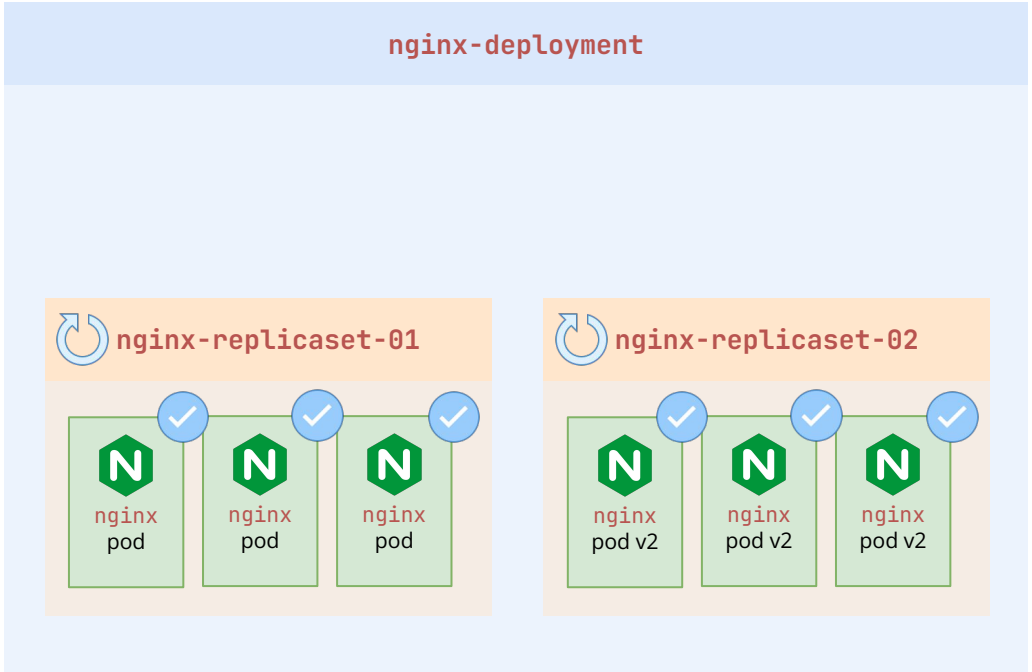


Kubernetes

Deployments

Manage and update applications at scale

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 labels:
   app: nginx
spec:
 replicas: 3
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers: ...
 strategy:
   type: RollingUpdate
   rollingUpdate: ...
```



Kubernetes

Services

Services

Section overview*

1. Discuss the multiple types of Services available in Kubernetes
2. Explore ClusterIP Services
3. Explore NodePort Services
4. Explore ExternalName Services

***We'll explore Headless Services when discussing StatefulSets, and LoadBalancer services when working with GKE.**

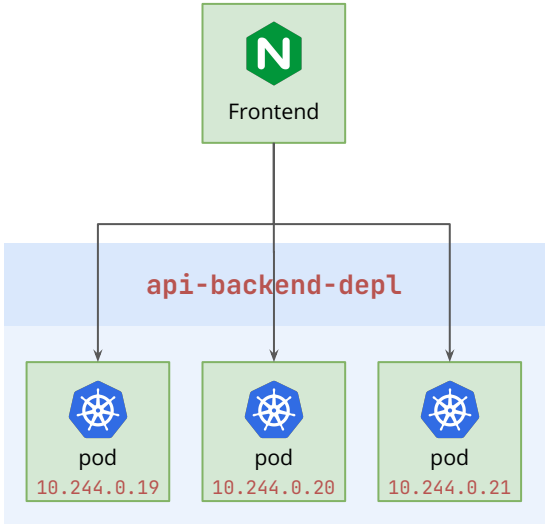
Kubernetes

Services

Services

Abstract pod details to facilitate connectivity

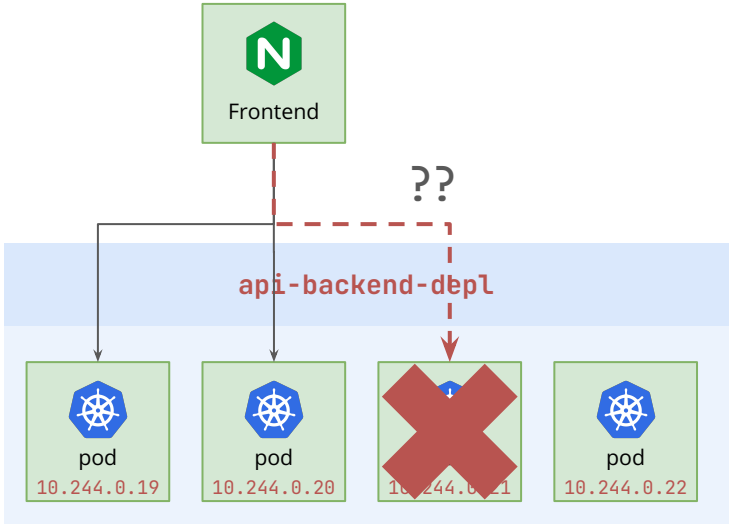
- Services allow us to expose, through a stable IP or internal DNS name, a network application that is running as one or more Pods in your cluster.



Services

Abstract pod details to facilitate connectivity

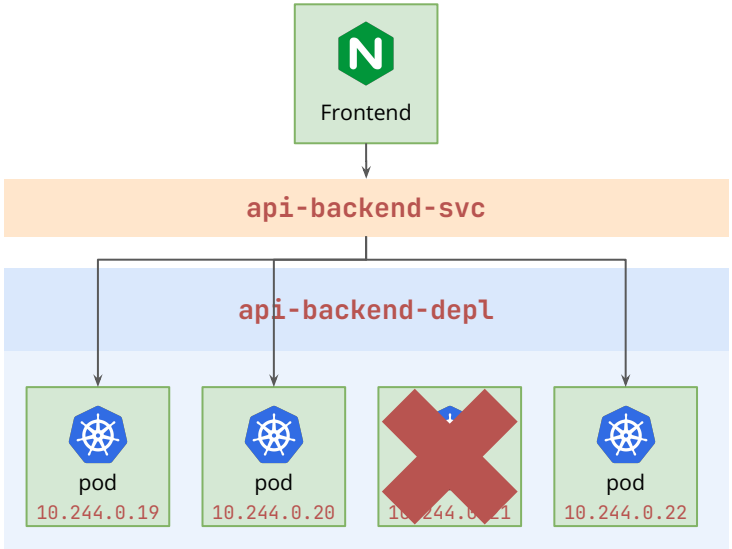
- Services allow us to expose, through a stable IP or internal DNS name, a network application that is running as one or more Pods in your cluster.



Services

Abstract pod details to facilitate connectivity

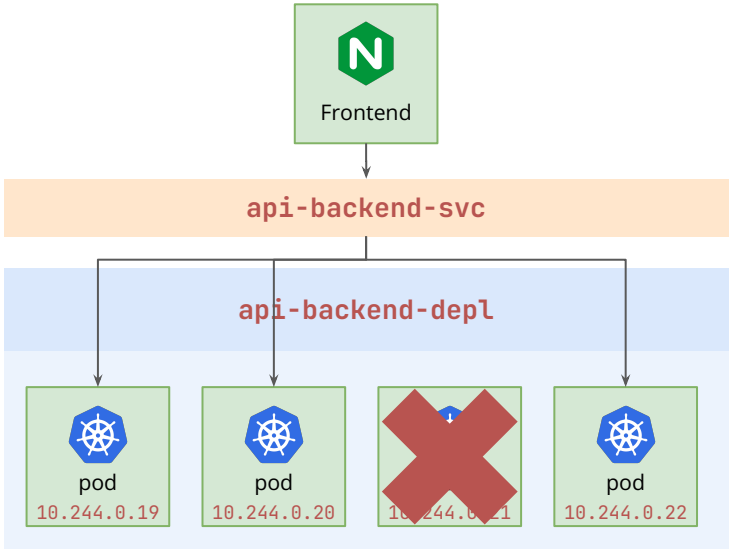
- Services allow us to expose, through a stable IP or internal DNS name, a network application that is running as one or more Pods in your cluster.



Services

Abstract pod details to facilitate connectivity

- Services allow us to expose, through a stable IP or internal DNS name, a network application that is running as one or more Pods in your cluster.




```yaml
apiVersion: v1
kind: Service
metadata:
  name: api-nodeport
spec:
  type: NodePort
  selector:
    app: api-backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30007
```

Services

Abstract pod details to facilitate connectivity

Different types of services available		
Service Type	**Behavior**	**When to use**
`ClusterIP`	■ Exposes the service on an internal, stable cluster IP. ■ It is only accessible within the cluster.	■ Best for internal communication between microservices within the cluster. ■ Often combined with ingresses to expose applications via HTTP and HTTPS.
`NodePort`	■ Exposes the service on each Node's IP at a static port. ■ If not specified, picks a port from the range 30000-32767. ■ Accessible from outside the cluster using `<NodeIP>:<NodePort>`.	■ Useful for exposing a service externally without a load balancer. ■ Often for development or testing.
`LoadBalancer`	■ Exposes the service externally using a cloud provider's load balancer. ■ Provides a single external IP to reach the service.	■ Ideal for production environments where external access with load balancing is required.
`ExternalName`	■ Maps the service to an external DNS name, redirecting traffic to a service outside the cluster.	■ Used to allow services within the cluster to access external services via a DNS name.

Kubernetes

Resource Management

Resource Management

Section overview

1. Explore how to work with labels and selectors
2. Discuss annotations and their role in managing objects
3. Explore namespaces
    1. Understand how namespaces can be used to isolate resources
    2. Discuss how to communicate with services across namespaces
4. Discuss resource quotas, requests and limits
    1. Learn how to set resource quotas in namespaces
    2. Explore how to set resource requests and limits in Pods
    3. Discuss common scenarios when managing compute and memory resources

Resource Management

Section overview

5 Explore health probes

1. Understand the lifecycle of each health probe
2. Practice working with startup probes
3. Practice working with liveness probes
4. Practice working with readiness probes

Kubernetes

Labels and Selectors

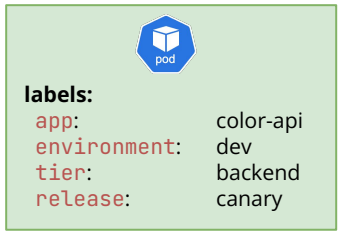
Labels and Selectors

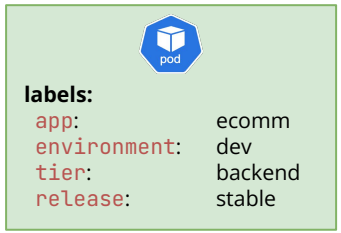
Identify and group resources meaningfully

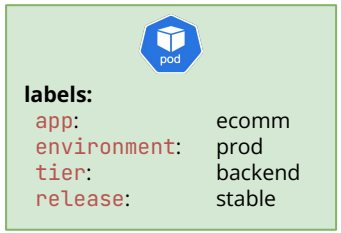
- Labels are key-value pairs attached to Kubernetes objects (e.g., pods, nodes, services). They provide metadata that helps identify and organize these objects.
- Labels allow Kubernetes users to categorize and organize resources, enabling sophisticated grouping and selection mechanisms. **Labels are not unique**, and multiple objects can have the same label.
- **Selectors** are expressions used to filter Kubernetes objects based on their labels. Selectors allow users and Kubernetes components to target specific objects that match certain criteria.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: api-server
  labels:
    app: color-api
    tier: backend
spec: ...
```

```yaml
apiVersion: v1
kind: Service
metadata: ...
spec:
  selector:
    app: color-api
    tier: backend
```



