DevOps

Docker Container

Monolithic vs Microservices

Monolithic Architecture

- Entire application built as **one single unit**.
- Components like **UI, Business Logic, Data Access Layer** are tightly coupled.
- Single deployment package (e.g., WAR/JAR file).
- Scaling requires **scaling the whole application**, even if only one part needs more resources.
- High risk: **failure in one module can crash the entire system**.

Microservices Architecture

- Application broken into **independent, loosely coupled services**.
- Each service handles a specific functionality (e.g., payments, catalog, authentication).
- Services communicate typically via APIs (REST/gRPC).
- Allows **scaling of individual services** based on demand.
- Failure in one service does not necessarily impact others.

Key Differences (Monolith vs Microservices)

Scaling

- Monolithic: Must scale the whole app.
- Microservices: Can scale **individual services** as needed.

Deployment

- Monolithic: Single, large unit → **risky, slow, redeployment of whole app for changes**.
- Microservices: Independent services → **fast, safe, rolling deployments possible**.

Technology Stack

- Monolithic: **Homogeneous** (one language/stack).
- Microservices: **Heterogeneous** (different languages/frameworks for different services).

Team Structure

- Monolithic: Large, **cross-functional teams** maintaining one codebase.
- Microservices: **Small, autonomous teams** owning individual services.

Fault Isolation

- Monolithic: **Single point of failure** can crash entire app.
- Microservices: Failure is **isolated to one service**, system remains mostly functional.

Real-World Example 1 (Scalability Challenge)

- In the beginning, your application runs on a single server, and this setup is enough to serve around one hundred users without any problems. As the website becomes popular, the user base grows to one million, with nearly ten thousand users active at the same time.
- Even though the server has powerful hardware — for example, sixty-four gigabytes of RAM and ten terabytes of disk space — it can only handle around eight hundred to one thousand users smoothly. Beyond that point, the server becomes overloaded.
- At this stage, there are two ways to scale. One option is **vertical scaling**, where you add more RAM or CPU to the same machine. However, this quickly becomes expensive and has physical limits. The other option is **horizontal scaling**, where you add more servers. But with a monolithic application, every new server must run a full copy of the entire application, even if only one part, such as the payment service, is causing the overload.
- The key problem is that a monolith forces you to scale the whole application instead of just the part that needs extra capacity. This wastes resources and makes scaling inefficient.

Challenge #2: Frequent Updates Without Full Redeployment

In a monolithic system, even a very small change, such as updating the payment service, requires redeploying the entire application. This process increases downtime, slows down development, and carries the risk of breaking unrelated parts of the application.

In contrast, a microservices-based system allows each component, such as the payment service or the authentication service, to be updated independently. Using modern CI/CD pipelines, these updates can be deployed quickly and safely without disturbing the rest of the application. This makes deployment cycles faster, reduces risk, and enables continuous delivery.

Monolith vs Microservice for Payments

1. Scalability

In a monolithic application, if the payment service is under heavy load, you must scale the entire application. This means even unrelated parts, such as product search or user profiles, get extra resources they don't need, which wastes money and computing power.
With microservices, the payment service can be scaled independently. For example, during a holiday sale, only the payment service might need to handle ten times more traffic while other services remain steady. However, this comes with a challenge: setting up advanced **load balancing** and **auto-scaling** rules is more complex than in a simple monolithic setup.

2. Resilience and Fault Isolation

In monoliths, a payment failure could bring down the entire application because everything runs in one process. Imagine the payment gateway going offline — now even browsing products may stop working.
With microservices, failures are isolated. If the payment service fails, users can still browse products or add items to the cart, and payment can be retried later. Developers can also add safety mechanisms such as circuit breakers, which stop one failure from triggering a chain reaction. On the flip side, retry handling and error recovery in distributed systems must be carefully designed.

