Aim Of Module

The micro-services component of this module is based on Java. It mainly focuses on structured design, implementation, testing, and documenting. The cloud computing component will give an overview and an in-depth study of cloud computing, enabling students to become practitioner or carry out research in this domain.

Learning Outcomes

On successful completion of this module, students should be able to:

- LO-1. Explain the microservices architecture.
- LO-2. Understand microservices design principles.
- LO-3. Understand and implement microservices integrations.
- LO-4. Learn about popular technologies related to microservices.
- LO-5. Understand monolithic decomposition and helpful practices when deploying a monolithic application.
- LO-6. Understand inner operations of deploying and maintaining microservices.
- LO-7. Explain the core concepts of the cloud computing paradigm: how and why this paradigm shift came about, the advantages and challenges brought about by the various models and services in cloud computing.
- LO-8. Apply fundamental concepts in cloud infrastructures to understand the trade-offs in power, efficiency and cost, and then study how to leverage and manage single and multiple data centers to build and deploy cloud applications that are resilient, elastic and cost-efficient.
- LO-9. Discuss system, network and storage virtualization and outline their role in enabling the cloud computing system model also high light the multi cloud approach.
- LO-10. Illustrate the fundamental concepts of cloud storage and demonstrate their use in storage systems such as Amazon S3 and HDFS.
- LO-11. Analyses various cloud programming models and apply them to solve problems on the cloud.

Topics in Detail

LO-1. The Emergence of Microservices Architecture

- Explore the ideal software development practice
- Learn how a fine-grained Service-Oriented Architecture (SOA) can help to achieve the ideal
- Learn how Microservices attempts to achieve the ideal

LO-2. Microservice Design Principles

- Designing small microservices
- Designing independent microservices
- Designing resilient microservices

LO-3. Integrating Microservices

- Understand design goals when integrating microservices
- Explore effective message formats and lightweight inter-service communication approaches
- Review the pros and cons of various service communication patterns

LO-4. Microservice Technologies

- Learn about popular technologies that enable the development, deployment, and support of Microservices.

LO-5. Decomposing the Monolith

- Understand monolithic decomposition as an approach towards application modernization
- Review successful decomposition patterns

- Understand helpful practices when decomposing a monolithic application

LO-6. Deploying and Maintaining Microservices

- Explore the intersection of DevOps and microservices
- Learn to leverage virtual, cloud, and containerized environments for microservice deployment
- Discover how to monitor a microservices environment and take appropriate action to enable scaling or react to system faults
- Service Mesh Implementation with Microservices

LO-7. Introduction to Cloud Computing

- What is cloud computing
- Evolution of cloud computing
- Technologies, Services, and Deployment models
- Benefits, pitfalls, risks, and challenges in cloud computing
- Economic models and SLAs
- Cloud security overview

LO-8. Cloud Infrastructure

- History of data centers
- Components of data centers
- Design considerations: requirements, power, efficiency, redundancy power calculations, and data center challenges
- Cloud management and cloud software deployment
- Multi-Cloud Approach

LO-9 Virtualization

- Virtualization (CPU, memory & 1/0).
- Amazon EC2
- Software Defined Networks
- Software Defined Storage

LO-10. Cloud Storage

- Introduction to storage systems
- Cloud storage concepts
- Distributed file systems
- Cloud databases (HBase, MongoDB, Cassandra, DynamoDB)
- Could object storage (Amazon S3, OpenStack Swift)

LO-11. Programming Models

- Cloud-distributed programming
- Data-parallel analytics with Hadoop MapReduce

Session – 01

Enterprise Approach in Software Development

What is an Enterprise?

An enterprise refers to any organized collection of entities or organizations that share a common purpose or goal.

The TOGAF Standard defines an enterprise as encompassing entities that work collaboratively to achieve shared objectives.

