

Unit 4:

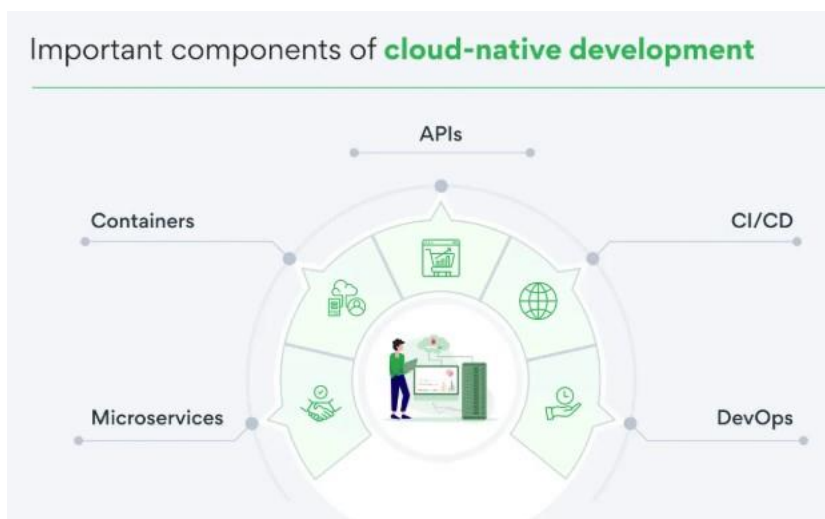
1. Cloud native development (Application Development in the Cloud)

Application development in the cloud refers to the complete process of **designing, building, testing, deploying, and maintaining software applications** using cloud computing platforms such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform.

In traditional software development, applications are hosted on **local servers or physical data centers**, which require significant investment in hardware, maintenance, and manual management. However, cloud-based development eliminates these limitations by using **remote servers hosted on the internet**, allowing developers to create applications that are **highly scalable, flexible, and globally accessible**.

Core components of cloud-native development

Cloud-native development is a modern way to build, deploy, and manage software directly in the cloud. It focuses on building applications that fully utilizes cloud computing's potential. Here are four major components of cloud-native that you should know:



1. Microservices

- Microservices are a collection of small, loosely coupled, independently deployable services that function collaboratively as cloud-native software.
- Cloud-native applications are **divided into smaller independent components** called microservices that communicate with each other via APIs.
- These microservices are **loosely coupled**, independently deployable, and work together to form a complete cloud-native system.
- Each microservice performs a **specific business function**, enabling independent development, deployment, and scaling.
- The **modular structure** helps applications adapt to varying demand levels while using resources efficiently and minimizing waste.
- It enhances flexibility, accelerates development cycles, and supports continuous delivery. Even if a single microservice fails, the overall application continues to operate smoothly.

- The number of microservices varies depending on project size, ranging from a few to hundreds.

2. Containers

- Containers are virtual packages for microservices and the smallest computing units of a cloud-native application. They bundle the microservice code and its necessary dependencies (e.g., resource files, libraries, and scripts) in cloud-native systems.
- They are considered the **smallest computing units** in cloud-native systems.
- A container orchestration system manages all components of a cloud-native application.
- Popular **container management platforms like Kubernetes** and Amazon ECS help in deploying, managing, and scaling containerized applications efficiently.

3. API

- An API (Application Programming Interface) **is a medium for communication between two or more** software systems. APIs is how the talking happens between microservices that are loosely coupled with each other.
- APIs **act as the glue** that connects different microservices. They enable communication and data exchange regardless of differences in underlying technologies.
- REST API is the most widely used protocol due to its simplicity and broad support.

4. CI/CD pipelines

- Cloud-native development fundamentally depends on a **Continuous Integration and Continuous Deployment (CI/CD) pipeline**, which **automates** and streamlines the entire software development lifecycle from coding to deployment.
- This pipeline enables a **collaborative and iterative workflow**, where developers, testers, and operations teams (DevOps) work **together efficiently by continuously** improving and delivering the application in small, incremental updates.
- In **Continuous Integration (CI)**, developers frequently commit their code changes to a **shared centralized repository** (such as Git), allowing multiple team members to contribute simultaneously while maintaining version control.
- In **Continuous Deployment (CD)**, once the code passes all testing stages, it is **automatically deployed** to production or staging environments without requiring manual intervention, enabling faster and more reliable releases.