3. Latency and Network Overhead

Monoliths run inside a single process, so internal communication is usually faster. In microservices, each service communicates over the network, which can introduce some delay. For example, the payment service may need to talk to external gateways like Stripe or PayPal, and that adds latency.
To reduce this overhead, developers often use efficient protocols like gRPC or asynchronous message queues such as Kafka and RabbitMQ. Still, the unavoidable reality is that more network calls increase the risk of slowdowns compared to the all-in-one nature of monoliths.

4. Technology Flexibility

Monoliths typically lock you into one tech stack — one language, one database. This makes things consistent but limits choices. For example, if the entire e-commerce system is written in Java with a MySQL database, you must use the same for payments, even if another tool might be better.
Microservices give you the freedom to choose the best technology for each job. For payments, you might use PostgreSQL for financial transactions and Redis for caching recent payments. You might even write the payment service in Go for speed while keeping the catalog service in Python. The trade-off is that this "polyglot" setup increases complexity, because your team must maintain many different technologies at once.

5. Development and Deployment Speed

In a monolith, even a small change in the payment logic requires rebuilding and redeploying the entire application. This slows down delivery and creates risk — a change in payments could accidentally break product search.

In microservices, the payment service can be updated independently. For example, if you want to add a new payment provider like Apple Pay, you can deploy only the payment service using CI/CD pipelines, without touching the rest of the system. This makes releases faster and safer. The downside is that managing many independent deployments increases DevOps workload, since orchestration and coordination become critical.

Virtualisation

Read from previous Slides

Cloud-Native Apps – Characteristics

Cloud-native applications are built with **microservices architecture** rather than monolithic design. This means different parts of the application can be developed, scaled, and deployed independently. To make this work, cloud-native apps usually run inside **containers** such as Docker or Podman, which ensure the application behaves the same on any environment, whether on a developer's laptop or a production cloud server.

Managing many containers requires **orchestration platforms** like Kubernetes, AWS ECS, or Google Cloud Run. These tools handle container scheduling, scaling, and failover automatically. In addition, **serverless computing** (e.g., AWS Lambda or Google Cloud Functions) allows developers to focus purely on writing code while the cloud provider handles servers behind the scenes.

Cloud-native apps also rely heavily on **auto scaling and automation**. For example, an online ticket-booking service can automatically spin up extra instances during a concert ticket release and scale back down when demand drops. Finally, **CI/CD pipelines** enable teams to release new features rapidly without waiting for long release cycles.

Cloud-Native Apps – Obstacles

While powerful, cloud-native development comes with challenges. **Complexity** is a major issue: splitting an application into dozens of microservices requires careful design and ongoing management. For example, keeping track of dependencies between services like "orders," "payments," and "notifications" can get overwhelming.

There is also a **cultural shift**. Traditional organisations used to slow, centralised IT processes need to embrace DevOps culture, where teams take more responsibility for deployment and operations. **Legacy systems** are another obstacle. For instance, a bank running decades-old mainframes may find it extremely difficult to migrate smoothly into a microservices-based, containerised setup. Finally, **security** is more complicated in a distributed environment. Instead of protecting one big application, you now have to secure dozens of services communicating over networks.

Cloud-Native Apps – Enablers

Several enablers make cloud-native adoption possible. **Automation tools**, such as CI/CD pipelines and Infrastructure as Code (Terraform, Ansible), allow teams to manage infrastructure and releases at scale. **Container orchestration** through platforms like Kubernetes ensures that hundreds of containers can be scheduled, scaled, and updated automatically without manual intervention.

A strong **DevOps culture** is also critical, encouraging collaboration between developers and operations staff. For example, developers don't just hand over code to operations — they take part in monitoring and improving it in production. Finally, **cloud services** such as managed databases, serverless platforms, and monitoring tools reduce the operational burden so that teams can focus more on business features than on infrastructure.