Examples include:

- Corporations: A whole company or specific divisions within it (e.g., a finance or marketing division).
- Government Bodies: Entire agencies or specific departments (e.g., a health ministry or defense department).
- Geographically Distributed Organizations: Chains or franchises linked by common ownership (e.g., multinational companies).
- Collaborative Groups: Countries, militaries, or intergovernmental organizations working on shared deliverables or infrastructure.
- Business Alliances: Partnerships, consortia, or supply chains where companies collaborate to meet mutual goals.

Enterprise Approach in Software Development

The Enterprise Approach is a strategic methodology used to design, develop, and manage software systems at an organizational level. It focuses on meeting the diverse and complex needs of enterprises by ensuring systems are scalable, reliable, and aligned with business objectives.

Key Principles of the Enterprise Approach

Alignment with Business Goals:

- Ensures that software development supports the organization's strategic objectives.
- Focuses on solving real business challenges rather than isolated technical problems.

Standardization:

- Promotes the use of common frameworks, tools, and practices to improve efficiency and reduce redundancy.
- Enables interoperability across departments and systems.

Modularity and Scalability:

- Systems are designed as modular components to simplify maintenance and upgrades.
- Ensures that solutions can scale with organizational growth.

Shared Resources and Infrastructure:

- Encourages the reuse of existing resources and platforms across the enterprise to optimize costs.
- Example: Cloud infrastructure shared across multiple departments.

Collaboration Across Stakeholders:

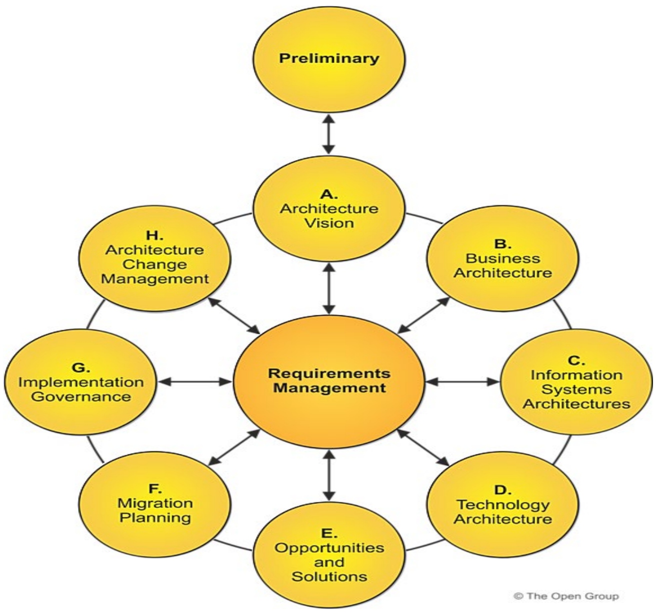
- Involves input from diverse departments and teams to ensure a comprehensive solution that meets enterprise-wide needs.

Relevance to Software Development

- Integration: Facilitates seamless integration of systems across different departments, regions, or organizations within the enterprise.
- Adaptability: Supports quick adjustments to changing business needs or market conditions.

- Efficiency: Reduces duplication of efforts and streamlines processes through shared infrastructure and reusable frameworks.
- Sustainability: Ensures the system remains functional, efficient, and relevant over the long term.

Enterprise Architecture Development Cycle


Preliminary
A. Architecture Vision
B. Business Architecture
C. Information Systems Architectures
D. Technology Architecture
E. Opportunities and Solutions
F. Migration Planning
G. Implementation Governance
H. Architecture Change Management
Requirements Management
© The Open Group

Overview of TOGAF

- TOGAF (The Open Group Architecture Framework) is a widely adopted framework for developing, managing, and implementing enterprise architectures.
- It ensures alignment between an organization's IT and business strategies, focusing on delivering business value through efficient IT systems.
- The framework is flexible and iterative, allowing organizations to adapt it to their specific needs.

The TOGAF Architecture Development Method (ADM) Cycle

1. Preliminary Phase:

- Define the scope, objectives, and stakeholders of the architecture effort.
- Set the direction by understanding the business context and priorities.

2. Architecture Vision:

- Establish a high-level view of the desired future state of the architecture.
- Define principles, constraints, and target architectures.