5. DevOps

- DevOps is an **integral part of cloud-native development**, playing a key role in improving how applications are built, tested, and deployed.
- It combines **development (Dev) and operations (Ops)** teams to work collaboratively instead of functioning separately.
- It emphasizes **communication, collaboration, and shared responsibility** among team members throughout the software lifecycle.
- DevOps principles focus on the **automation of workflows**, including building, testing, integration, and deployment processes.
- DevOps practices enable teams to **deliver software applications more quickly**, reducing the time from development to production.

How do you choose the right cloud service or cloud service provider?

Choosing the right cloud provider is very important because it affects how well your applications run, how much you spend, and how easy it is to manage cloud services. A good provider can save money and make cloud management easier.

To choose the right provider, you should look at **major options** like **AWS, Azure, and Google Cloud**, and consider a few key factors:

- **Scalability:** All three scale easily. AWS offers the largest global reach, Azure excels in enterprise and hybrid setups, and GCP is strong for big data and analytics.
- **Pricing:** AWS has flexible but complex pricing, Azure is competitive for Windows environments, and GCP provides simple, transparent pricing with discounts for long-running workloads.
- **Support for Cloud-Native Tools:** AWS supports containers, serverless, and DevOps; Azure integrates well with Microsoft services and DevOps; GCP is excellent for Kubernetes, AI/ML, and analytics.

Each platform has its **own strengths**. For example, some are better for **AI/ML services**, while others work well with **enterprise systems or developer tools**.

Before choosing, clearly define your **project requirements, goals, and budget**. When you select a provider that fits your needs, using the cloud becomes **easier, more efficient, and cost-effective**, and your applications will run smoothly.

As a Cloud Service Consultant, my suggestions for choosing the right cloud provider are:

1. For Global E-commerce Platforms:

If your business needs **massive global infrastructure, fast scalability, and multiple services** for microservices, serverless apps, and databases, I recommend **AWS**. It offers **data centers worldwide**, a wide range of cloud services, and strong DevOps support.

2. For Enterprise Software Companies:

If your organization runs mostly **Windows-based enterprise applications** like SAP, Microsoft Dynamics, or Office 365, **Azure** is the best choice. It integrates seamlessly with Microsoft products, supports **hybrid cloud setups**, and simplifies user and identity management.

3. For AI-Powered Analytics Startups:

If you are building an **AI or machine learning platform**, choose **Google Cloud (GCP)**. It provides excellent tools like **TensorFlow, BigQuery, and advanced analytics**, and supports easy scaling for AI-heavy workloads.