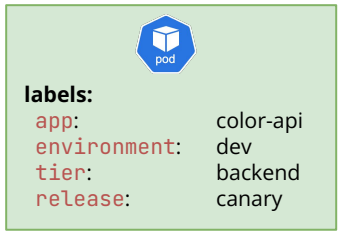


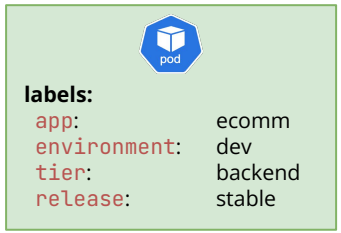


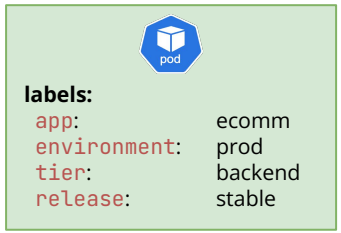
Labels and Selectors

Identify and group resources meaningfully

Equality-based selectors: Equality-based selectors match objects that have a specific label key-value pair. They use either `=`, `==`, or `!=` for inclusion or exclusion.





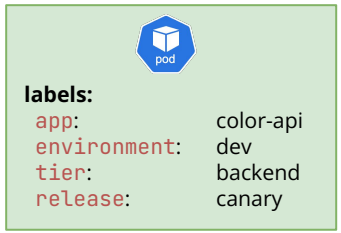


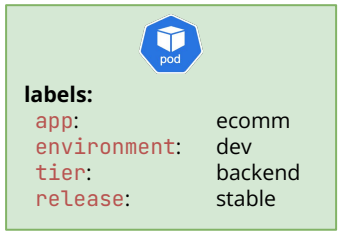
Labels and Selectors

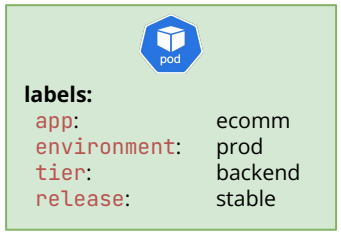
Identify and group resources meaningfully

Equality-based selectors: Equality-based selectors match objects that have a specific label key-value pair. They use either =, ==, or != for inclusion or exclusion.

```
selector:
  matchLabels:
    app: color-api
    tier: backend
```





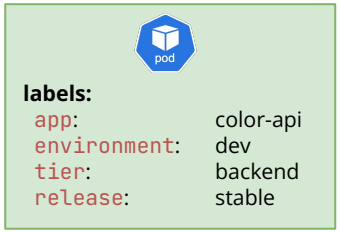


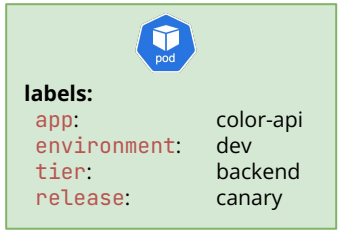
Labels and Selectors

Identify and group resources meaningfully

Equality-based selectors: Equality-based selectors match objects that have a specific label key-value pair. They use either =, ==, or != for inclusion or exclusion.

```
selector:
  matchLabels:
    app: color-api
    tier: backend
```



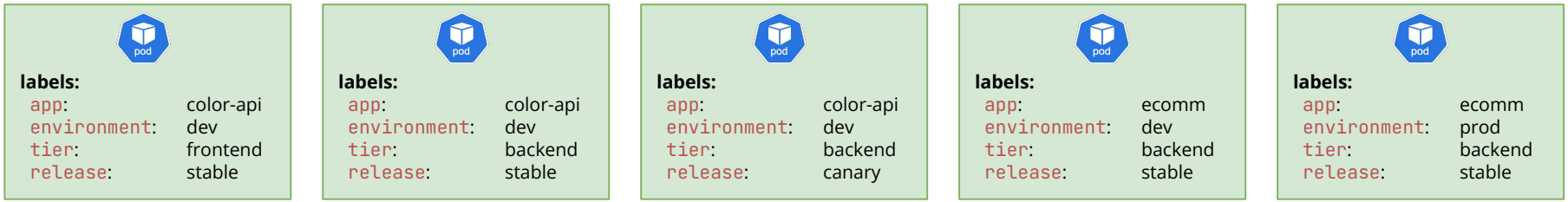
Labels and Selectors

Identify and group resources meaningfully

Equality-based selectors: Equality-based selectors match objects that have a specific label key-value pair. They use either `=`, `==`, or `!=` for inclusion or exclusion.

```
selector:
  matchLabels:
    app: color-api
    tier: backend
```

Set-based selectors: Set-based selectors match objects based on whether a label's key is part of a set of values. They use operators like `In`, `NotIn`, `Exists`, and `DoesNotExist`. Supported only by certain Kubernetes objects (e.g., Jobs, Deployments, ReplicaSets)



Labels and Selectors

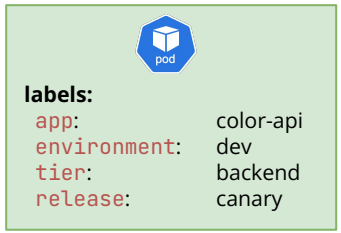
Identify and group resources meaningfully

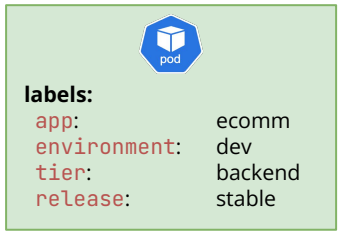
Equality-based selectors: Equality-based selectors match objects that have a specific label key-value pair. They use either `=`, `==`, or `!=` for inclusion or exclusion.

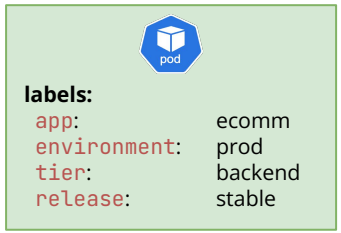
```
selector:
  matchLabels:
    app: color-api
    tier: backend
```

Set-based selectors: Set-based selectors match objects based on whether a label's key is part of a set of values. They use operators like `In`, `NotIn`, `Exists`, and `DoesNotExist`. Supported only by certain Kubernetes objects (e.g., Jobs, Deployments, ReplicaSets)

```
selector:
  matchExpressions:
    - key: tier
      operator: In
      values: [frontend, backend]
```







Labels and Selectors

Identify and group resources meaningfully

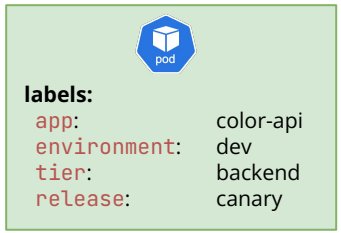
Equality-based selectors: Equality-based selectors match objects that have a specific label key-value pair. They use either `=`, `==`, or `!=` for inclusion or exclusion.

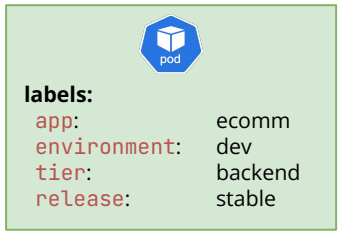
```yaml
selector:
  matchLabels:
    app: color-api
    tier: backend
```

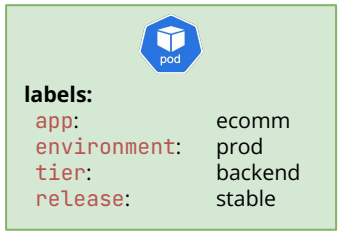
Set-based selectors: Set-based selectors match objects based on whether a label's key is part of a set of values. They use operators like `In`, `NotIn`, `Exists`, and `DoesNotExist`. Supported only by certain Kubernetes objects (e.g., Jobs, Deployments, ReplicaSets)

```yaml
selector:
  matchExpressions:
    - key: tier
      operator: In
      values: [frontend, backend]
```

```yaml
selector:
  matchExpressions:
    - key: release
      operator: NotIn
      values: [canary]
```







Labels and Selectors

Identify and group resources meaningfully

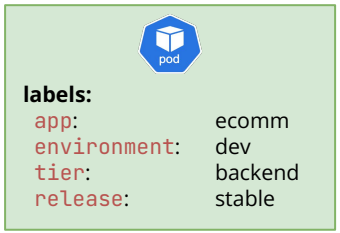
Equality-based selectors: Equality-based selectors match objects that have a specific label key-value pair. They use either `=`, `==`, or `!=` for inclusion or exclusion.