3. Business Architecture:

- Analyze and document business processes, capabilities, and organizational structures.
- Ensure alignment between IT and business objectives.

4. Information Systems Architectures:

- Design IT systems to support business processes.
- Focus on data models, application architectures, and technology platforms.

5. Technology Architecture:

- Define the infrastructure, including hardware, software, networks, and data centers.
- Ensure the technology stack supports the business and IT systems effectively.

6. Opportunities and Solutions:

- Identify potential solutions and assess their feasibility.
- Perform cost-benefit analyses to make informed decisions.

7. Requirements Management:

- Gather, analyze, and document business and technical requirements.
- Ensure the architecture meets organizational needs.

8. Migration Planning:

- Create a roadmap for transitioning from the current state to the target state.
- Address risks, change management, and implementation planning.

9. Implementation Governance:

- Monitor and control the implementation to ensure alignment with the strategy.

- Adapt the architecture based on real-world feedback.

10. Architecture Change Management:

- Manage changes to the architecture over time.
- Update documentation, assess the impact, and maintain consistency.

Enterprise Approach — Deep Dive

What Clients Expect

Clients demand expertise, professionalism, and a comprehensive, enterprise-grade approach to deliver solutions that align with their business goals. Here are the key expectations:

Not an Outcome of a POC (Proof of Concept):

- Clients expect full-fledged solutions, not just experimental prototypes.
- Deliverables must be production-ready, scalable, and aligned with long-term goals.

No Learners:

- They want experienced professionals, not individuals who are learning on the job.
- Teams must demonstrate confidence, competence, and domain knowledge.

Versatilists Over Specialists:

- Clients value versatilists — professionals who can adapt, think holistically, and work across domains, instead of siloed specialists.

Guidance from a Consultant:

- Beyond execution, clients expect strategic advice.
- Consultants should act as trusted advisors, offering actionable insights and steering the project toward success.

Value Addition:

- Solutions should deliver measurable business value.
- Clients look for innovations, optimizations, and a clear return on investment (ROI).

Enterprise Approach:

- Clients expect a structured methodology that integrates business goals with IT capabilities.
- Solutions should address scalability, maintainability, and compliance with organizational policies.

Delivery Team Focus Areas

The delivery team must focus on three core architectural layers:

1. Business Architecture

Business Architecture defines the structure and operation of an organization, focusing on its goals, processes, and capabilities. It ensures alignment between business objectives and IT solutions. This layer identifies **what the business does**, **how it operates**, and **how it delivers value** to stakeholders.

Business Case:

- Demonstrate the project’s value to stakeholders, including ROI and strategic benefits.
- Build compelling justifications for resource and budget allocations.

Requirements Understanding:

- Gather red-hot requirements with absolute clarity.
- Use tools like use case diagrams to visually represent requirements.

Project Charter and SRS:

- Create a Project Charter to outline objectives, scope, and constraints.
- Draft the Software Requirements Specification (SRS) to capture detailed

functional and non-functional requirements.

Process Maps & User Journeys:

- Develop process maps to represent workflows, business processes, and user journeys.
- Highlight pain points and opportunities for process improvement.

Policy Documents:

- Draft and secure sign-offs from stakeholders for business processes.
- Ensure alignment with organizational policies and compliance standards.

User Stories Linked to Process Maps:

- Write user stories that directly correlate with the process maps to ensure traceability.

Business Use Cases:

- Develop business use cases to illustrate how the solution solves real-world problems.

UI/UX with User Journeys:

- Design user interfaces and experiences that align with the defined user journeys.
- Ensure intuitive, user-centric designs.

2. Information Systems Architecture

Information Systems Architecture focuses on designing and organizing the IT systems that support business processes. It defines how **data, applications, and IT systems** interact to meet organizational needs.

Software Architecture

Software architecture defines the high-level structure of a software system. It involves decisions on how the system is organized, how components interact, and how they achieve goals like modularity, scalability, and maintainability.