4. For Financial Services Firms:

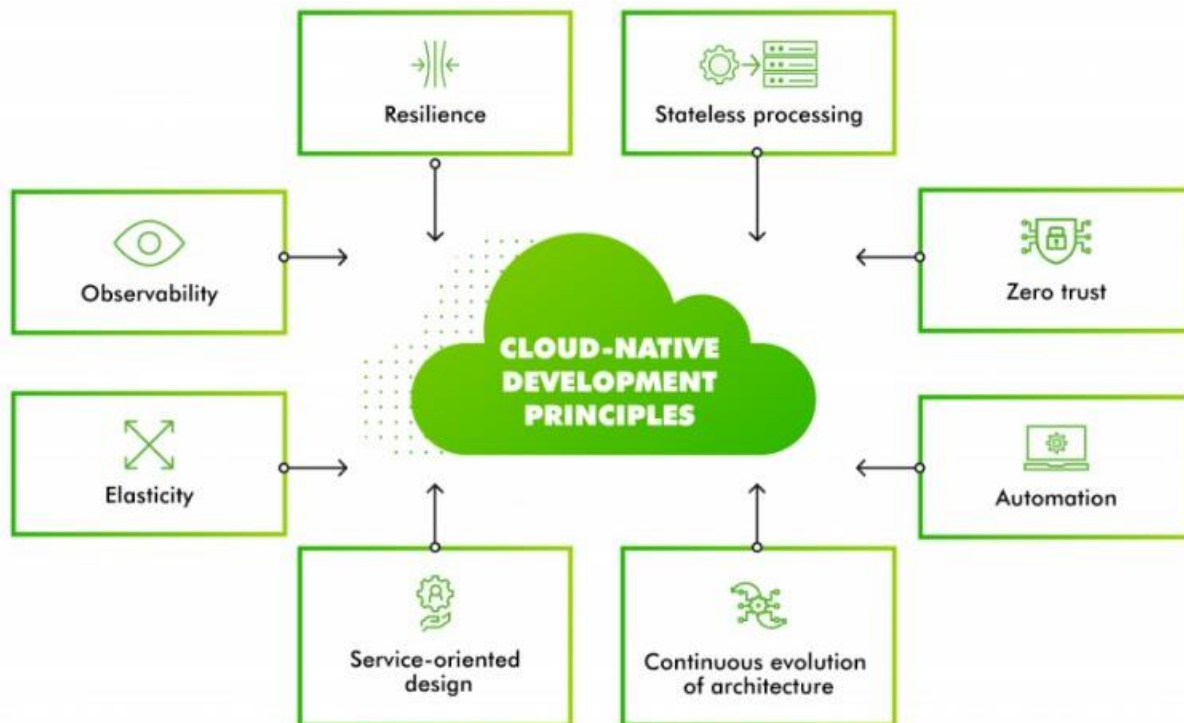
Banks or fintech companies needing **high security, compliance, and global reliability** should consider a **multi-cloud approach**: AWS for global infrastructure, Azure for enterprise integrations, and GCP for analytics.

5. For Media Streaming Companies:

If your service requires **fast content delivery, global scalability, and containerized microservices**, I suggest **AWS**, which provides a global network, scalable compute resources, and robust container orchestration tools.

8 Principles/Basis of Cloud-Native Development

Building cloud-native applications follows **eight key principles**. These principles act as **guidelines** for architects and developers. They help create applications that are **not just hosted in the cloud**, but can fully **take advantage of all cloud features**.



1. Service-oriented design

- This principle focuses on **breaking applications into smaller, modular, and independent services**, each responsible for a specific business function.
- **Microservices architecture** is a key example of this service-oriented approach, but the principle is broader and highly flexible.
- It can be applied to **serverless computing**, where individual functions or components encapsulate distinct pieces of business logic.
- Service-oriented design enhances **flexibility, scalability, resilience, and ease of management** in cloud-native applications.

2. Elasticity Principle

- The elasticity principle ensures that **cloud-native applications can handle changing workloads efficiently**, maintain performance, and use resources optimally.
- **Auto-scaling mechanisms** adjust the resources allocated to an application based on metrics like CPU usage, network traffic, or other performance indicators.
- Achieving elasticity involves **designing stateless services**, implementing **load-balancing mechanisms**, and using **distributed data storage**.
- This principle is key for **cloud cost optimization**, as it ensures that resources are only used for the current workload.

3. Observability Principle

Observability is the ability to **understand what is happening inside an application** by collecting and analyzing data. It helps developers and system administrators monitor performance, detect problems early, and keep the system running smoothly.

Observability mainly depends on three key components:

- **Metrics:** These are numerical values that show how the system is performing. For example, CPU usage, memory usage, number of requests, and response time.
- **Tracing:** Tracks the flow of requests across services, helping identify bottlenecks and understand dependencies
- **Logging:** Logs record detailed information about system events, errors, and activities. They are useful for debugging problems and understanding what happened in the past.

Another important part of observability is **alerting**:

- It sends notifications (email, message, dashboard alerts) when something unusual happens, such as high error rates or system failures. This allows teams to respond quickly before the problem becomes serious

4. Resilience Principle

Resilience is a **core principle** of cloud-native application. **It** means a cloud application can **handle problems and quickly recover** from failures or unexpected situations without stopping completely.

A resilient cloud system uses these main strategies:

- **Fault tolerance:**
The system can **detect problems and prevent them from affecting everything**.
- **Auto-healing:**
The system can **find issues and fix them automatically** without human help.
- **Redundancy:**
Backup systems are available, so if one fails, another can take over.
- **Graceful degradation:**
If there is a problem, the system **keeps working but with limited features** instead of stopping completely.