CNCF Landscape

The **Cloud Native Computing Foundation (CNCF)** plays a central role in shaping the ecosystem. It hosts open-source projects that have become industry standards. For instance, **Kubernetes** is the leading platform for orchestrating containers. **Prometheus** provides real-time monitoring and alerting, which is essential for detecting failures early. **Envoy** is a service proxy used to manage secure communication between microservices. Finally, **Fluentd** helps collect and unify logs across many services, making it easier to troubleshoot issues in distributed systems.

Cloud DevOps

Cloud DevOps combines cloud infrastructure with DevOps practices to streamline how software is built and maintained. **CI/CD pipelines** automate the integration and deployment of code so that teams can deliver new features or bug fixes multiple times per day. **Infrastructure as Code (IaC)** tools like Terraform or Ansible allow infrastructure to be described in files and versioned like code, which reduces human error and makes environments reproducible. **Monitoring and logging** tools such as Prometheus or the ELK stack (Elasticsearch, Logstash, Kibana) give teams real-time visibility, so they can quickly detect and fix performance issues.

Microservices in Cloud-Native Ecosystem

Microservices are the backbone of cloud-native apps. Each service is **independent** — meaning it can be built, tested, and scaled separately. For example, an e-commerce platform may have separate services for user authentication, product catalog, and payments.

These services communicate using lightweight protocols like HTTP/REST or through messaging queues like Kafka. This allows them to remain loosely coupled while still working together. Microservices also bring **flexibility**, making it easier to update or replace one component without affecting the entire application. For example, a payment service can switch from PayPal to Stripe without changing the product catalog or the shopping cart.

Containers in Cloud-Native Ecosystem

Containers are **lightweight, portable units** that package an application with all its dependencies. This ensures that the application runs the same on a developer’s laptop, in a staging environment, and in production. Containers provide **isolation**, so even if multiple services run on the same host, they don’t interfere with each other.

They are also **efficient**, consuming fewer resources compared to full virtual machines. For instance, you can run hundreds of small containers on a single server, whereas the same server might only run a few VMs. Finally, containers are highly **portable**, meaning a service built and tested on Docker locally can run seamlessly on AWS, Google Cloud, or even an on-premise data centre.

Why Do We Need Containers?

Traditionally, applications were run on **virtual machines (VMs)**. A VM includes not only the application and its libraries but also a full copy of the operating system. This makes VMs **heavyweight**. For example, an Ubuntu VM image can be around **4 GB**, and booting it up takes a lot of time before you can even start the application. Imagine needing to spin up multiple such VMs quickly — it's like waiting for several laptops to start up just to run a single app.

This inefficiency raises the question: *can we do better?* That's where containers come in. Containers are lightweight and don't need a full OS inside each unit. They share the host operating system, which makes them faster to start and smaller in size.

Hypervisors vs Containers

There are two main types of hypervisors for VMs:

1. **Type 1 Hypervisors (Bare Metal)**
   These run directly on the hardware. The hypervisor itself manages the hardware and creates VMs. Each VM still needs its own operating system. For example, a cloud provider like VMware ESXi or Microsoft Hyper-V uses this model.

2. **Type 2 Hypervisors (Hosted)**
   These run on top of an existing operating system. The hypervisor is essentially another software layer. VMs still carry their own OS. This setup is common in developer laptops using VirtualBox or VMware Workstation.

In both cases, every VM carries an entire OS image, making it **bulky and slow**.

Where Containers Differ

Containers solve these problems by running on a **container runtime** (like **Docker** or **containerd**) on top of the host OS. Instead of carrying a full OS, each container only packages the **application and its dependencies**. This makes containers:

- **Lightweight**: Instead of gigabytes, container images can be a few hundred MBs or less.
- **Fast to Start**: Containers can start in seconds, compared to minutes for VMs.
- **Efficient**: Multiple containers can run on the same OS, sharing its kernel.

For example, if you run two applications — one for managing orders and another for processing payments — using virtual machines would mean allocating separate operating systems for both, wasting storage and memory. With containers, both applications can run on the same host operating system, each in its own isolated environment, consuming fewer resources and starting quickly.