Example:

Imagine an e-commerce application:

- It has separate modules for User Management, Product Catalog, Order Processing, and Payments.
- Each module has clearly defined roles and responsibilities, making it easier to scale and maintain.

Architectural Patterns

Architectural patterns are templates or best practices for organizing software systems. They provide a blueprint to solve specific architectural problems effectively.

- MVC (Model-View-Controller)
- Microservices
- Event-Driven Architecture

Design Patterns

Design patterns are reusable solutions to common problems encountered during software development. They focus on solving challenges within components or modules.

- Singleton Pattern
- Factory Pattern
- Repository Pattern

3. Technology Architecture

Technology Architecture outlines the technical infrastructure and platforms that support the information systems and business processes. It focuses on the hardware, software, networks, and tools required for operations.

Operating Systems (OS):

- Ensure the solution is compatible with enterprise-grade operating systems (Linux, Windows Server, etc.).

Cloud-Native Technologies:

- Focus on scalability, elasticity, and resilience using cloud-native principles (e.g., Kubernetes, Docker).

Low Code / No Code:

- Utilize platforms like OutSystems, Mendix, or PowerApps for rapid development of specific components.

Development Platform:

- Choose robust platforms like .NET Core, Spring Boot, or Node.js based on the project's requirements.

Data Engineering:

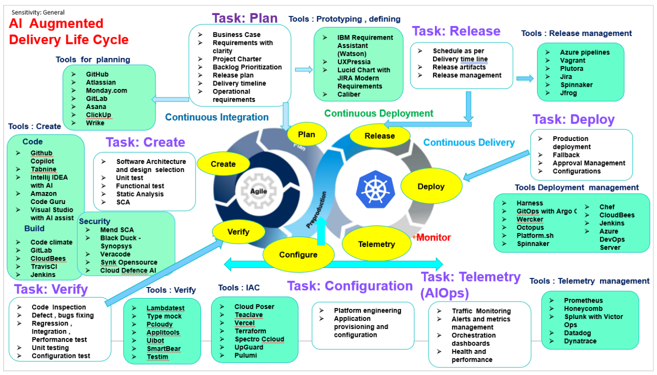
- Design systems for effective data storage, processing, and analytics (e.g., ETL pipelines, data lakes).

Middleware:

- Implement middleware solutions to enable seamless integration and communication between systems.

AI Augmented Delivery Life Cycle

The AI-Augmented Delivery Life Cycle refers to a project or product development framework where Artificial Intelligence (AI) is integrated into every phase of the delivery process to enhance efficiency, decision-making, and outcomes. By leveraging AI tools and techniques, organizations can optimize processes such as planning, development, testing, deployment, and monitoring, ultimately driving innovation and reducing time-to-market.



Key Benefits of the AI-Augmented Delivery Life Cycle

- Enhanced Efficiency: Automates repetitive and time-consuming tasks.
- Improved Accuracy: Reduces human errors in requirements gathering, coding, and testing.
- Predictive Insights: Anticipates risks, failures, and bottlenecks early.
- Continuous Learning: AI learns from past projects to improve future delivery processes.
- Faster Time-to-Market: Streamlines processes, enabling quicker delivery cycles.

Session - 02

Architecture / AR Patterns / Design Patterns

Software Architectural Styles

Software Architectural Styles refer to standardized, conceptual ways of designing and organizing software systems. Each style defines a set of principles, structures, and rules that guide the development of a system to achieve specific goals, such as scalability, maintainability, or reusability.

Architectural styles provide a high-level blueprint for how components interact within a system and how the system operates as a whole. They help in creating robust and efficient software solutions by offering predefined approaches for solving recurring design problems.

1. Client-Server Architecture

Definition: A two-tier architecture where a client requests services or resources from a centralized server. The server processes the requests and sends back responses.

Key Features:

- Centralized server for data and services.
- Clients interact with the server via a defined protocol (e.g., HTTP, TCP/IP).