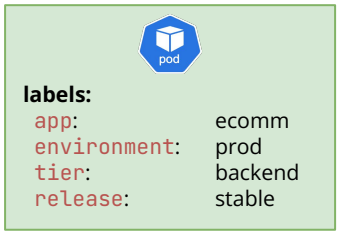
```yaml
selector:
  matchLabels:
    app: color-api
    tier: backend
```

Set-based selectors: Set-based selectors match objects based on whether a label's key is part of a set of values. They use operators like `In`, `NotIn`, `Exists`, and `DoesNotExist`. Supported only by certain Kubernetes objects (e.g., Jobs, Deployments, ReplicaSets)

```yaml
selector:
  matchExpressions:
    - key: tier
      operator: In
      values: [frontend, backend]
```

```yaml
selector:
  matchExpressions:
    - key: release
      operator: NotIn
      values: [canary]
```





Kubernetes

Annotations

Annotations

Add useful information and configuration to resources

- Annotations are key-value pairs attached to Kubernetes objects, just like labels. Unlike labels, annotations are not supposed to store identifying metadata, and they are often used by tools or the Kubernetes system itself.
- Common use-cases:
  - **Tool-specific metadata and configuration:** external tools (monitoring systems, logging agents) leverage annotations to attach custom data to resources (configurations, metrics collection endpoints, etc.)
  - **Configuration for ingress controllers:** used to configure ingress controllers (traffic routing, SSL termination, security settings). For example, you can define path rewrites or custom load balancing rules via annotations.
  - **Storing build and version information:** annotations can store metadata such as build timestamps, version numbers, or Git commit hashes.
  - **Runtime configuration for operators:** Kubernetes operators or controllers can use annotations to customize runtime behavior for specific resources.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
    nginx.ingress.kubernetes.io/proxy-body-size: 10m
spec: ...
```

Prefix

- Must be a DNS subdomain

Name

- Less than 63 characters
- Alphanumeric, dots, underscores and dashes

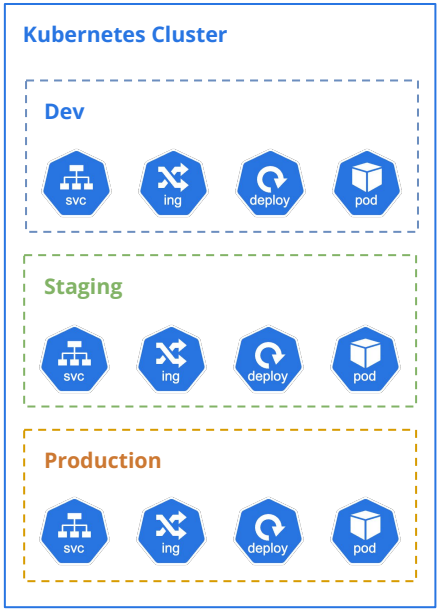
Kubernetes

Namespaces

Namespaces

Improve isolation for groups of resources

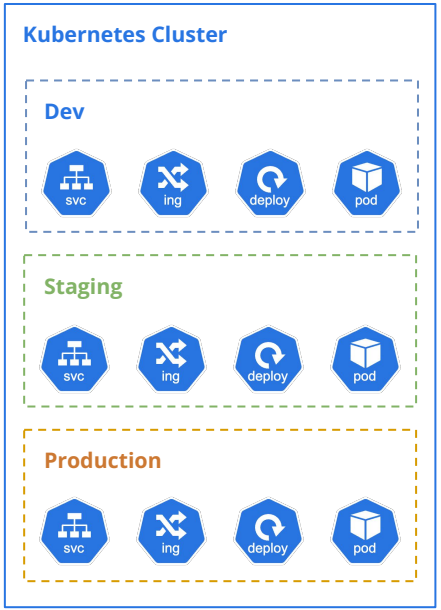
- Namespaces provide a way to divide cluster resources between multiple users, teams, applications, or environments. It allows for resource isolation and helps organize workloads in large Kubernetes clusters.
  - Namespaces enable logical isolation of resources within the same physical cluster.
  - Initial namespaces: `default`, `kube-system`, `kube-public`, and `kube-node-lease`.
- Common use-cases:
  - **Multi-tenant clusters:** Each team can take care of their applications, and keep their resources (pods, services, etc.) logically separated.
  - **Environment separation:** Each environment can have its own resources, with different policies and quotas applied.
  - **Resource Quotas:** We wish to limit the CPU, memory, and number of resources that a namespace can use. This prevents one team or environment from monopolizing cluster resources.
  - **Security and Access Control:** We wish to limit user or service account access to specific resources via RBAC mechanisms. This ensures that users or services can only access resources within their allowed namespaces.



Namespaces

Improve isolation for groups of resources

- Using namespaces:
  - We must inform in which namespace we wish to create our resources
  - We can set a current namespace for all `kubectl` commands, or pass it explicitly in each command
  - Service communication requires the fully qualified domain name (FQDN) of the service
- Best practices around namespaces:
  - **Don't overuse them:** just because we can create namespaces, doesn't mean we should. Consider whether there is a solid case for logically isolating the cluster resources.
  - **Combine namespaces with RBAC** to improve security around resources deployed in each namespace.
  - **Limit the resources a namespace can use** by implementing resource quotas to namespaces and resource requests and limits to their respective resources.
  - **Use meaningful dimensions to define the namespaces**, and make sure they align with the overall team and environment setup of your projects.



Kubernetes

Resource Quotas, Requests and Limits

Resource Quotas, Requests and Limits

Set limits on how much CPU, memory, and storage objects might use

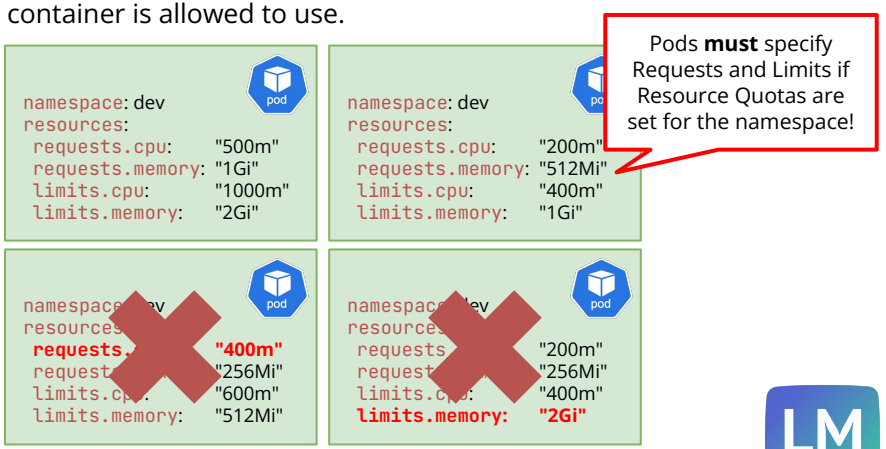
- **Resource Quotas**, **Requests**, and **Limits** ensure that resources such as CPU, memory, and storage are distributed fairly among applications and prevent a single application or namespace from consuming excessive resources.

Resource Quotas: policy **applied at the Namespace level** that defines hard limits on the number of resources (such as CPU, memory, and object counts) that can be consumed by all objects within that namespace.

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
 name: dev-quota
 namespace: dev
spec:
 hard:
   requests.cpu: "1000m"
   requests.memory: "2Gi"
   limits.cpu: "2000m"
   limits.memory: "4Gi"
```