Example

Suppose you're running an e-commerce platform with two main apps: a **catalog service** and a **payment service**.

- In the VM world, each app would run inside its own VM, each carrying a full operating system. Starting them up could take minutes, and you'd waste resources maintaining duplicate OSes.

- In the container world, you just package each service with its required libraries and run both on the same OS using Docker. They'll start within seconds, consume fewer resources, and scale up or down much faster during high traffic (like Black Friday sales).

Base Images in Docker: User-Space Without the Kernel

When you run Docker, the container does **not** need a full operating system kernel like a virtual machine. Instead, it shares the host's kernel. That's why containers are lightweight compared to VMs.

However, many Docker images (like `ubuntu`, `debian`, or `alpine`) are called **base images**. These contain just enough of the **user-space tools and libraries** from that distribution to run applications. They don't include the Linux kernel, because the kernel is already provided by the host operating system.

So, when you see Docker pulling an `ubuntu` image:

- It's not downloading a full 4 GB Ubuntu VM with its kernel.
- It's downloading a stripped-down user-space environment (libraries, shell, package manager, etc.).
- The size is usually a few hundred MB (sometimes less if it's minimized).

This way, developers can build apps in a familiar environment (say Ubuntu), while still benefiting from container efficiency.

The Problem with Developer Onboarding

When a new developer joins a team working on a data analytics product, they need to set up the entire technology stack before they can contribute. This usually means installing multiple tools and dependencies, which vary depending on the operating system. For instance, if the team uses Oracle Data Integrator as an ETL tool, the developer first needs to install the Java Development Kit (JDK) and also configure Redis for caching.

The installation steps differ between Windows and Linux, and the process often involves many tools and manual configuration. This not only takes time but is also error-prone, especially for someone new to the team. As a result, valuable onboarding time is lost, and the developer may face frustrating setup issues before even starting real work.

How Docker Simplifies Onboarding

With Docker, the entire setup can be packaged into a single custom image that contains all the required tools, libraries, and configurations. Instead of spending hours or days installing everything manually, the new developer only needs to install Docker, pull the pre-built image, and run it.

This drastically reduces onboarding time and ensures consistency across environments. Whether it’s a development machine, a QA test environment, or production, the same Docker image guarantees that the application runs in a predictable way. This eliminates the “works on my machine” problem and accelerates productivity for the whole team.

Docker Architecture

Docker works on a simple but powerful architecture. On your machine, you have the **Docker client**, which is what you interact with when you type commands like `docker build`, `docker pull`, or `docker run`. These commands are sent to the **Docker daemon** (the background service), which does the heavy lifting — building images, running containers, and managing resources.

The daemon stores **images**, which are blueprints containing your application code plus its dependencies. From these images, Docker launches **containers**, which are lightweight, isolated environments where your applications actually run.

When you need an image (say Ubuntu, Redis, or Nginx), Docker can **pull it from a registry** (like Docker Hub). This is like an app store for containers. Once pulled, you can reuse that image across many containers.

Advantages of Using Docker

The true power of Docker is in the benefits it gives developers and teams:

- **Accelerated Developer Onboarding**: Instead of spending hours installing tools and dependencies, a new developer can just run `docker run …` and instantly get a ready-made environment. This saves days of setup frustration.
- **Eliminate App Conflicts**: Different apps might need different versions of Java, Python, or databases. On a normal machine, this causes conflicts. Docker isolates each app in its own container, so multiple versions can run side by side without clashing.
- **Environment Consistency**: One of the biggest developer pains is “it works on my machine but not in production.” With Docker, the same container image runs the same way in development, testing, and production — removing these inconsistencies.
- **Ship Software Faster**: Since everything is packaged into containers, deployment is as simple as shipping the image. This speeds up delivery cycles and makes continuous integration/continuous deployment (CI/CD) pipelines smoother.