Example:

Web Applications: A browser (client) sends HTTP requests to a web server (e.g., Apache or IIS), which processes the request and sends back a webpage.

2. Component-Based Architecture

Definition: Decomposes the system into independent, reusable components, each with a well-defined interface for communication.

Key Features:

- Promotes reusability and maintainability.
- Components are location-transparent (don't depend on physical location).

Example:

A banking application where components like Account Management, Transaction Processing, and Reporting can be reused across different projects.

3. Layered Architecture

Definition: Organizes the system into layers (e.g., presentation, business, data, and service layers). Each layer has specific responsibilities and interacts only with its adjacent layers.

Key Features:

- Clear separation of concerns.
- Easier to manage and test individual layers.

Example:

A web application with:

Presentation Layer: HTML, CSS, JavaScript.

Business Layer: Logic and rules implemented in .NET or Java.

Data Layer: Relational database like SQL Server or MySQL.

4. Hexagonal Architecture

Definition: In this style, the application's core logic communicates with external systems via well-defined ports and adapters.

Key Features:

- Promotes flexibility and testability.
- Decouples the core logic from external dependencies.

Example:

A payment processing system where the core logic interacts with multiple payment gateways via adapters (e.g., PayPal, Stripe).

5. Message-Bus Architecture

Definition: A software system where components communicate asynchronously via a central message bus, which routes messages based on predefined rules.

Key Features:

- Decouples components from direct dependencies.
- Promotes scalability in distributed systems.

Example:

E-commerce platforms using message brokers like RabbitMQ or Kafka to handle events such as order placement, payment confirmation, and inventory updates.

6. N-Tier (3-Tier) Architecture

Definition: Divides the system into multiple physical tiers, typically including a presentation tier, application tier, and data tier.

Key Features:

- Tiers can run on separate machines, improving scalability.
- Separation of concerns between layers.

Example:

A mobile banking app:

- Presentation tier: Mobile app interface.
- Application tier: Middleware for processing transactions.
- Data tier: Backend database.

7. Object-Oriented Architecture

Definition: Focuses on dividing the system into self-sufficient objects, each encapsulating data and behavior relevant to that object.

Key Features:

- Promotes reusability and modularity.
- Based on principles like encapsulation, inheritance, and polymorphism.

Example:

A shopping cart application where Cart, Product, and User are objects, each with its own properties and methods.

8. Separated Presentation

Definition: Separates the user interface (presentation) from the business logic and data to enable independent development and easier maintenance.

Key Features:

- Promotes UI/UX flexibility.
- Reduces coupling between layers.

Example:

- MVC Architecture:
- View: HTML for user interface.
- Model: Database schema and data processing.
- Controller: Handles user interactions and updates.

9. Service-Oriented Architecture (SOA)

Definition: Organizes applications into a collection of services, each providing specific functionality. These services communicate using APIs or messages.

An Enterprise Service Bus (ESB) server is a middleware solution that facilitates communication between different services in a Service-Oriented Architecture (SOA). It acts as a central hub that connects, mediates, and orchestrates interactions between services, applications, and data sources within an enterprise.

In Service-Oriented Architecture (SOA), multiple independent services communicate with each other. Without an ESB, integrating these services can become complex, requiring direct point-to-point connections between services.

Key Features:

- Promotes interoperability across platforms.
- Services are loosely coupled and reusable.

Example:

A travel booking system where separate services handle flights, hotels, and car rentals, all exposed via APIs.

10. Microservices Architecture

Definition: Develops an application as a suite of small, independently deployable services. Each service is focused on a specific business capability and communicates via lightweight protocols (e.g., HTTP).

A Discovery Server (also known as a Service Registry or Service Discovery) is a key component in a Microservices Architecture that helps dynamically locate and manage services in a distributed system. It allows microservices to register themselves and enables other services to discover and communicate with them without needing hardcoded IP addresses or URLs.

In a microservices-based system, services are dynamically deployed across multiple servers, containers, or cloud instances. Hardcoding service locations (IP addresses and ports) is impractical because:

- Services scale up/down dynamically (due to auto-scaling).
- Services may restart or move to different machines or containers.
- Load balancing needs dynamic service discovery.