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
 name: prod-quota
 namespace: prod
spec:
 hard:
   requests.cpu: "6000m"
   requests.memory: "12Gi"
   limits.cpu: "12000m"
   limits.memory: "24Gi"
```

Requests and Limits: defined at the container level within Pods, and specify how much of each resource a container needs. **Requests** specify the minimum amount of CPU or memory a container should always have available after scheduling, and **Limits** specify the maximum amount of CPU or memory that a container is allowed to use.



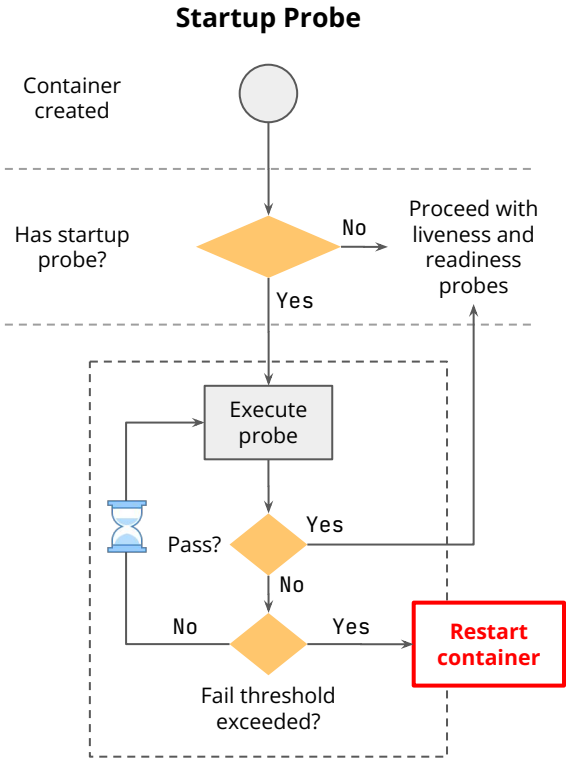
Kubernetes

Liveness, Readiness & Startup Probes

Liveness, Readiness and Startup Probes

Continuously monitor container health

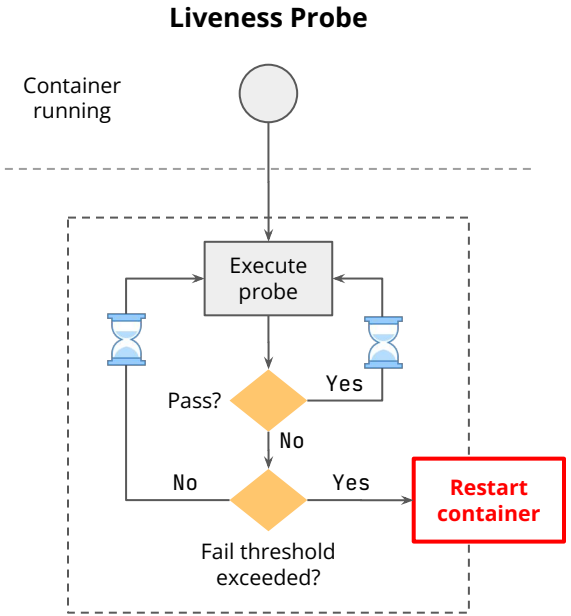
- Probes are periodic health checks performed by Kubernetes to determine the status of a container. They allow Kubernetes to manage the lifecycle of containers by checking if they are healthy and ready to serve traffic.
- Types of probes:
  - **Startup Probe:** Ensures that a container has started successfully. If the startup probe fails, Kubernetes will kill the container and try to restart it.
  - **Liveness Probe:** Checks if the container is still running. If the liveness probe fails, Kubernetes will restart the container. **Keeps executing throughout the container lifecycle.**
  - **Readiness Probe:** Checks if the container is ready to accept traffic. If the readiness probe fails, the container will be removed from the list of endpoints that receive traffic. **Keeps executing throughout the container lifecycle.**



Liveness, Readiness and Startup Probes

Continuously monitor container health

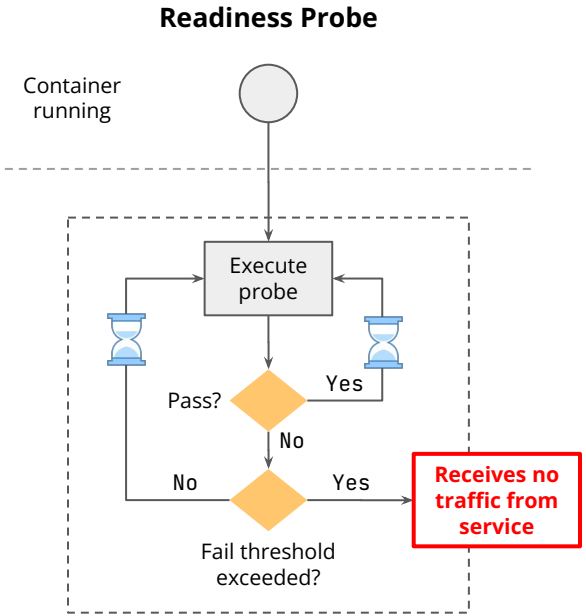
- Probes are periodic health checks performed by Kubernetes to determine the status of a container. They allow Kubernetes to manage the lifecycle of containers by checking if they are healthy and ready to serve traffic.
- Types of probes:
  - **Startup Probe:** Ensures that a container has started successfully. If the startup probe fails, Kubernetes will kill the container and try to restart it.
  - **Liveness Probe:** Checks if the container is still running. If the liveness probe fails, Kubernetes will restart the container. **Keeps executing throughout the container lifecycle.**
  - **Readiness Probe:** Checks if the container is ready to accept traffic. If the readiness probe fails, the container will be removed from the list of endpoints that receive traffic. **Keeps executing throughout the container lifecycle.**



Liveness, Readiness and Startup Probes

Continuously monitor container health

- Probes are periodic health checks performed by Kubernetes to determine the status of a container. They allow Kubernetes to manage the lifecycle of containers by checking if they are healthy and ready to serve traffic.
- Types of probes:
  - **Startup Probe:** Ensures that a container has started successfully. If the startup probe fails, Kubernetes will kill the container and try to restart it.
  - **Liveness Probe:** Checks if the container is still running. If the liveness probe fails, Kubernetes will restart the container. **Keeps executing throughout the container lifecycle.**
  - **Readiness Probe:** Checks if the container is ready to accept traffic. If the readiness probe fails, the container will be removed from the list of endpoints that receive traffic. **Keeps executing throughout the container lifecycle.**



Kubernetes

Storage and Persistence

Storage and Persistence

Section overview

1. Discuss how persistent storage is handled in Kubernetes
2. Practice working with the EmptyDir volume
3. Explore Persistent Volumes and Persistent Volume Claims
    1. Work with local volumes
    2. Practice dynamically creating Persistent Volumes in Minikube*
    3. Understand the configuration and lifecycle of Persistent Volumes and Persistent Volume Claims
4. Explore StatefulSets
    1. Understand the main characteristics and use-cases for StatefulSets
    2. Understand how headless services facilitate communication with StatefulSets
    3. Practice creating and managing StatefulSets and dynamic PVs

*** We'll also explore dynamic PVs when deploying our applications in GKE**

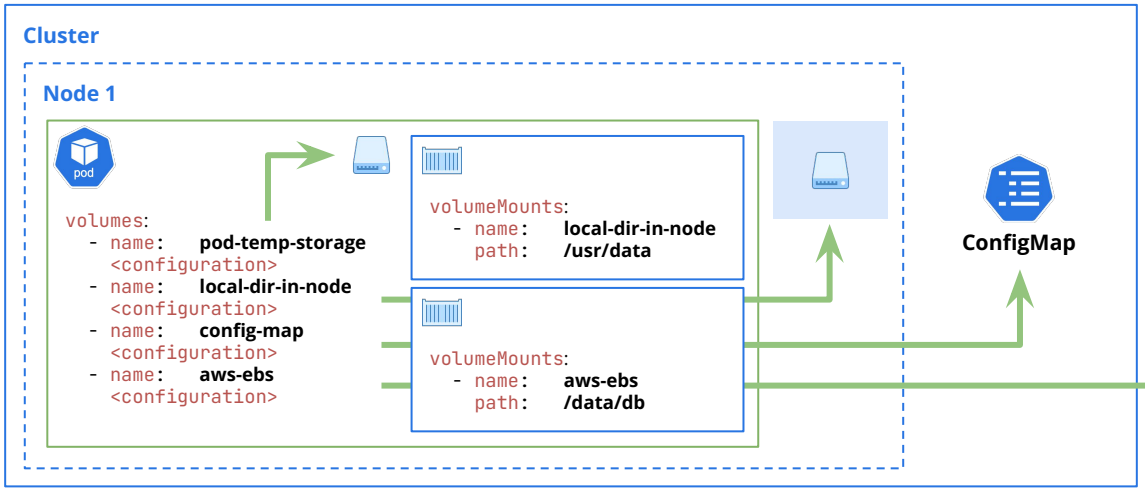
Kubernetes

Introduction to Volumes

Introduction to Volumes

Persist and share data in Kubernetes

- Volumes are directories, possibly with some data in it, which are **accessible to the containers in a pod**. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.
- To use a volume, specify the volumes in the Pod's `.spec.volumes` and declare where to mount those volumes into containers in `.spec.containers[*].volumeMounts`.



Kubernetes also supports many other storage backends, and we can create our own as long as we comply with the specification.

Introduction to Volumes

Persist and share data in Kubernetes

Different types of Volumes	
Volume Type	**When to use**
`emptyDir`	Ephemeral storage within the Pod. Follows the lifecycle of the Pod: when the Pod is terminated, its contents are also deleted.
`local`*	Durable storage within a specific node. Requires setting node affinity to correctly schedule Pods on the correct nodes. Preferred over `hostPath`
`Persistent Volume`	Durable storage backed by multiple technologies (e.g., cloud storage). Can be statically or dynamically provisioned.
`ConfigMap`	Inject configuration data into Pods, without having to hard-code the data into the Pod definition.
`Secret`	Inject sensitive data into Pods, without having to hard-code the data into the Pod definition.

*** `hostPath` may introduce many security vulnerabilities and is currently discouraged.**

Introduction to Volumes

Persist and share data in Kubernetes

Different types of Volumes	
Volume Type	**When to use**
`emptyDir`	Ephemeral storage within the Pod. Follows the lifecycle of the Pod: when the Pod is terminated, its contents are also deleted.
`local`*	Durable storage within a specific node. Requires setting node affinity to correctly schedule Pods on the correct nodes. Preferred over `hostPath`
`Persistent Volume`	Durable storage backed by multiple technologies (e.g., cloud storage). Can be statically or dynamically provisioned.
`ConfigMap`	Inject configuration data into Pods, without having to hard-code the data into th
`Secret`	Inject sensitive data into Pods, without having to hard-code the data into the Po

We'll cover working with ConfigMaps and Secrets later in the course!

*** `hostPath` may introduce many security vulnerabilities and is currently discouraged.**

Kubernetes

EmptyDir and Local

EmptyDir and Local

Pod- and Node-level Storage

- `emptyDir` volumes are **ephemeral** and defined at the Pod level.
  - The volume is created when the Pod is assigned to a node.
  - The volume is initially empty.
  - All containers in the Pod can read and write the same files in the `emptyDir` volume.
  - Containers might mount the volume in different paths.
  - When a Pod is removed from a node, the data in the `emptyDir` is deleted permanently.
- `local` volumes are **persistent** and defined at the Node level.
  - kube-scheduler knows how to assign Pods to Nodes based on the affinity constraints defined in the `PersistentVolume` configuration.
  - Setting a `PersistentVolume` `nodeAffinity` is mandatory when using local volumes.
  - Like other `PersistentVolume`s, it requires creating `PersistentVolumeClaim`s so that Pods can use the storage.
  - Only supports static provisioning.

Data follows the Pod's lifecycle

Data follows the Node's lifecycle

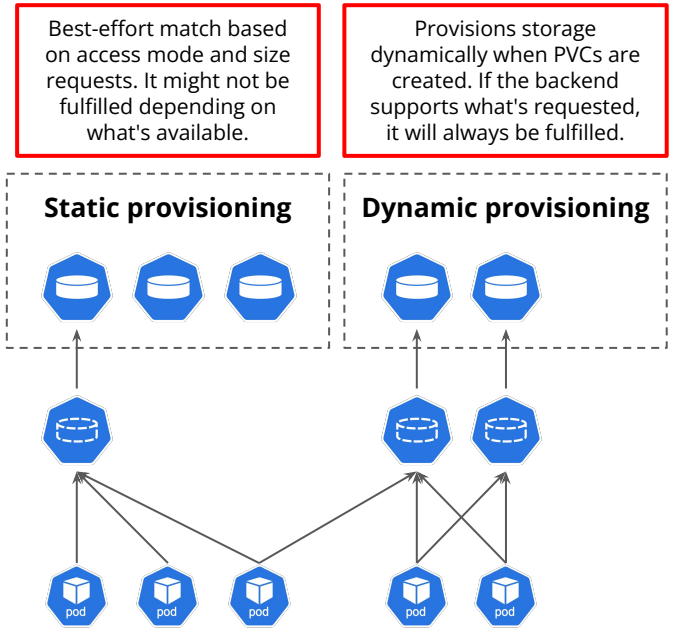
Kubernetes

Persistent Volume Claims

Persistent Volume Claims

Claim durable storage for usage in Pods

- A `PersistentVolumeClaim` is what actually reserves a PV (when using static provisioning) or creates and reserves a PV (when using dynamic provisioning).
- Claims are bound to a single `PersistentVolume`, and a `PersistentVolume` can have **at most one** claim bound to it.
  - We can set specific criteria in a claim, so that only PVs that match these criteria are considered for binds.
  - Be mindful of the possibility for extra unused capacity!
  - Reclamation policies can be `Retain`, `Delete`, or `Recycle` (deprecated).
- Access modes:
  - `ReadWriteOnce`: the volume can be mounted as read-write by a single node, and can be used by any number of Pods within that node.
  - `ReadOnlyMany`: the volume can be mounted as read-only by many nodes.
  - `ReadWriteMany`: the volume can be mounted as read-write by many nodes.
  - `ReadWriteOncePod`: the volume can be mounted as read-write by a single Pod.



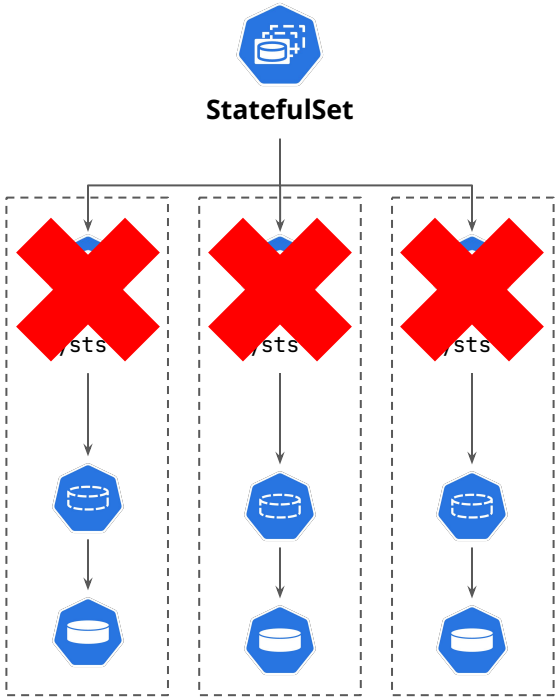
Kubernetes

StatefulSets

StatefulSets

Manage stateful applications more effectively

- `StatefulSets` maintain stable identities for each pod, ensuring that even if a pod is restarted, it retains its unique identity and connection to persistent storage. More specifically, `StatefulSets` provide:
  - A stable, unique network identity for each pod created as part of the `StatefulSet`.
  - Consistent and stable persistent storage associated with each pod, also across restarts.
  - Ordered pod creation, scaling, and deletion, should that be necessary when working with databases or replicated services.
- We can specify a `PersistentVolumeClaim` template in the Pod definition, and as long as this stays the same, the Pod will use the same claim and have access to the same data.
  - Each replica in a `StatefulSet` will have its own `PersistentVolumeClaim`, so data is not shared across different replicas.
  - The `PersistentVolumes` are not deleted automatically when a replica is deleted!
- Pod names and networking identities are also stable, and will follow the following pattern: `<statefulset name>-<ordinal id>`