Cloud providers often **handle failures automatically**. For example, if one server fails, a new one is created to continue the work.

5. Automation Principle

- **Automation** is a key concept in cloud-native applications because these systems are made up of many different components (servers, services, containers, databases). Managing all these manually would be **time-consuming, complex, and prone to human errors**. Automation helps by allowing the system to **perform tasks automatically without constant human intervention**.
- Automation allows cloud systems to:
 - **Adapt to changing demand** (increase or decrease resources automatically)
 - **Reduce manual work** for developers and administrators
 - **Improve speed and efficiency** in managing applications
 - **Minimize human errors**, which are common in manual processes

- In cloud environments, many routine tasks are automated, such as:
 - **Server provisioning:** Automatically creating and configuring servers when needed
 - **Data backups:** Regularly saving data without manual action
 - **Resource management:** Detecting and removing unused or idle resources
 - **System updates:** Automatically applying patches and updates

6. Zero Trust Principle

- In traditional security systems, organizations protect their network using a **perimeter-based approach** (like a firewall). This means that once a user is inside the network, they are usually trusted.
- But in cloud environments, this is **not safe enough**, especially with modern technologies like serverless computing.
- **Zero Trust** means **no user or system is trusted automatically**, whether inside or outside the network.
- Zero Trust is a security model that says:” **Never trust, always verify.**”
 - This means:
 - No user, device, or system is trusted automatically
 - Every access request must be **verified every time**
- Zero Trust also emphasizes **encryption, multi-factor authentication, and authorization at every layer**, helping to protect sensitive data and resources.

7. Stateless System Principle

- In a stateless system, each client request is treated as a completely **independent task**, and the server does not store or remember any information from previous requests, ensuring that every interaction is fresh and self-contained.
- Because each request is complete on its own, **application design becomes simpler**, allowing developers to focus on building modular and independent components without worrying about shared session data.
- Requests can be **easily distributed across multiple servers**, which improves performance and makes it simple to **scale the system up or down** depending on demand, without affecting other parts of the application.
- Stateless systems are **more predictable and reliable**, which makes maintenance easier, simplifies troubleshooting, and ensures that debugging can be done efficiently without hidden dependencies between requests.

8. Continuous Evolution Principle

Continuous Evolution Principle means cloud applications are always **improving and adapting** over time.

- It involves **regularly reviewing and upgrading** technologies, methods, and practices.
- Cloud services and platforms keep adding **new features** that can replace custom-built components.
- Using these new features can **increase productivity, reduce costs, and meet changing business needs**.
- Continuous evolution ensures applications stay **modern, efficient, and up-to-date with technology**.

What Are Containers?

- A container is a **lightweight package** that includes an application along with all its dependencies, libraries, configuration files, and runtime environment.
- This ensures the application **runs the same way** on a developer's laptop, a testing system, or a production server.
- Containers **isolate applications**, so each one runs separately and does not interfere with others.
- Unlike virtual machines, containers **share the host operating system** and do not include a full OS, making them **smaller, faster, and more efficient**.
- Containers make applications **portable**, so they can move easily across different systems or cloud platforms without changes.
- They also help with **scalability and flexibility**, allowing multiple instances of an application to be deployed easily.
- A simple example is a **shipping container**, which securely packs cargo and can be transported anywhere; similarly, containers package applications to run consistently anywhere.
- One limitation is that containers **depend on the host OS**, so Linux containers run on Linux hosts, and Windows containers run on Windows hosts.

Architecture of Containers

A container-based system is composed of multiple layers and components that work together to ensure applications are **portable, scalable, efficient, and consistent across environments**.

1. Container Engine

- The container engine is the **main software responsible for building, running, and managing containers**.
- It provides the necessary tools and commands to **create container images, start and stop containers, and monitor their lifecycle**.
- It acts as an **interface between the user (developer/admin) and the underlying system**, simplifying container operations.
- A widely used example is **Docker**, which offers a complete platform for container development and deployment.