Docker vs VMs wrt Compatibility

- VMs can run any OS image on any host (Windows host running Ubuntu VM).
- Docker requires kernel support (Linux containers need Linux kernel).
- Old Windows versions (7/8) cannot run Linux containers.
- Newer Windows versions (10/11 with WSL2) support Linux containers natively.

Docker Basics

When working with Docker, you’ll often hear about **images** and **containers**. These two concepts are closely related, but they serve different purposes.

Docker Image – The Blueprint

Think of a **Docker image** as a *blueprint* or a *recipe*.

- Just like a house blueprint specifies the structure, materials, and layout before construction, a Docker image defines everything needed to run an application.
- This includes:
  - The **operating system environment** (like Ubuntu, Alpine Linux, etc.)
  - The **libraries and dependencies** (for example, Python runtime, database drivers, or system utilities)
  - The **application code** itself

The key thing is: an image is **static**—it does not change when it’s just sitting there. It’s more like a saved template.

Docker Basics

Docker Container – The Live Instance

Now imagine you take that blueprint and actually put into action.

- A container is a **running instance of an image**.
- When you “start” a container, Docker takes the image, sets up the environment it defines, and launches your application inside it.
- Unlike the static image, the container is **dynamic**: you can interact with it, run commands inside it, and even modify it while it’s running.

Getting Started with Docker

```
docker -v
# If not found → install:
sudo snap install docker   # or: sudo apt install docker.io
docker -v
# Docker version 28.1.1+1
docker images
# empty at first
docker pull ubuntu
# permission denied → use sudo:
sudo docker pull ubuntu
docker images
# now shows ubuntu:latest
sudo docker run --name ubuntu1 ubuntu
docker ps
# permission denied → use sudo:
sudo docker ps
sudo docker ps -a
# shows all containers (running + stopped)
```

Comment: Containers stop when no foreground process runs. Unlike VMs, containers don't "run an OS," they run a process.

```
docker pull ubuntu           # download latest Ubuntu
docker pull ubuntu:jammy      # download a specific tagged version
docker pull mysql            # download MySQL server image
```

Passing Environment Variable
We can also pass environment variables when starting a container, which is especially useful for services like databases:

```
docker run --name mysql2 -d -e MYSQL_ROOT_PASSWORD=mysql123 mysql
```

Port Binding

Here, the root password is provided as an environment variable.

To make containers accessible from outside, we use **port binding**. For example:

```
docker run --name mysql5 -d -e MYSQL_ROOT_PASSWORD=mysql123 -p 3307:3306 mysql
```

This maps **host port 3307** to **container port 3306**, allowing us to connect to MySQL from the host machine. Multiple containers can be run with different host ports (e.g., 3307, 3308, 3310).

Cleaning Up

When we are done, Docker allows us to stop and remove containers and images to free up resources:

```
docker stop mysql5        # stop the container
docker rm mysql5          # remove the container
docker rmi mysql          # remove the image
```

Keeping Container Alive

When you start a Docker container with just an image like `ubuntu`, it exits immediately because there's no process to keep it running. A container only stays alive **as long as there's a foreground process** inside it.

To keep it alive temporarily, you can run a dummy command like **sleep**:

```
docker run --name ubuntu2 ubuntu sleep 100
```

This keeps the container running for 100 seconds before stopping.
If you want to use the container interactively, start it with **-it**:

```
docker run -it ubuntu
```

This opens a shell inside the container where you can type commands.

Running Real Services

The main goal of containers is to **host applications** (not just run dummy commands).

A container keeps running **as long as the service inside it is running**.

For example:

```
docker run -d nginx
```

This starts an NGINX web server in the background, and the container stays alive because the server process is active.

no active process = container stops. Real services keep the process (and container) alive.

