Day4 of Internship

- **Introduction**

This documentation covers key topics in modern software deployment, including **Docker**, **Kubernetes**, **Monolithic vs. Microservices Architecture**, and **Blue-Green Deployment**. It explains how containerization, orchestration, and deployment strategies improve scalability, reliability, and continuous delivery in today's software systems.

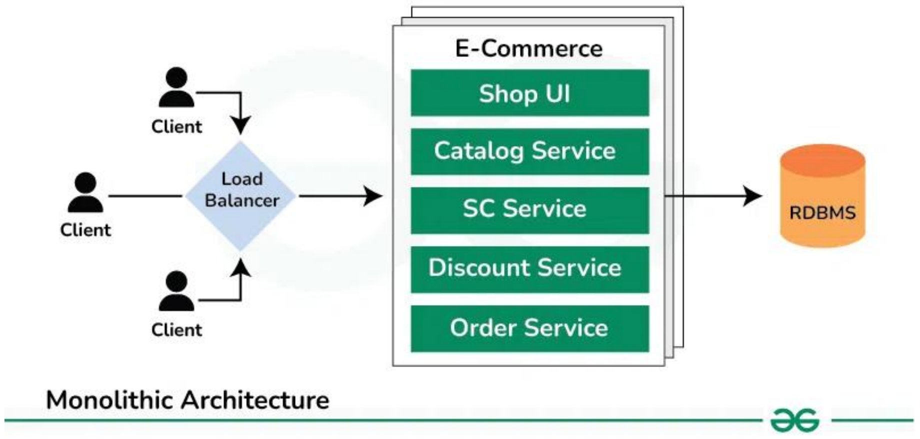
- **Research Topics**

Monolithic vs Microservices architecture

Monolithic Architecture

Software is traditionally designed using a monolithic architecture, in which the entire program is constructed as a single, indivisible unit.

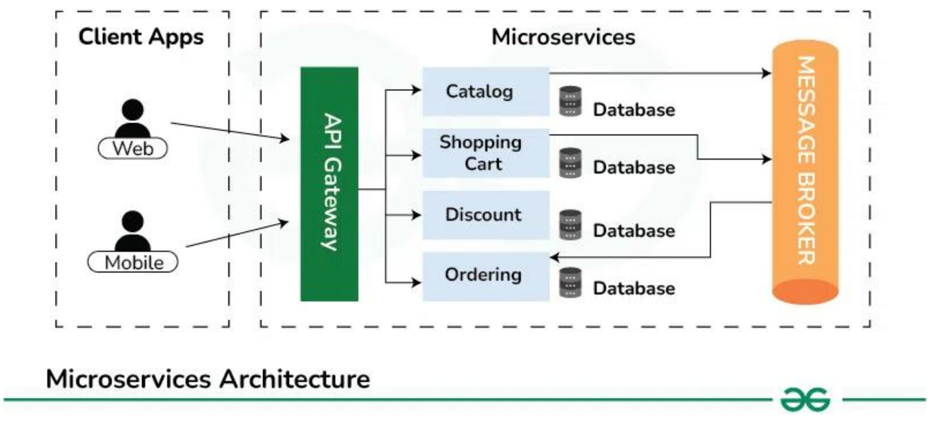
- This means that any changes or updates to the application require modifying and redeploying the entire monolith.
- Monolithic architectures are often characterized by their simplicity and ease of development, especially for small to medium-sized applications.
- However, they can become complex and difficult to maintain as the size and complexity of the application grow.



Microservices Architecture

A microservices architecture results in an application designed as a set of small, independent services. Each one represents a business capability in itself. The services loosely couple with one another and communicate over the network, typically making use of lightweight protocols such as HTTP or messaging queues.

- Each service is responsible for a single functionality or feature of the application and can be developed, deployed, and scaled independently.
- The Microservice architecture has a significant impact on the relationship between the application and the database.



Differences between Monolithic and Microservices Architecture

Below are the differences the Monolithic and Microservice architecture:

Aspect	Monolithic Architecture	Microservice Architecture
Architecture	Single-tier architecture	Multi-tier architecture
Size	Large, all components tightly coupled	Small, loosely coupled components
Deployment	Deployed as a single unit	Individual services can be deployed independently
Scalability	Horizontal scaling can be challenging	Easier to scale horizontally

Aspect	Monolithic Architecture	Microservice Architecture
Development	Development is simpler initially	Complex due to managing multiple services
Technology	Limited technology choices	Freedom to choose the best technology for each service
Fault Tolerance	Entire application may fail if a part fails	Individual services can fail without affecting others
Maintenance	Easier to maintain due to its simplicity	Requires more effort to manage multiple services
Flexibility	Less flexible as all components are tightly coupled	More flexible as components can be developed, deployed, and scaled independently
Communication	Communication between components is faster	Communication may be slower due to network calls

Source: https://www.geeksforgeeks.org/software-engineering/monolithic-vs-microservices-architecture/

Blue-Green deployment environment

A blue/green deployment is a software release management strategy that minimizes downtime and reduces risk by running 2 identical production environments, referred to as "blue" (production) and "green" (new version). Only one of these environments serves live production traffic at any time, while the other remains idle. Blue/green deployments let you deploy updates and new features with minimal disruption and maximum reliability.