  - We can use headless services to expose the pods via more stable domains.



Kubernetes

Configuration Management

Configuration Management

Section overview

1. Discuss native options for configuration and sensitive data management in Kubernetes
2. Explore working with ConfigMaps
    1. Create and manage ConfigMaps in Kubernetes
    2. Pass data from ConfigMaps via environment variables
    3. Mount ConfigMaps as volumes in containers
3. Explore working with Secrets
    1. Create and manage Secrets in Kubernetes
    2. Pass data from Secrets via environment variables
    3. Mount Secrets as volumes in containers

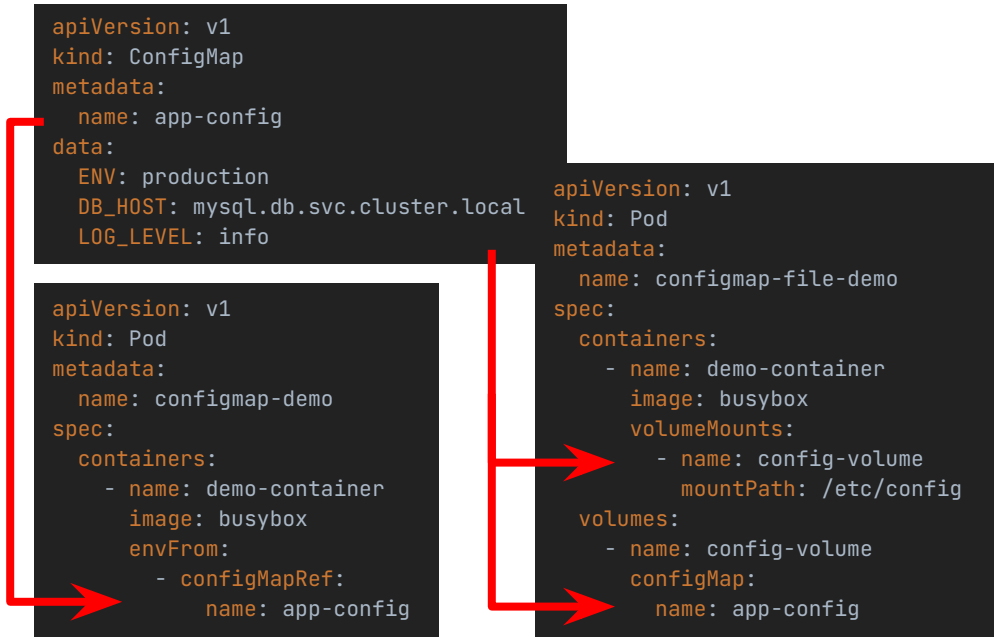
Kubernetes

ConfigMaps

ConfigMaps

Decouple and inject configuration data into Pods

- `ConfigMaps` can be used to store non-sensitive data in key-value pairs and decouple this data from the Pod definitions and lifecycle.
- `ConfigMaps` can be referenced in multiple ways:
  - Passed as environment variables
  - Passed as files via volume mounts
- Data cannot exceed 1MB in size.
- Pods must be in the same namespace as the `ConfigMaps` they reference.
  - **Exception:** when fetching values directly via the Kubernetes API.
- We can also set a `ConfigMap` as immutable so that it cannot be updated, and must be deleted and recreated.



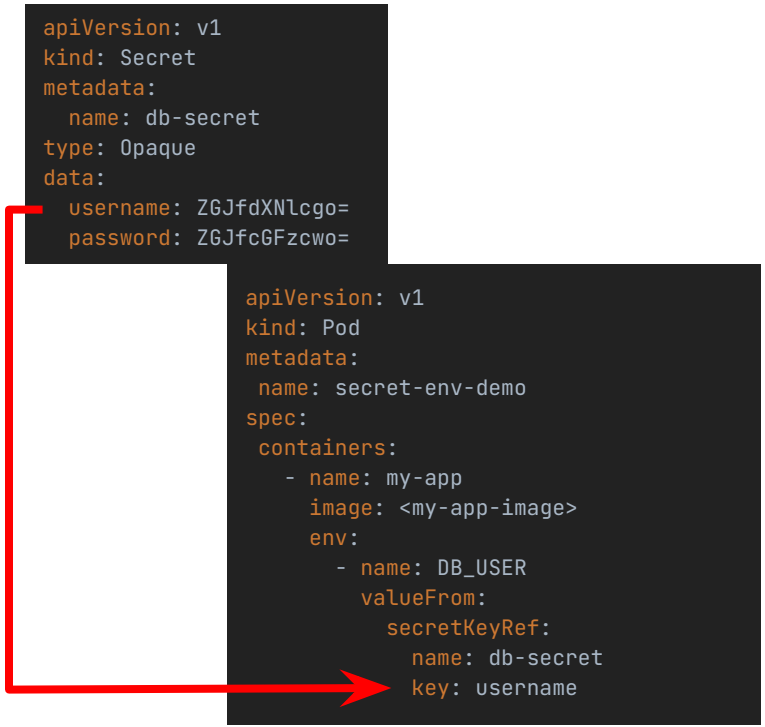
Kubernetes

Secrets

Secrets

Decouple and inject sensitive data into Pods

- `Secrets` can be used to store and inject sensitive data into containers. They help us not have to include sensitive information in application code.
- Data is stored in base64-encoded and unencrypted by default, but it is possible to set up encryption at rest for secrets.
- It's best practice to set up RBAC rules with least-privilege permissions, so only the authorized parties are allowed to retrieve or update the values of secrets.
- Accessing `Secrets` is similar to accessing `ConfigMaps` (both objects work similarly to each other from the Pod's / container's perspective), and they can be:
  - Passed as environment variables
  - Passed as files via volume mounts
- In addition to generic secrets, Kubernetes also has other specific secret types, such as `ServiceAccount` tokens, TLS secrets, among others.
- Cloud providers and their managed Kubernetes offerings normally provide native integration with secrets managers for improved and more secure secret storage.



Kubernetes

Security Fundamentals

Security Fundamentals

Section overview

1. Understand the multiple dimensions of handling security in Kubernetes clusters
2. Discuss and practice Role-Based Access Control (RBAC)
    1. Learn the concepts of roles, cluster roles, and their respective bindings
    2. Practice creating and managing them in the cluster
    3. Explore how to link roles to users, groups, and service accounts
3. Deep dive into Kubernetes' API structure
    1. Understand API groups and how they relate to RBAC
4. Explore Service Accounts
    1. Understand the goal of Service Accounts
    2. Explore how to create, manage, and use Service Accounts in Pods

Security Fundamentals

Section overview

5 Deep dive into Network Policies

1. Understand the need for and purpose of Network Policies
2. Understand how ingress and egress Network Policies work
3. Practice creating, updating, and managing Network Policies

6 Discuss Pod Security Standards

1. Explore what Pod Security Standards are and how to enforce them
2. Practice deploying compliant and non-compliant Pods

Kubernetes

Introduction to Security in K8s

Introduction to Security in K8s

Understanding security aspects of running K8s clusters

- So far, we have been working with the default configuration shipped with Minikube. These are very permissive settings, and should not be used in real-world clusters.
- Additionally, Kubernetes has many components and runs across multiple layers (control plane, nodes, network, applications, developers), and all of which require strong security measures.
- So... What could wrong in Kubernetes?

Introduction to Security in K8s

Understanding security aspects of running K8s clusters

Common Security Risks	
Risk	**Common Causes**
Exposed Control Plane and API Access: Unauthorized access to the Kubernetes API server can compromise the entire cluster.	⭕ API server accessible over the internet without proper auth. ⭕ Weak or default credentials being used.
Insecure Workloads: Running containers as root or with excessive privileges can allow for privilege escalation and other attacks.	⭕ Containers running with default root user or running in privileged mode. ⭕ Mounting sensitive host directories into containers via volumes.
Overly Permissive Roles: Roles with excessive permissions can lead to data exposure if compromised.	⭕ Assigning broad permissions to users or service accounts. ⭕ Lack of role separation and adherence to the principle of least privilege.
Lack of Network Segmentation and Pod Isolation: Attackers can move laterally across services and pods once inside the network.	⭕ Default network settings allow unrestricted communication between pods. ⭕ Absence of Network Policies to control traffic flow.
Unsecured Data at Rest and In Transit: Data stored in etcd or persistent volumes can be accessed if not encrypted.	⭕ Default settings without encryption enabled. ⭕ Use of insecure communication protocols.

Introduction to Security in K8s

Understanding security aspects of running K8s clusters

- Kubernetes offers many constructs we can use to tackle several of these problems (the list below is not extensive! There are many more tools we can leverage in the Kubernetes ecosystem):
  - **Role-Based Access Control (RBAC):** allows us to apply the principle of least privilege by defining roles and binding them to specific subjects.
  - **Network Policies:** allow us to control traffic flow between pods and network endpoints. We can start by denying all traffic, and then allow only the necessary traffic to flow between Pods.
  - **Encryption at Rest:** allows us to protect the data stored in the cluster's etcd store and persistent volumes.
  - **Pod Security Standards (PSS):** allows us to enforce, warn, or audit certain security standards on a namespace basis when creating new pods.
  - **Image Security:** allows us to leverage images that have as few as possible known vulnerabilities, and to swiftly accommodate security patches.

Kubernetes

Role-Based Access Control

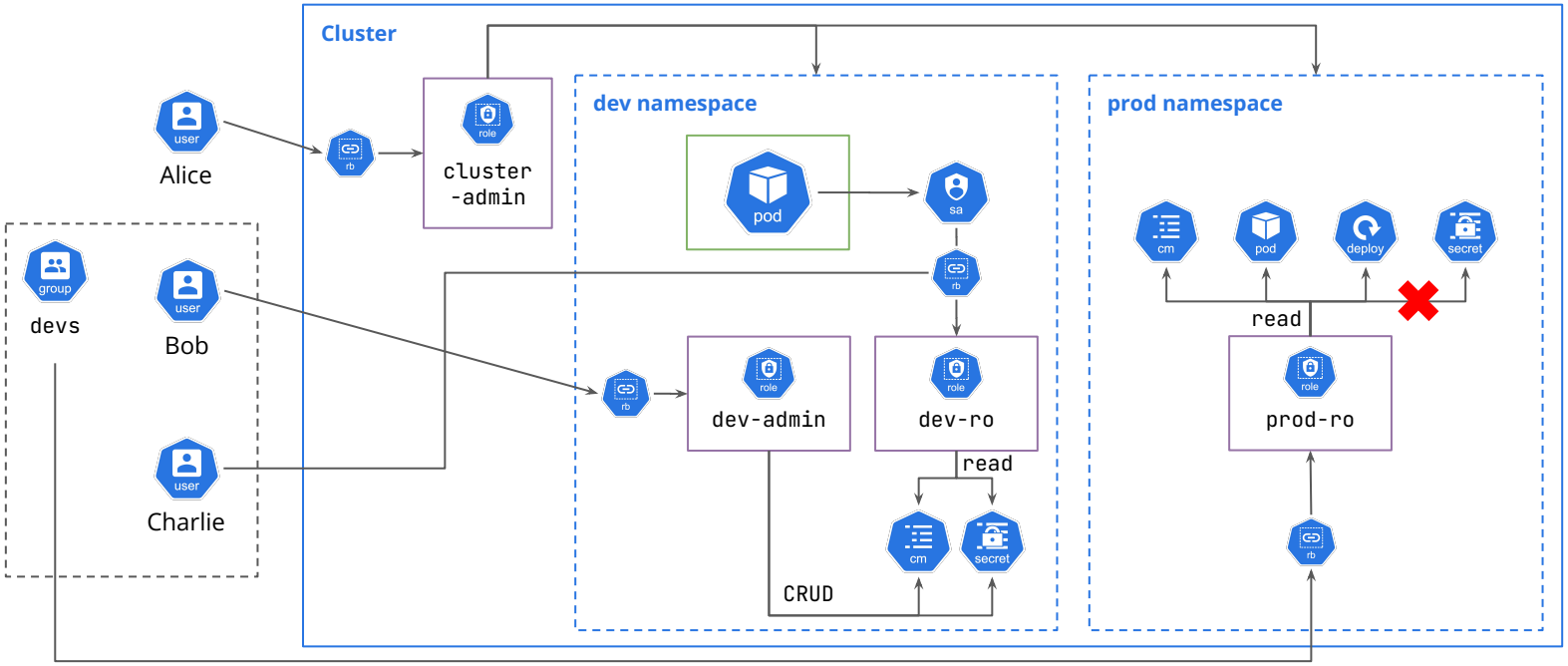
Role-Based Access Control

Define permissions for users, groups, and service accounts

- **Role-Based Access Control (RBAC)** is a security model in Kubernetes that defines and enforces permissions for users, service accounts, and groups to interact with cluster resources.
- Permissions can target many different resources (pods, deployments, secrets, configmaps, and many others) and operations (read, create, update, delete, as well as specific operations such as exec) in the cluster, allowing for very fine-grained access control.
- **RBAC is a must** for any serious Kubernetes cluster.
- Five key components of **RBAC:**
  - **Users, groups, and service accounts:** the entities against which RBAC policies will be enforced
  - **Roles:** defines a set of permissions within a specific namespace. It specifies the actions (verbs), such as `get`, `list`, `create`, that can be performed on particular Kubernetes resources (like pods, services, secrets).
  - **RoleBindings:** assigns a `Role` to a user, group, or service account within a namespace.
  - **ClusterRoles:** similar to `Roles`, but has effect across the entire cluster. They can be used to grant permissions to resources that are not namespaced, or to resources across all namespaces.
  - **ClusterRoleBindings:** assigns a `ClusterRole` to a user, group, or service account.

Role-Based Access Control

Define permissions for users, groups, and service accounts



Kubernetes

The Kubernetes API

The Kubernetes API

Understand how the API paths and groups are structured

- The Kubernetes API is the interface used to interact with and manage the resources in a Kubernetes cluster.
  - `kubectl`, the Kubernetes dashboard, or external services all perform operations through API calls.These API requests are sent to the Kubernetes API server, which processes them and communicates with the rest of the cluster components.
- Every resource in Kubernetes (pods, services, configmaps, deployments, and others) is accessible via the API under its corresponding RESTful endpoint.
- We can pass the `--v=8` command-line flag to `kubectl` to get more details about the underlying HTTP request made by the CLI

API call examples	
kubectl command	**Respective API call**
`kubectl get pod`	`GET https://<api-server>/api/v1/namespaces/default/pods`
`kubectl get pod -A`	`GET https://<api-server>/api/v1/pods`
`kubectl get role`	`GET https://<api-server>/apis/rbac.authorization.k8s.io/v1/namespaces/default/roles`

The Kubernetes API

Understand how the API paths and groups are structured

- Kubernetes organizes its API resources into logical groups known as API Groups.
  - API Groups allow Kubernetes to extend and evolve without breaking existing resources or creating conflicts.
  - Different API Groups can define related resources, and each Group typically represents a versioned API with its own path.
  - Groups are referenced in the `apiVersion` field of Kubernetes manifests: `apiVersion: <group name>/<version>`
- Resources are split into the **core** API group (no group name in the apiVersion of our manifests) and **named** API groups.

Core API group

- Default API Group for Kubernetes' most essential resources, such as pods, services, configmaps, and namespaces.
- Accessible under the `/api` top-level path of the Kubernetes API.