Running NGINX in a Container

```
sudo docker run -d nginx
```

```
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
d107e437f729: Pull complete
cb497a329a81: Pull complete
f1c4d397f477: Pull complete
f72106e86507: Pull complete
899c83fc198b: Pull complete
a785b80f5a67: Pull complete
6c50e4e0c439: Pull complete
Digest:
sha256:d5f28ef21aabddd098f3dbc21fe5b7a7d7a18
4720bc07da0b6c9b9820e97f25e
Status: Downloaded newer image for
nginx:latest
0949151f1cb5f2330fde10faeb09363f26f8fe9286ff
28e50934c8a2fddbdfd9
```

- Docker could not find the `nginx:latest` image locally.
- It automatically pulled the image from **Docker Hub** (official registry).
- Several layers were downloaded and combined to form the NGINX image.
- Docker started the container in **detached mode (-d)**, meaning it runs in the background.
- You now have an **NGINX web server** running inside a container.
- The container stays alive because the **nginx server process** keeps running.

How Docker Images Work: Layers

A **Docker image** is not one big file.

Instead, it is made up of **multiple layers**, stacked on top of each other.

What happened with `docker run -d nginx`:

- Docker saw you don't have the `nginx:latest` image locally.
- It pulled several **read-only layers** (each line you saw: "Pull complete").
- Each layer represents part of the filesystem (e.g., **Ubuntu base**, **libraries**, **NGINX binaries**, **config files**).
- Docker then **combines these layers into one final image**.

Why layers?

- Layers make Docker **efficient**:
  - If two images share a base (like Ubuntu), that layer is downloaded once and reused.
  - Updates only download new or changed layers, not the whole image again.

Docker Containers – Example

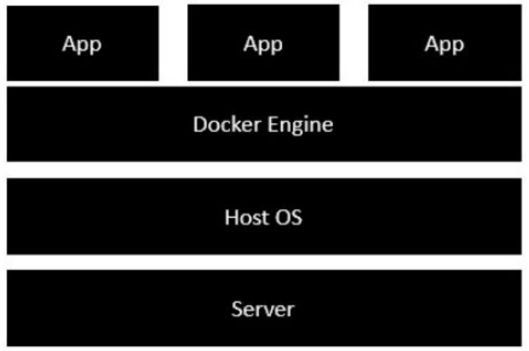
The **server** is the physical hardware used to host applications.

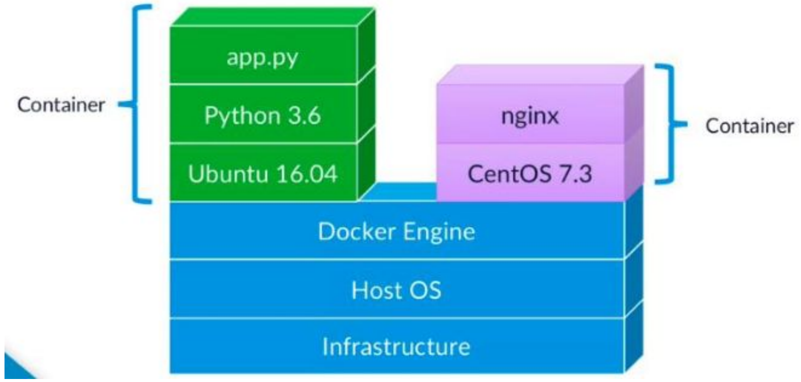
The **Host Operating System** (Linux/Windows) runs directly on this server.

The **Docker Engine** is installed on the Host OS to provide **containerisation**.

Applications run as **Docker containers**, each packaging its required environment (e.g., Python with Ubuntu, or NGINX with CentOS).

Containers share the Host OS kernel but remain **isolated** from one another, ensuring efficiency and consistency compared to full virtual machines.





Running a Container in Background (Detached Mode)

By default, if you run a container with a foreground process (e.g., `sleep 100`), your terminal is blocked until it finishes.

To avoid blocking, you can run the container in **detached mode** using the `-d` option.