In a typical blue/green deployment setup, the blue environment is the active environment serving users. When a new application version is ready, you deploy it to the green environment, which mirrors the blue environment but doesn't handle live traffic yet.

Benefits of blue/green deployments

Blue/green deployments reduce the risk of downtime during deployments. They're also a powerful way to use hardware that was traditionally used for staging environments. A blue/green setup serves 3 purposes:

- Staging: When blue is active, green becomes the staging environment for the next deployment.
- Rollback: After deploying to blue and making it active, you discover a problem. Since green still runs the old code, you can roll back easily.
- Disaster recovery: After deploying to blue and being satisfied that it's stable, you can deploy the new release to green, too. This gives you a standby environment ready in case of disaster.

Blue/green deployments are especially useful for:

- Zero-downtime releases: Organizations that require high availability can use blue/green deployment to release new features without downtime.
- Performance benchmarking: Organizations can use the idle environment to run performance tests on the new version without affecting the live environment. This helps fine-tune and optimize the application before it goes live.
- Compliance and audits: In regulated industries, maintaining a robust deployment strategy is essential. Blue/green deployments ensure that changes are transparent and reversible, smoothing compliance audits.

The blue/green deployment process involves several steps:

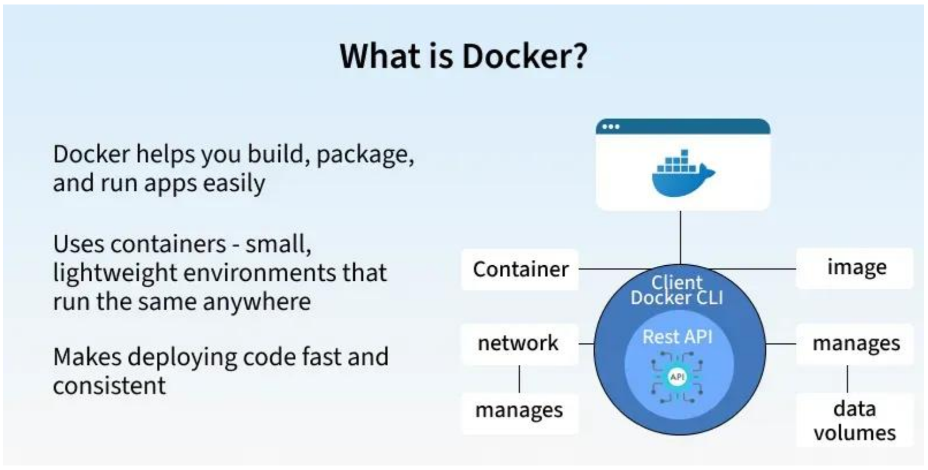
1. Prepare the new release: Develop and test the new version of the application in a staging environment.
2. Deploy to the green environment: Deploy the new application version to the green environment, which is a clone of the blue environment but not yet live.
3. Testing: You can test the new version on the green environment to ensure it functions as expected without affecting the user experience.
4. Switch traffic: Redirect user traffic from the blue environment to the green environment. You can do this using a load balancer or DNS switch.
5. Monitor: Closely monitor the green environment for any issues or anomalies after it goes live.
6. Fallback plan: If there are any critical issues, you can quickly switch traffic back to the blue environment, ensuring minimal disruption.

7. Cleanup: After the green environment is stable and running without issues, you can update the blue environment with the same changes, keep it as a backup for the next deployment cycle, or phase it out to reduce costs.