```
https://<api-server>/api/v1/pods
https://<api-server>/api/v1/namespaces
https://<api-server>/api/v1/namespaces/default/secrets
```

Named API groups

- Contain additional resources and extend the functionality of Kubernetes.
- Accessible under the `/apis` top-level path of the Kubernetes API and under their own group path.

```
https://<api-server>/apis/apps/v1/deployments
https://<api-server>/apis/apps/v1/namespaces/default/deployments
```

The Kubernetes API

Understand how the API paths and groups are structured

- Some resources also have subresources, allowing us to be very fine-grained when setting permissions via RBAC roles.
  - Pods have, among others, the following subresources:

Subresource	API endpoint
`logs`	`/api/v1/namespaces/{namespace}/pods/{name}/log`
`exec`	`/api/v1/namespaces/{namespace}/pods/{name}/exec`
`attach`	`/api/v1/namespaces/{namespace}/pods/{name}/attach`

  - Deployments have, among others, the following subresources:

Subresource	API endpoint
`scale`	`/apis/apps/v1/namespaces/{namespace}/deployments/{name}/scale`
`status`	`/apis/apps/v1/namespaces/{namespace}/deployments/{name}/status`

Kubernetes

Service Accounts

Service Accounts

Interact with Kubernetes clusters from within applications and services

- When applications or processes inside Kubernetes pods need to perform actions on resources (such as reading configmaps, writing logs, or scaling deployments), they need permissions to access the Kubernetes API. This access is handled through service accounts rather than traditional user credentials.
- Service accounts authenticate via JWT tokens, which are automatically generated and mounted by Kubernetes inside each pod using a service account.
  - Mounted under `/var/run/secrets/kubernetes.io/serviceaccount/token`
  - We can use this token to authenticate our requests against the Kubernetes API.
- Some key differences between service accounts and user accounts:
  - **User accounts** are for humans. **Service accounts** are for application processes, which (for Kubernetes) run in containers that are part of pods.
  - **User account** names must be unique across all namespaces of a cluster. **Service accounts** are namespaced: two different namespaces can contain ServiceAccounts that have identical names.
  - User and Service accounts have different lifecycles, and Service Accounts are more lightweight and easier to create. This makes it easier for workloads to follow the principle of least privilege.

Kubernetes

Network Policies

Network Policies

Regulate ingress and egress traffic for Pods

- Network Policies define how pods can communicate with each other and with other network endpoints. They can specify rules for both ingress (incoming) and egress (outgoing) traffic.
  - By default, all traffic between pods is allowed in Kubernetes. Network Policies provide one solution to isolate pods and reduce the attack surface in case a pod becomes compromised.
- In order for Network Policies to have any effect, the CNI plugin used in the cluster must support this feature.
  - Examples of solutions that support this include Calico, Cilium and Weave Net.
- A Network Policy includes three main configuration pieces:
  - Which pods the policy applies to.
  - [Optional] Ingress rules.
  - [Optional] Egress rules.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-allow-backend
spec:
  podSelector:
    matchLabels:
      app: mongodb
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: backend
      ports:
        - protocol: TCP
          port: 27017
```