2. Container Image

- A container image is a **read-only, immutable template** that serves as a blueprint for creating containers.
- It includes everything needed to run an application, such as:
 - Application source code
 - Required libraries and dependencies
 - System tools and configuration files

- Images are built in layers, allowing **efficient storage and reuse of common components**.
- Once created, an image ensures that the application behaves **exactly the same across development, testing, and production environments**.

3. Container Runtime

- The container runtime is responsible for **executing containers from container images**.
- It ensures that each container runs in an **isolated and secure environment**, separate from other containers and the host system.
- The runtime manages **resource allocation**, such as CPU, memory, and storage, ensuring fair and efficient usage.
- It directly interacts with the **host operating system kernel**, using features like namespaces and control groups (cgroups) for isolation and resource control.

4. Host Operating System

- The host operating system provides the **foundation on which containers run**.
- All containers **share the host OS kernel**, which is the core part of the operating system.
- The host OS is responsible for **managing hardware resources** such as CPU, memory, disk, and networking.
- A key limitation is that containers must be **compatible with the host OS kernel** (e.g., Linux containers run on Linux hosts).

5. Orchestration Layer

- The orchestration layer is responsible for **managing large numbers of containers across multiple systems or clusters**.
- It automates complex operations such as:
 - **Deployment:** Launching containers automatically
 - **Scaling:** Increasing or decreasing container instances based on demand
 - **Load balancing:** Distributing traffic across containers
 - **Self-healing:** Restarting failed containers automatically
 - **Monitoring and updates:** Managing health and rolling updates
- A popular orchestration tool is **Kubernetes**, widely used in cloud-native environments.
- This layer is essential for **large-scale applications**, where manual management of containers is not practical.

How Containers Work

- Containers use **OS-level virtualization**, which means they share the host operating system instead of running separate operating systems like virtual machines.
- This makes containers **lightweight, fast, and efficient**, as they do not require full hardware virtualization.
- The core technologies that make containers work are **Namespaces** and **Control Groups (cgroups)**.
- **Namespaces provide isolation**, while **cgroups manage resource usage**.
- Together, they allow multiple containers to run **independently and securely on the same system**.
- Each container behaves like a **separate mini-environment**, even though all share the same OS kernel.

Features of Containers

1. Lightweight Nature

Containers are small and efficient because they do not include a full operating system and share the host OS kernel. This makes them fast to start, stop, and use minimal CPU, memory, and storage resources.

2. Portability

Containers can run consistently across different environments, including a developer's machine, private data centers, or public cloud platforms. All dependencies are included inside the container, so no modifications are needed when moving it.

3. Isolation

Each container runs independently with its own processes, file system, and network interface, so problems in one container do not affect others. This improves security, stability, and prevents interference between applications.

4. Scalability

Containers can be quickly duplicated or removed based on workload, allowing applications to handle changing traffic efficiently. Tools like Kubernetes automate scaling to maintain performance and save resources.

5. Rapid Deployment

Containers can be started in seconds, enabling fast application delivery and updates. This supports modern practices like Continuous Integration and Continuous Deployment, reducing time to market and improving reliability.

What is Containerization?

- **Containerization** is the process of **packaging an application together with everything it needs**—like libraries, dependencies, and configuration files—into a single unit called a **container**.
- This ensures the application **runs the same way** on a developer's laptop, testing server, or cloud platform.
- A container is like a **portable box** that includes everything the app needs, so differences in operating systems or missing software do not cause problems.
- Each container runs independently, ensuring **security, consistency, and minimal interference** between applications.
- This approach follows the principle “**build once, run anywhere**”, because the same container works on any system with a container runtime.
- Containers are widely used in **modern development**, including microservices, DevOps, and cloud-native applications, because they make **deployment, scaling, and management easier**.

What is Container Orchestration ?

Container orchestration is the process of **automatically managing and coordinating multiple containers** in large-scale systems. It gives administrators a **clear view of where containers are running** and how workloads are distributed.