Source: https://octopus.com/devops/software-deployments/blue-green-deployment/

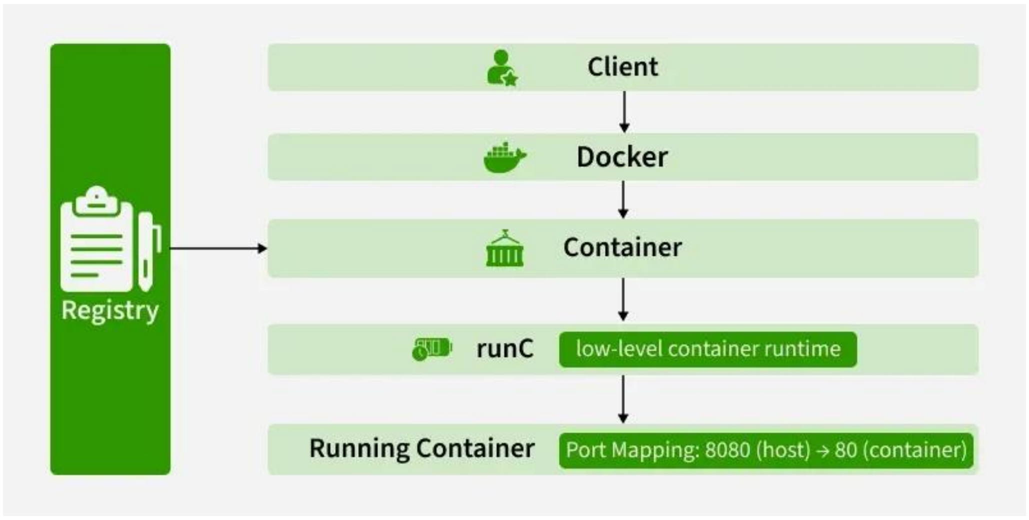
Docker

Docker is a tool that simplifies the process of developing, packaging, and deploying applications. By using containers, Docker allows you to create lightweight, self-contained environments that run consistently on any system, minimising the time between writing code and deploying it into production.


What is Docker?
Docker helps you build, package, and run apps easily
Uses containers - small, lightweight environments that run the same anywhere
Makes deploying code fast and consistent
Container
image
Client Docker CLI
Rest API
API
network
manages
manages
data volumes

Architecture of Docker

Docker follows a client-server architecture. The Docker client communicates with a background process, the Docker Daemon, which does the heavy lifting of building, running, and managing your containers. This communication happens over a REST API, typically via UNIX sockets on Linux (e.g., /var/run/docker.sock) or a network interface for remote management.


Registry
Client
Docker
Container
runC
low-level container runtime
Running Container
Port Mapping: 8080 (host) → 80 (container)

The Core Architectural Model

- Docker Client: This is your command center. When you type commands like docker run or docker build, you're using the Docker Client.
- Docker Host: This is the machine where the magic happens. It runs the Docker Daemon (dockerd) and provides the environment to execute and run containers.
- Docker Registry: This is a remote repository for storing and distributing your Docker images.

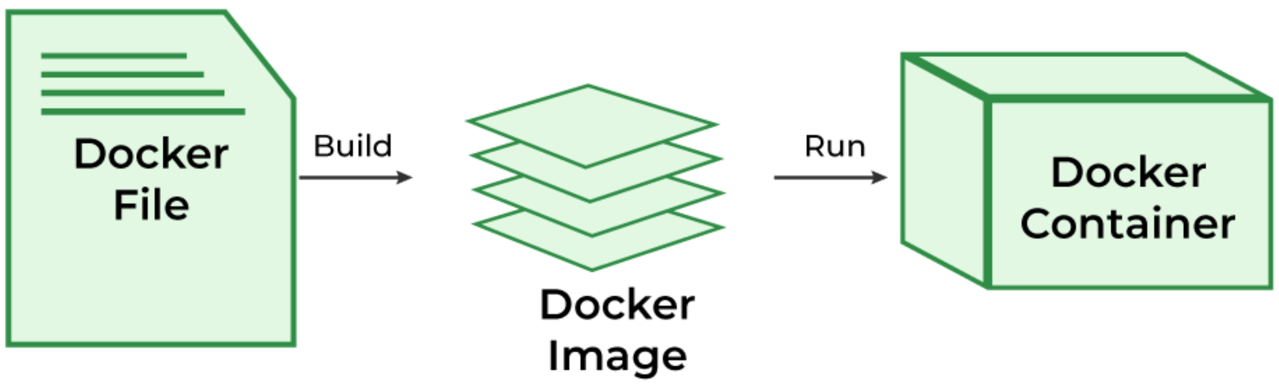
This interaction forms a simple yet powerful loop: you use the Client to issue commands to the Daemon on the Host, which can pull images from a Registry to run as containers.

Docker file

A Docker file is a simple text file that contains a script of instructions for building a Docker image. Think of it as a recipe for creating your application's environment. The Docker engine reads this file and executes the commands in order, layer by layer, to assemble a final, runnable image.