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-egress
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Egress
  egress:
    - to:
        - podSelector:
            matchLabels:
              app: mongodb
      ports:
        - protocol: TCP
          port: 27017
```

Network Policies

Regulate ingress and egress traffic for Pods

- There are multiple ways we can configure the rules for ingress and egress traffic:
  - Pod Selector: use `matchLabels` and `matchExpressions`
  - Namespace Selector: use `matchLabels` and `matchExpressions`
  - IP Block: use CIDR blocks
  - Ports and Protocols: define specific ports and protocols
- Different list elements under the `from` or `to` conditions behave as an `OR` condition, while multiple selection options under the same list element behave as an `AND` condition.

```
ingress:
 - from:
     - namespaceSelector:
         matchLabels:
           name: staging
       podSelector:
         matchLabels:
           app: frontend
```

```
ingress:
 - from:
     - namespaceSelector:
         matchLabels:
           name: staging
     - podSelector:
         matchLabels:
           app: frontend
```

Kubernetes

Pod Security Standards

Pod Security Standards

Leverage different security levels to ensure security best practices in Pods

- Pod Security Standards (PSS) offer a set of guidelines that define different security contexts for pods. They can be used to ensure that pods adhere to specific security standards, reducing the risk of potential vulnerabilities and misconfigurations.

Overview of Pod Security Standards	
Profile	**Characteristics**
`Privileged`	Imposes no restrictions on the pod's security settings. This is the least restrictive policy and allows full access to system resources. It is suitable for pods that require elevated privileges, such as those that manage the cluster or perform administrative tasks.
`Baseline`	Enforces a minimal set of security restrictions that are suitable for general-purpose workloads. This policy allows common container configurations while preventing escalations in privilege, such as mounting sensitive host paths or modifying the host's network settings.
`Restricted`	Enforces the strictest set of security standards. It is designed for applications that run in highly sensitive or regulated environments and require maximum security controls.

Pod Security Standards

Leverage different security levels to ensure security best practices in Pods

- Pod Security Standards (PSS) are enforced via the Pod Security Admission Controller.
  - The controller is enabled by default in later Kubernetes versions.
- The admission control uses labels for configuration, and is set on a namespace basis:
  - `pod-security.kubernetes.io/<MODE>: <LEVEL>`
  - Available modes:
    - `enforce`: blocks the creation of pods that violate the security standards.
    - `warn`: Shows a user-facing warning if violations occur, but still allows the pod to be created.
    - `audit`: Logs a security violation in Kubernetes audit logs, but still allows the pod to be created.
  - Available levels: `privileged`, `baseline`, `restricted`.
- We should combine PSS with other security mechanisms, such as RBAC, to properly enforce access rules to namespaces that accept privileged workloads.
- It's possible to configure the built-in admission controller to set cluster-wide defaults, and then override these defaults at the namespace level.

Kubernetes

Kustomize

Kustomize

Section overview

1. Discuss what is Kustomize and the benefits it brings
2. Understand how to leverage bases and overlays for customization
3. Learn about common transformations we can easily apply
4. Learn how to work with config map and secret generators
5. Deep dive into working with patches
    1. Practice how to create inline patches
    2. Practice how to work with strategic merge patches
    3. Practice how to work with JSON patches

Kubernetes

Introduction to Kustomize

Introduction to Kustomize

Declaratively customize and manage multiple Kubernetes resources

- Kustomize allows you to customize K8s manifests without needing to duplicate and modify the original files or their copies.
- It follows the principle of declarative management, in which you declare which customizations and patches you wish to apply, and Kustomize calculates the final resulting manifests.
- Kustomize addresses the challenges of **managing configurations of similar applications for different environments**, which can lead to a lot of **duplication** and **overhead** when using only Kubernetes-native manifests.
- Key concepts in Kustomize:
  - Bases and Overlays: A **base** is a set of common YAML configurations that are shared across environments. **Overlays** apply environment-specific customizations on top of a **base**. For example, your dev overlay might use a lower replica count, while your production overlay might use a higher one.
  - No Templating: Since Kustomize uses only standard YAML and does not require templating syntax (like Helm does), it has a lower learning curve and simpler configuration files.
  - Resource Patching: Kustomize allows you to modify only specific parts of a resource.
  - Label and Annotation Management: we can set common labels and annotations across multiple resources to make resource grouping and tracking easier.
  - Handling Secrets and ConfigMaps: With Kustomize, you can dynamically generate ConfigMaps and Secrets from files or environment variables, making it easier to manage sensitive information and configuration data.

Introduction to Kustomize

Declaratively customize and manage multiple Kubernetes resources

Kustomize vs. Helm		
Dimension	**Kustomize**	**Helm**
Overall purpose	Customize existing Kubernetes YAML manifests by overlaying changes also defined in YAML.	Package manager for Kubernetes with support for templating, dependency management, and versioning of applications.
Complexity	Simpler to work with, since it leverages only native YAML constructs and does not introduce templating languages.	More complex to work with, since it introduces the need to learn Go templates and the overall structure of charts.
Customization features	Strategic merge patches, JSON patches, name prefixes/suffixes, common labels, and annotations.	Full templating system with conditionals, loops, and variable substitution.
Use-cases	■ Managing environment-specific customizations (e.g., dev, staging, prod) ■ Applying patches and modifications without duplicating YAML	■ Packaging and managing applications and their dependencies ■ Versioning of applications ■ More advanced customizations via templates and values files.

Kubernetes

Bases and Overlays

Bases and Overlays

Compose and customize different environments

- A **base** is a set of common Kubernetes resources (like deployments, services, ConfigMaps) that are shared across multiple environments. Bases are reusable and provide the foundation for your different environment configurations.
- An **overlay** is a set of environment-specific changes that are layered on top of the base. Through overlays, environments can apply their own customizations like changing replica counts, image tags, or enabling/disabling certain features.
  - When building the final configuration, Kustomize merges the overlay on top of the base, leaving the base intact while still being able to generate a customized configuration for the specific environment.

proj/base/kustomization.yaml

```yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
 - deployment.yaml
 - service.yaml
```

proj/overlays/prod/kustomization.yaml

```yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
 - ../../base

patches:
 - path: prod-replicas.yaml
```

```yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
 - ../../base

patches:
 - path: dev-replicas.yaml
```

proj/overlays/dev/kustomization.yaml

Kubernetes

Transformations

Transformations

Straightforward ways of customizing resources

Transformation examples		
Field	**Example**	**Effect**
`namespace`	`namespace: dev`	Assign resources to a specific Kubernetes namespace.
`namePrefix / nameSuffix`	`namePrefix: dev-` `nameSuffix: -01`	Prepend / append strings to the name of all resources in the included bases.

`commonLabels`	`commonLabels:`
	`  app: nginx`
	`  tier: frontend`
		Add the same set of labels to all resources and selectors.

`commonAnnotations`	`commonAnnotations:`
	`  project: ecommerce`
	`  team: finance`
		Add the same set of annotations to all resources.

`images`	`images:`
	`  - name: nginx`
	`  newName: newNginx`
	`  newTag: "1.27.0"`
		Modify the name, tags and/or digest for images without creating patches.

Kubernetes

Strategic Merge Patch and JSON Patch

Strategic Merge Patch and JSON Patch

Apply more complex customizations to your objects

- Kustomize offers two more advanced methods for us to apply configuration adjustments in overlays:
  - **Strategic Merge Patch:** YAML-based patching mechanism.
  - **JSON Patch:** Operations to apply to any JSON document.
- The **Strategic Merge Patch** method merges changes into resources based on the object schemas in Kubernetes. Since it understands the structure and semantics of Kubernetes API objects, we can provide partial configurations containing only the changes we wish to apply.
  - Strategic Merge Patch is schema aware: it recognizes special lists and maps in Kubernetes objects, and merges lists based on defined merge keys.
  - It's preferred for most common modifications and additions due to its simplicity and ease of use.
- The **JSON Patch** method specifies operations with precise JSON paths, and modifies the resource at the exact specified location.
  - Since it's operation-based, JSON Patches allow for high precision when targeting and editing specific fields.
  - It also allows removing and reordering list elements, as well as more complex operations when compared to the Strategic Merge Patch method.

Strategic Merge Patch and JSON Patch

Apply more complex customizations to your objects

When to use each?

Strategic Merge Patch

- Updates and additions are straightforward.
- Modifications target specific and known fields.
- Focus on readability and ease of use.

JSON Patch

- Performing complex or precise modifications not easily achievable with SMP (for example, removing containers from a Pod or replacing an element in a list).

Limitations

Strategic Merge Patch

- Not possible to easily remove items from lists
- There is the potential for conflicts if multiple patches are targeting the same fields with different modifications.
- Complex modifications are difficult to implement correctly.

JSON Patch

- More complex to define and harder to read.
- Since JSON Patches are not schema aware, then can lead to incorrect and potentially harmful/breaking modifications.
- Since JSON Patches identify elements by list indexes in the JSON path, changes in the order of the lists can also lead to incorrect configurations.

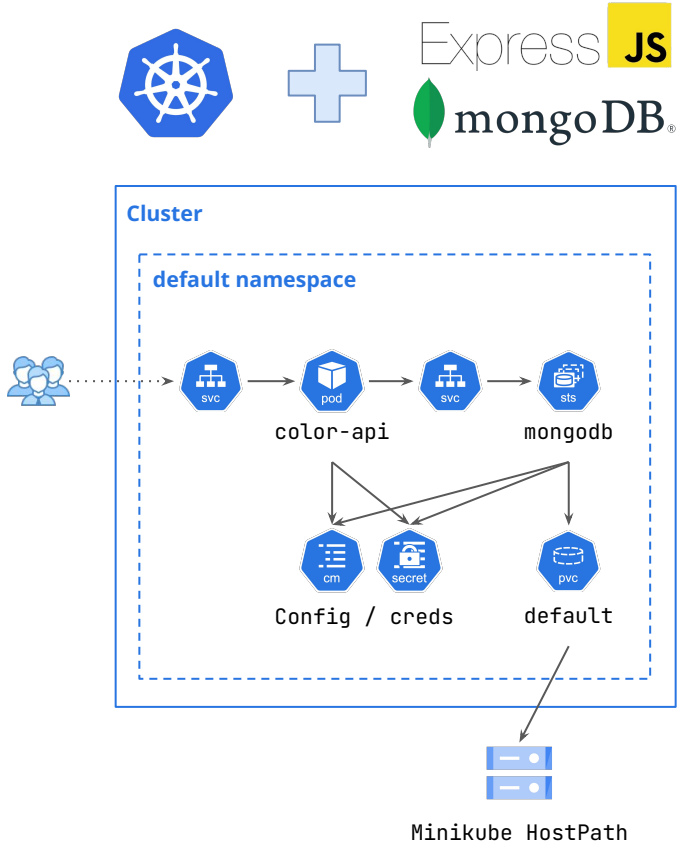
Kubernetes

Project - Persist Color API Data

Project - Persist Color API Data

Project overview

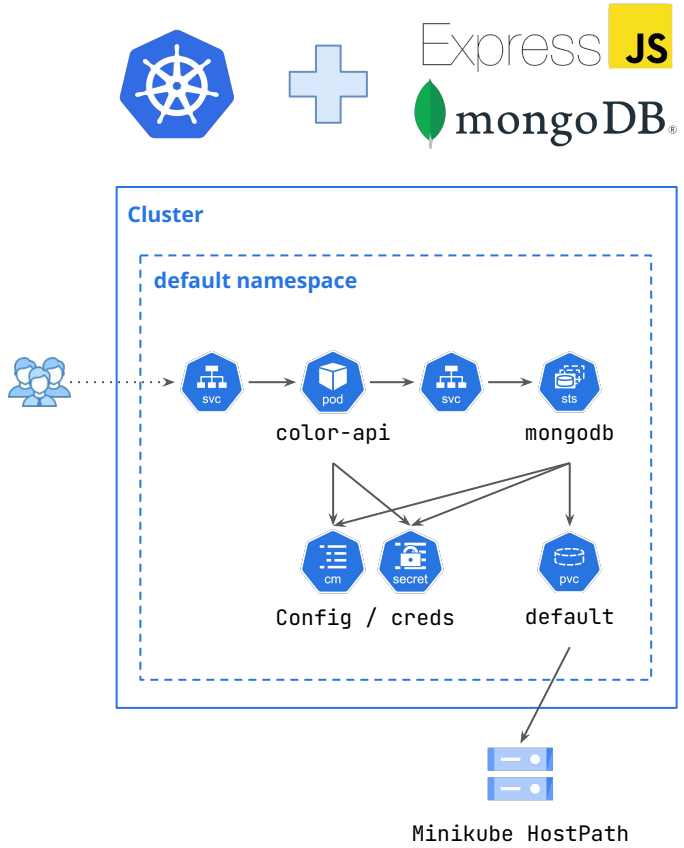
- **Project goal:** deploy a MongoDB stateful set and update our Color API to store the color information in the batabase.
- In order to achieve that, we will work through the following concepts:
  - Deploying a Stateful Set, Headless Service, and Persistent Volume Claim for dynamically creating our Persistent Volumes in Minikube.
  - Creating the necessary config maps and secrets for the database initialization and connection.
  - Update our Color API to receive the database connection information via environment variables, to be retrieved from the respective secrets.
  - Update our Color API to connect to MongoDB and use Mongoose for handling database interactions.
  - Add the relevant paths to our REST API.



Project - Persist Color API Data

Project overview

Color API REST API specification	
API method and path	**Expected behavior**
`GET /api`	■ Accepts a `colorKey` query URL parameter ■ Returns a message containing the color and the hostname
`GET /api/color`	■ Returns all stored colors
`GET /api/color/:key`	■ If a color with provided `key` does not exist, returns `404` ■ Returns the color information
`POST /api/color/:key`	■ If a color with provided `key` does not exist, saves the color in the DB ■ If a color with provided `key` exists, updates the color in the DB
`DELETE /api/color/:key`	■ Deletes a color by `key`



Kubernetes

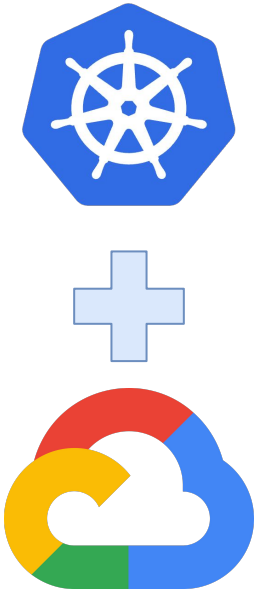
Kubernetes

Project - Deploy the Color API in GKE

Project - Deploy the Color API in GKE

Project overview

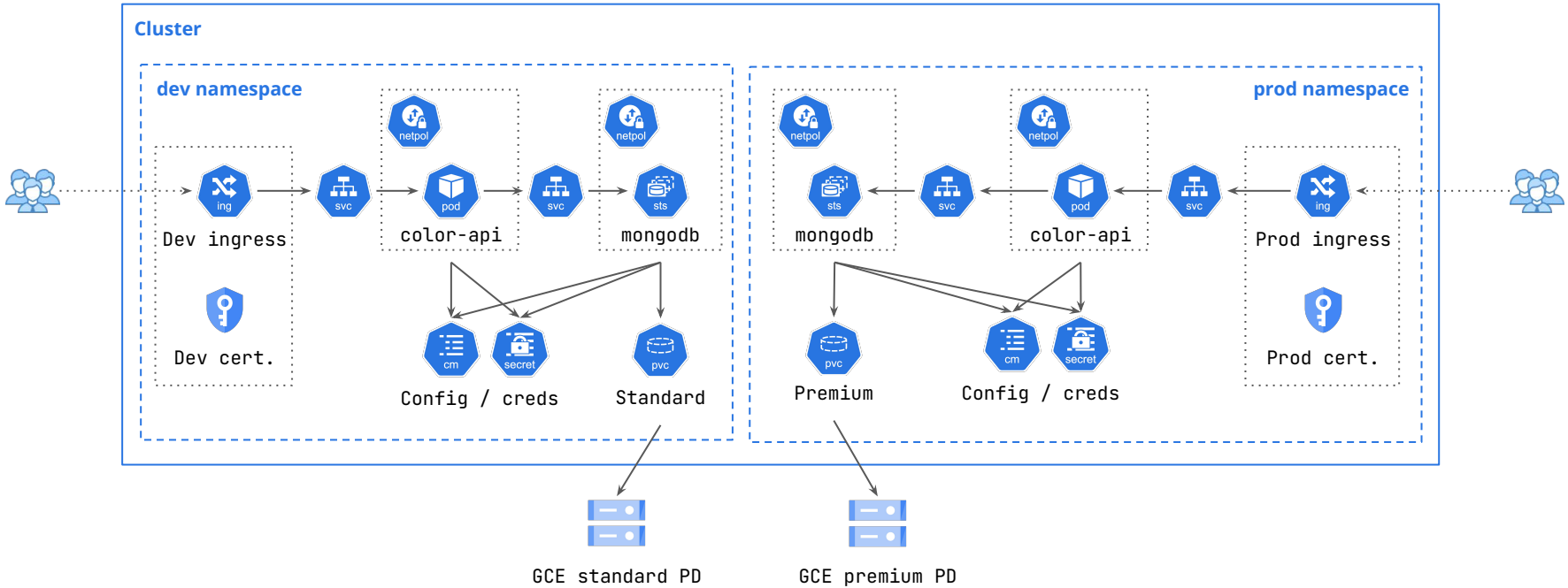
- **Project goal:** deploy two environments of our Color API application in a managed GKE cluster. For that, we will:
  - Bring everything we learned together in terms of Kubernetes to deploy an application in our GKE cluster: namespaces, deployments, stateful sets, persistent volumes, network policies, among others.
  - Explore how GKE's managed solutions provide many integrations with other services provided by Google Cloud, and how much easier our lives are with these integrations.
  - Explore how to leverage Kustomize's capabilities for configuring our Kubernetes objects.
- Versions to deploy:
  - Dev environment: `lmacademy/color-api:2.1.0-dev`
  - Prod environment: `lmacademy/color-api:2.0.0`
- We will also deploy ingresses, managed certificates, and network policies to ensure that our application is accessible from outside the cluster and that only necessary traffic is allowed between Pods.



Kubernetes

Project - Deploy the Color API in GKE

Project overview



Kubernetes

Introduction to GKE

Introduction to GKE

Exploring the managed K8s offering from GCP

- What is a managed Kubernetes offering?
  - **Managed services** normally handle the operational aspects, such as infrastructure management, maintenance, and updates.
  - In the context of Kubernetes, cloud providers manages the Kubernetes control plane and sometimes the worker nodes.
  - This gives the users focus on deploying and managing applications rather than the underlying Kubernetes system.
- Key features of GKE (Google Kubernetes Engine):
  - **Automatic Updates:** GKE automatically updates the control plane and nodes.
  - **High Availability:** Offers regional clusters for multi-zone resiliency.
  - **Scalability:** Supports seamless cluster autoscaling and node pools.
  - **Security:** Leverages Google Cloud IAM and provides network security features.
  - **Integration:** Seamless integration with other cloud services (e.g., storage, networking, IAM).
- GKE deployment modes:
  - **Standard Mode:** Users manage node pools with control over node configurations.
  - **Autopilot Mode:** Google manages the nodes, and users focus solely on workloads.