- Orchestration is important when containers are running across **different servers or cloud platforms**, because managing each container manually is slow and prone to errors.
- Orchestration tools let IT teams define **rules for container behavior**, like fault tolerance, resource use, scaling, and recovery. These rules can apply to a single container or a group of containers.
- A key feature is **automatic workload management** across multiple servers (nodes). For example, if one server goes into maintenance, the orchestrator **redirects workloads to other servers** so performance continues smoothly.
- This process happens **without human intervention**, which improves reliability, scalability, and fault tolerance for distributed applications.
- Popular tools like **Kubernetes** can **scale applications automatically, restart failed containers, and manage updates smoothly**, ensuring minimal downtime and consistent performance.

What Is Kubernetes?

- Kubernetes, or K8s for short, is an open-source tool that helps automatically **deploy, manage, and scale applications** that run in containers. It is very popular in modern cloud computing.
 - “Kubernetes” comes from Greek, meaning **helmsman or pilot**. Just like a ship’s pilot steers a ship, Kubernetes guides applications, keeping them balanced, scalable, and running smoothly. Its ship-wheel logo represents this control.
- 
- Kubernetes is like a **pilot for applications in containers**. It ensures that apps run efficiently, stay online, and can handle more users when needed. It is a key tool for modern cloud computing and DevOps.
 - In today’s world, applications are not just one big program (monolith). They are split into many containers, often running on different computers. Managing all these containers manually is hard, especially when applications grow or need to handle more users.
 - Kubernetes helps by **automatically organizing and managing containers** across many computers, so developers and IT teams don’t have to handle each one by hand.

Kubernetes Architecture/ Core Components of Kubernetes:

Kubernetes is a **container orchestration platform** that manages containerized applications across multiple servers. Its architecture is divided into two main parts: the **Control Plane** and **Worker Nodes**.

1. Control Plane (Master Node)

The **Control Plane**, also called the **Master Node**, is the brain of the Kubernetes cluster. Its primary role is to **control, manage, and monitor the cluster**, making decisions about scheduling, scaling, and maintaining the desired state of applications.

Key Components of the Control Plane:

1. API Server

- The **API Server** acts as the **entry point** for all administrative commands and interactions with the cluster.
- It communicates with the Worker Nodes to instruct them on what containers or pods to run.

2. Scheduler

- The **Scheduler** is responsible for **assigning pods to worker nodes** based on resource availability, policies, and workload requirements.
- It ensures that containers are deployed on nodes with enough CPU, memory, and other resources to run efficiently.

3. Controller Manager

- The **Controller Manager** continuously monitors the cluster and ensures the **desired state** of the applications.
- For example, if a pod fails or a node goes down, the controller manager will automatically **restart or reschedule containers** on available nodes.

4. etcd (Optional, for completeness)

- **etcd** is a distributed key-value store that stores **all cluster configuration data and state**.
- It ensures consistency and acts as a single source of truth for the cluster.

2. Worker Nodes

Worker Nodes **take instructions** from the Master Node and **run the actual workloads/applications**. They host containers, packaged inside **pods**, which execute the application workloads.

Key Components of Worker Nodes:

1. Kubelet

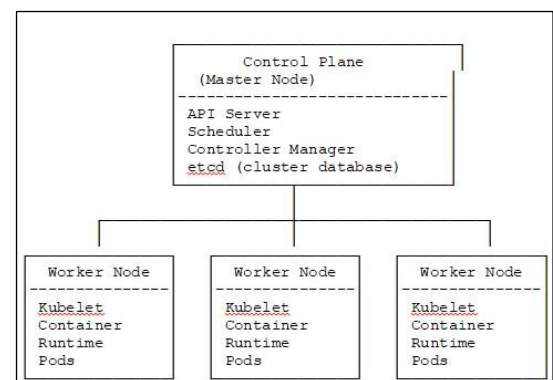
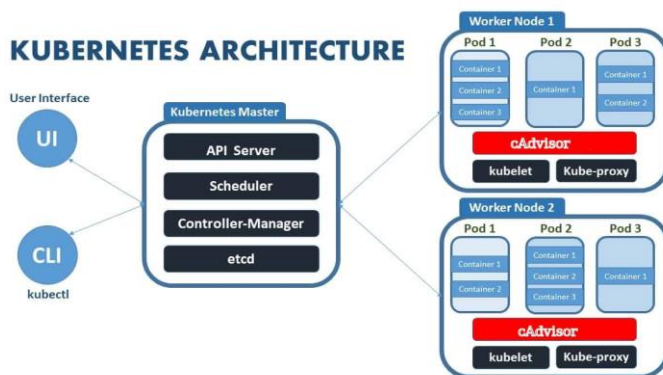
- The Kubelet is an **agent running on each worker node** that communicates with the Control Plane.