Docker Image

A Docker Image is a lightweight, standalone, and executable software package that includes everything needed to run an application: the code, a runtime, system tools, libraries, and settings. The docker image includes the following to run a piece of software. A docker image is a platform-independent image that can be built in the Windows environment and it can be pushed to the docker hub and pulled by others with different OS environments like Linux.



Docker Hub

Docker Hub is Docker’s official cloud-based registry where developers store, share, and download Docker images. It acts like a central library of pre-built images — for example, you can pull images of databases (MySQL, MongoDB), programming languages (Python, Node.js), or even full applications.

Docker Commands

Docker originally used straightforward, individual commands for each task. Over time, as more features were added, this approach became harder to manage. To make the interface clearer and more structured, Docker introduced an object-based command system, grouping commands by the type of resource they control.

This makes the CLI more organized and easier to understand, and while the old commands remain for compatibility, the newer format is recommended for ongoing use.

Old Command	New (Recommended) Command	Purpose
docker ps	docker container ls	List containers
docker run	docker container run	Run a command in a new container
docker rm	docker container rm	Remove one or more containers
docker images	docker image ls	List images
docker rmi	docker image rm	Remove one or more images

Docker Container Stop

This command allows you to stop a container if it has crashed or you want to switch to another one.

```
$ docker container stop <container_ID>
```

Docker Container Start

Suppose you want to start the stopped container again, you can do it with the help of this command.

```
$ docker container start <container_ID>
```

Docker rm

Removes one or more **stopped** containers. You cannot remove a container that is still running; you must stop it first. You can use the docker stop <container_name or ID> command to stop the container.

```
$ docker rm {options} <container_name or ID>
```

Kubernetes

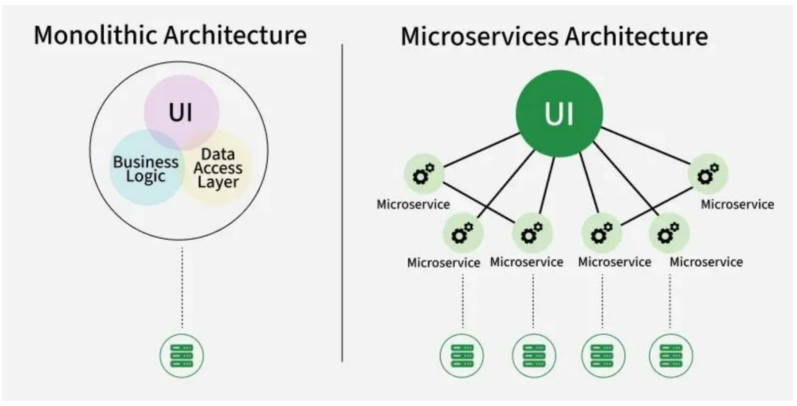
Kubernetes**,** often shortened to K8s (K, 8 letters, s), is an open-source platform that automates the deployment, scaling**,** and management of containerized applications.

- **Origin**: Developed by Google, inspired by internal systems Borg and Omega.
- **Launch**: Officially released in 2014.
- **CNCF Donation**: Donated to the Cloud Native Computing Foundation (CNCF) in 2015, which now maintains it.
- **Adoption**: Widely used across major cloud providers today.
- **Name Meaning**: *Kubernetes* comes from Greek, meaning **"**helmsman" or "pilot**"**, symbolizing its role in steering applications.

Think of Kubernetes as an orchestra conductor. Each container is a musician. While you can manage a few musicians yourself, you need a conductor to coordinate an entire orchestra to play a complex symphony. You simply give the conductor the sheet music (your desired configuration), and they ensure every musician plays their part correctly, replacing someone who falls ill and bringing in more players.

Monolithic Vs Microservices

In the past, applications were built using a monolithic architecture, where everything was interconnected and bundled into one big codebase. This made updates risky for example, if you wanted to change just the payment module in an e-commerce app, you had to redeploy the entire application. A small bug could crash the whole system.To overcome this, the industry moved toward microservices**,** where each feature (like payments, search, or notifications) is built and deployed independently. This made applications more flexible and scalable.But with **microservices** came a new challenge:instead of running one big app, companies now had to manage hundreds or thousands of small containerized services. Containers solved the packaging problem, but without a way to orchestrate them, things got messy. That's where Kubernetes came in acting like a smart manager that automates deployment, scaling, and coordination of all those microservices.



Terminologies in K8s

Think of Kubernetes as a well-organized company where different teams and systems work together to run applications efficiently. Here's how the key terms fit into this system:

1. Pod

A Pod is the smallest unit you can deploy in Kubernetes. It wraps one or more containers that need to run together, sharing the same network and storage. Containers inside a Pod can easily communicate and work as a single unit.