Example for Discovery Servers

- Netflix Eureka (Spring Boot)
- Consul (HashiCorp)
- Zookeeper (Apache)
- Kubernetes Service Discovery

A Discovery Server helps manage these dynamic service instances efficiently.

Key Features:

- Decentralized management and deployment.
- Services can use different programming languages and databases.

Example:

Netflix:

- Individual services for user profiles, recommendations, and content delivery.

11. Cell Architecture

Definition: An emerging style where the enterprise system is divided into smaller, autonomous units or cells, each capable of functioning independently while interacting with others.

Key Features:

- Promotes scalability and resilience.
- Each cell contains all necessary components for a specific business function.

Example:

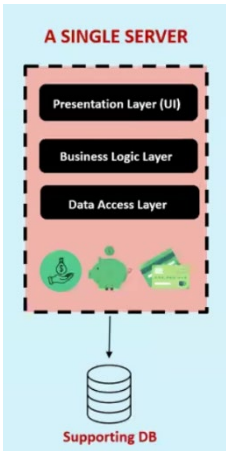
A telecom billing system where individual cells handle billing for different regions or services.

Session - 03

More Deeper Explanation About,

- "Monolith"
- "SOA"
- "Microservices Architecture"

Monolithic Architecture



A Monolithic Architecture is a traditional software development approach where the entire application, including the UI, business logic, and database, is deployed as a single unit within a single server. This approach was widely used a decade ago before Microservices Architecture became popular.

Pros:

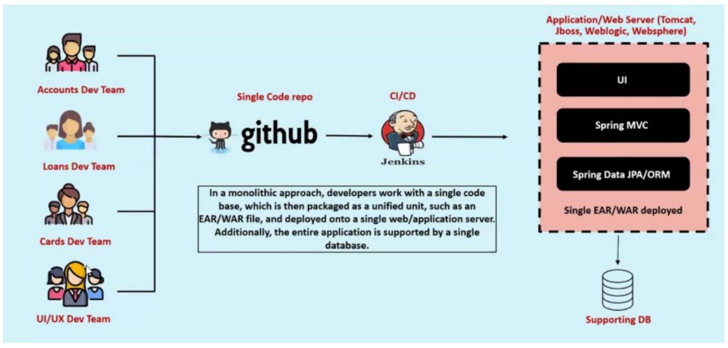
- **Simpler Development & Deployment** – Easier to develop and deploy since everything is in a single repository and build pipeline.
- **Fewer Cross-Cutting Concerns** – Security, logging, and error handling can be centrally managed.
- **Better Performance** – No network latency between services since everything runs within the same process.
- **Easier Debugging & Testing** – Since all components are in one system, debugging and local testing are straightforward.

Cons:

- **Difficult to Adopt New Technologies** – Upgrading specific components is challenging without affecting the whole system.
- **Limited Agility & Scalability** – Scaling requires deploying multiple copies of the entire application, rather than scaling individual components.
- **Single-Codebase Complexity** – As the application grows, maintaining a large monolithic codebase becomes challenging.
- **Not Fault Tolerant** – If one module crashes, it can bring down the entire application.
- **Full Deployment for Small Changes** – Even minor updates require rebuilding and redeploying the entire system.

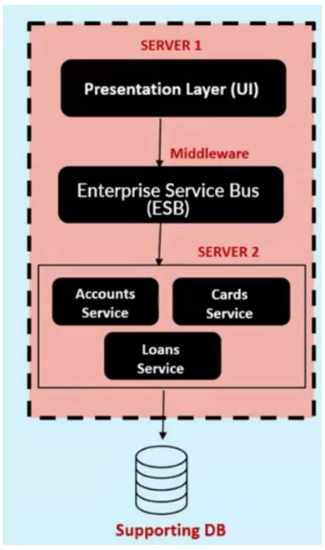
When to Choose Monolithic Architecture?