Example:

```
docker run --name ubuntu3 -d ubuntu sleep 100
```

Here:

- `--name ubuntu3` → names the container.
- `-d` → runs it in background (detached).
- `sleep 100` → makes it stay alive for 100 seconds.

Docker immediately returns control to the terminal and gives you the **container ID**.
Use **`docker ps`** to check running containers in the same prompt.

NGINX

- **Open-source web server** (pronounced *engine-x*).
- Acts as **web server, reverse proxy, load balancer, caching server**.
- Designed for **high concurrency** and efficient handling of thousands of connections.
- Serves **static content** (HTML, CSS, images) and forwards dynamic requests to app servers.
- Widely used in production by **large-scale platforms** due to speed and scalability.

NGINX (engine-x) is a fast, lightweight web server that powers many modern websites. It can serve static files like HTML and images directly, or act as a reverse proxy that sits in front of application servers, balancing traffic and improving performance. Because it can handle thousands of users at once with low resource usage, NGINX has become the backbone of many high-traffic platforms, making it one of the most trusted tools for running web applications at scale.

Why Use NGINX if We Already Have ALB + Auto Scaling?

Different layers of traffic handling

- ALB + Auto Scaling work at the **cloud infrastructure level**.
- NGINX works at the **application server level**.

ALB + Auto Scaling

- Spread requests across multiple servers.
- Add/remove servers automatically based on load.

NGINX inside each server/container

- Acts as a **reverse proxy**, routing requests to the right service.
- Serves **static files** (HTML, images, CSS, JS) faster than app code.
- Provides **caching**, so frequent requests don't always hit the backend.
- Handles **TLS termination** and adds an extra layer of security.
- Uses an **event-driven model**, making it very efficient with thousands of concurrent connections.

Why Use NGINX if We Already Have ALB + Auto Scaling?

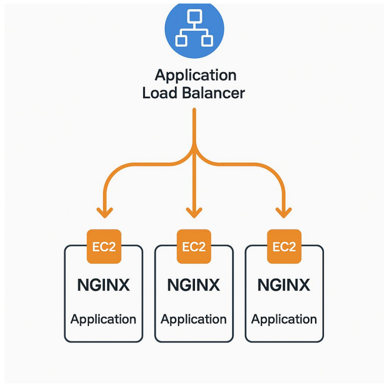
In a typical AWS setup:

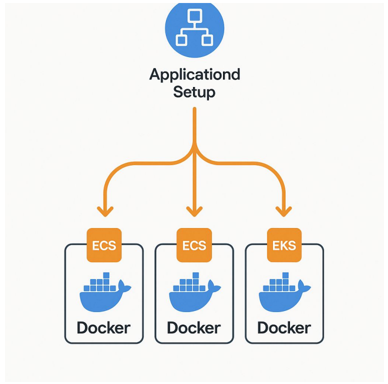
- **ALB (Application Load Balancer)** sits at the front, distributing traffic across multiple EC2 instances (or containers).
- Each **EC2 instance** (or container running on ECS/EKS) has **NGINX installed**.
- Inside the instance, NGINX acts as the **entry point**:
  - Serves static files directly.
  - Forwards dynamic requests to your backend app (e.g., Node.js, Python, Java service).
  - Handles SSL/TLS, caching, compression, etc.

EC2 vs Containers in AWS Architecture

In a **traditional VM-based setup**, you launch multiple EC2 instances. On each instance, you install NGINX and your application, and the ALB (Application Load Balancer) distributes traffic across these instances.

In a **containerised setup**, instead of installing applications directly on EC2, you package them as Docker containers. These containers still need a host — either an EC2 instance (self-managed) or a managed service like ECS (Elastic Container Service) or EKS (Elastic Kubernetes Service). The ALB then routes traffic directly to the containers running on those hosts.


Application Load Balancer
EC2
NGINX
Application
EC2
NGINX
Application
EC2
NGINX
Application


Applicationd Setup
ECS
Docker
ECS
Docker
EKS
Docker