2. Container Runtime

- This is the software responsible for **running the containers**, such as **Docker** or **containerd**. It pulls container images, starts/stops containers, and handles container execution.

3. Kube-Proxy

- Kube-Proxy manages **networking and communication** within the node and across the cluster.
- It handles **load balancing** and routing traffic to the correct pods.



3. Pods

- A **Pod** is the smallest deployable unit in Kubernetes. It can contain **one or more containers** that work closely together.
- All containers in a pod **share the same network, ports, and storage**, so they can communicate easily.
- Pods are **temporary**, meaning they can be created, destroyed, or replaced automatically when needed or if something fails.
- Kubernetes uses pods to **manage scaling, updates, and replication** of containers efficiently.

Why Kubernetes is Needed?

- **Managing many containers:**

Modern applications are no longer just a single program. They are made up of **hundreds or even thousands of small containers**, each doing a part of the application. These containers often run on **different computers or servers**. Managing all of them manually is extremely difficult and time-consuming. Kubernetes helps **organize and control all these containers automatically**.

- **Coordinating multiple servers (nodes):**

Large applications usually run across **multiple servers, called nodes**, and each server can have several containers. If done manually, it's easy to make mistakes—like putting too many containers on one server or not distributing them efficiently. Kubernetes automatically **balances containers across all nodes**, so resources are used efficiently and nothing gets overloaded.

- **Handling high user traffic:**

Applications can face sudden spikes in traffic, like during **sales events, launches, or viral trends**. Without automation, it's very hard to **quickly add more containers or servers** to handle the extra users. Kubernetes can **automatically scale** the application up or down depending on the demand, ensuring it stays fast and responsive.

- **Ensuring reliability and uptime:**

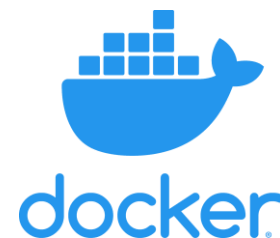
Containers or servers can sometimes **fail or crash**. Manually fixing this can take time and affect users. Kubernetes continuously **monitors the containers**, and if one fails, it **restarts or replaces it automatically**, keeping the application running smoothly without downtime.

- **Automation and efficiency:**

Overall, Kubernetes reduces the need for manual work. It **automates deployment, scaling, and recovery** of applications, saving time for developers and IT teams. This allows organizations to focus on **building new features** instead of constantly managing infrastructure.

What is Docker?

- Docker is a popular tool that helps developers and IT teams **create, run, and manage containers** easily. It makes working with containers simple and provides a standard way to run applications.
- Applications built with Docker can run **anywhere**—on a developer's computer, on a company server, or in the cloud (like **AWS, Azure, or Google Cloud**)—without changes.
- Docker is used on many **Linux systems** like Red Hat and Ubuntu, and by big companies such as **Microsoft, Amazon, and Oracle**, showing it works well across platforms and industries.



Kubernetes Vs Docker

- Docker handles **building and running individual containers**.
- Kubernetes handles **orchestrating, scaling, and managing multiple containers** across a cluster.
- Together, they enable **efficient, scalable, and reliable deployment of modern cloud-native applications**.
- **Docker** is like a **single shipping container**, packing your goods (application) in a standardized, portable form.
- **Kubernetes** is like a **shipping port control system**, managing hundreds of containers—making sure they're delivered to the right location, balanced across ships, and quickly rerouted if a ship fails.
- **Docker provides the containers, and Kubernetes manages them at scale.**
- You **cannot deploy Kubernetes without containers**—containers are the building blocks—but **you don't need Kubernetes to run containers** on a single machine.
 - **Docker = individual containers**
 - **Kubernetes = fleet manager for containers**
-