2. Node

A Node is a machine (physical or virtual) in a Kubernetes cluster that runs your applications. Each Node contains the tools needed to run Pods, including the container runtime (like Docker), the Kubelet (agent), and the Kube proxy (networking).

3. Cluster

A Kubernetes cluster is a group of computers (called nodes) that work together to run your containerized applications. These nodes can be real machines or virtual ones.

There are two types of nodes in a Kubernetes cluster:

1. **Master node (Control Plane):**
   - Think of it as the brain of the cluster.
   - It makes decisions, like where to run applications, handles scheduling, and keeps track of everything.
2. **Worker nodes:**
   - These are the machines that actually run your apps inside containers.
   - Each worker node has a Kubelet (agent), a container runtime (like Docker or containerd), and tools for networking and monitoring.

4. Deployment

A Deployment is a Kubernetes object used to manage a set of Pods running your containerized applications. It provides declarative updates, meaning you tell Kubernetes what you want, and it figures out how to get there.

5. ReplicaSet

A ReplicaSet ensures that the right number of identical Pods are running.

6. Service

A Service in Kubernetes is a way to connect applications running inside your cluster. It gives your Pods a stable way to communicate, even if the Pods themselves keep changing.

7. Ingress

Ingress is a way to manage external access to your services in a Kubernetes cluster. It provides HTTP and HTTPS routing to your services, acting as a reverse proxy.

8. ConfigMap

A ConfigMap stores configuration settings separately from the application, so changes can be made without modifying the actual code.

Imagine you have an application that needs some settings, like a database password or an API key. Instead of hardcoding these settings into your app, you store them in a ConfigMap. Your application can then read these settings from the ConfigMap at runtime, which makes it easy to update the settings without changing the app code.

9. Secret

A Secret is a way to store sensitive information (like passwords, API keys, or tokens) securely in a Kubernetes cluster.

10. Persistent Volume (PV)

A Persistent Volume (PV) in Kubernetes is a piece of storage in the cluster that you can use to store data and it doesn’t get deleted when a Pod is removed or restarted.

11. Kubelet

A Kubelet runs on each Worker Node and ensures Pods are running as expected.

12. Kube-proxy

Kube-proxy manages networking inside the cluster, ensuring different Pods can communicate.

Source: https://www.geeksforgeeks.org/devops/introduction-to-kubernetes-k8s/

Yaml

The YAML is an open-source language used for writing configuration files on your system, what makes the YAML language different from other computer languages is that it is human-readable, unlike other programming languages which makes it easier to work with.

The full form of the YAML language can be different depending on whom you are asking it, mostly there are two full forms of this language and both of them are considered to be correct, the first one is yet another markup language while the other full form is YAML ain't markup language. if you read the second full form of YAML, you will understand that the YAML is not a markup language which means that this language is used for data and not documents.

YAML syntax:

The syntax of the YAML programming language is very easy to understand and time efficient because it focuses on generating human-readable codes, here are some of the syntax rules for YAML:

Indentation: Just like python, the YAML language also heavily works on the basis of indentation in order to represent the nesting of the data as well as the hierarchy of it.

Note: In YAML, spaces are used for indentation rather than tabs.

Lists: YAML also allows us to create lists if we want, in YAML the lists are based on arrays which are a sequence of items. In YAML, each item which begins with a dash ('-') along with a space is known as a list.

For example:

```
“`yaml
hobbies:
Catering
Cooking
Coding
“`
```

Key-Value pair: This contains data which is stored in key value pairs and are separated by a colon.

Example:

"name:Kishan Kaushik"

File Extension: YAML files are saved with a file format that ends with either ".yaml" or ".yml"

YAML Scalar syntax:

YAML scalar means that a simple value for any key, as we discussed later that the scalar can be float, Boolean, integer etc..The scalars are known to be the basic blocks that are required for writing the YAML language and they can be complex as well as simple depending on the requirements. The scalars can be written in various possible ways, let's look at some of them as example:

1. Plain Style:

In the plain style of scalars in YAML, the scalars can be some simple strings which does not include any special characters or quotes, for example you can see:

```
key: value
```

2. Single-Quoted Style:

The scalars can also be closed in the single quotes this style is mostly used whenever you want to specify a string just the way it is without any special characters, for example:

```
key: 'value'
```

3. Double-Quoted Style:

Just like the single quotation marks we can also use the double quotations for the strings the only difference here is that the double-quoted string allows escape sequence to be interpreted, for example:

```
key: "value with\nnewline"
```

4. Flow Style:

Scalars can also be written in a more compact flow of style. this is suited for simple values and short values, for example:

```
key: { foo: bar, baz: 42 }
```