- For small applications & startups – If you're building a simple web app with limited features.
- For teams with limited expertise – When a small team is working on the project and managing a distributed system is too complex.
- For applications with tight performance constraints – If network calls between microservices would introduce unnecessary latency.



In a monolithic approach, developers work with a single code base, which is then packaged as a unified unit, such as an EAR/WAR file, and deployed onto a single web/application server. Additionally, the entire application is supported by a single database.

Service-Oriented Architecture(SOA)



Service-Oriented Architecture (SOA) is an architectural style that focuses on creating applications by composing a collection of loosely coupled and interoperable services. These services can communicate over a network and are designed to be independent, reusable, and modular. SOA aims to break down large, monolithic applications into smaller, more manageable components that can be developed, deployed, and maintained separately.

In SOA, each service represents a specific business functionality or process and can be accessed over a network using standard communication protocols such as SOAP (Simple Object Access Protocol) or REST (Representational State Transfer). These services are designed to be language-agnostic, allowing different technologies to interact with each other.

Pros:

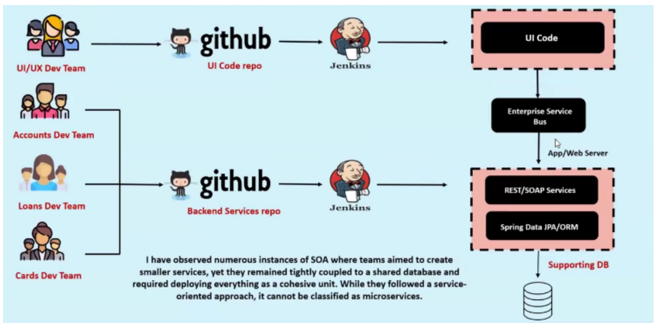
- **Reusability of Services** - Services in SOA are designed to be modular and can be reused across different applications or systems.
- **Better Maintainability** - As SOA breaks down the application into smaller services, maintaining individual services becomes easier.
- **Higher Reliability** - With services being independent, a failure in one service does not necessarily bring down the whole system.
- **Parallel Development** - Different teams can work on different services simultaneously, improving the development speed.

Cons

- **Complex Management** - SOA typically uses communication protocols such as SOAP or XML-based messages, which can be complex to manage, especially in large systems.
- **High Investment Costs in Middleware** - SOA often requires an Enterprise Service Bus (ESB) or middleware for communication and message routing.
- **Extra Overhead** - Communication between services introduces latency, especially when using SOAP (which is more heavyweight than REST).

When to Use Service-Oriented Architecture (SOA)?

- When building large, enterprise-level applications that require integration between different systems (e.g., ERP, CRM, legacy systems).
- When you need to ensure interoperability between systems built using different technologies.
- For systems that need reliable, reusable services that can be scaled independently.



I have observed numerous instances of SOA where teams aimed to create smaller services, yet they remained tightly coupled to a shared database and required deploying everything as a cohesive unit. While they followed a service-oriented approach, it cannot be classified as microservices.

Microservices Architecture

Microservices Architecture is an architectural style that structures an application as a collection of independent services, each responsible for a distinct business domain. Each service in a microservices-based system is self-contained, with its own database and business logic, and can be deployed, scaled, and updated independently. These services communicate with each other via lightweight protocols, usually over HTTP (e.g., using REST APIs or messaging queues).

Each microservice is designed to solve a specific problem or perform a specific business function, such as managing user accounts, processing payments, or handling product catalogs. In an enterprise scenario, microservices work together to form a large and complex system, like a banking application that might include microservices for Accounts, Cards, Loans, and more.

Key Characteristics

- Independent Deployability: Each service is independently deployable.
- Business Domain-Oriented: Services are designed around specific business functions (e.g., Accounts, Payments).
- Decentralized Data: Each service manages its own data.
- Lightweight Communication: Services communicate using simple protocols like HTTP or REST.
- Parallel Development: Multiple teams can develop and deploy services in parallel.

Pros

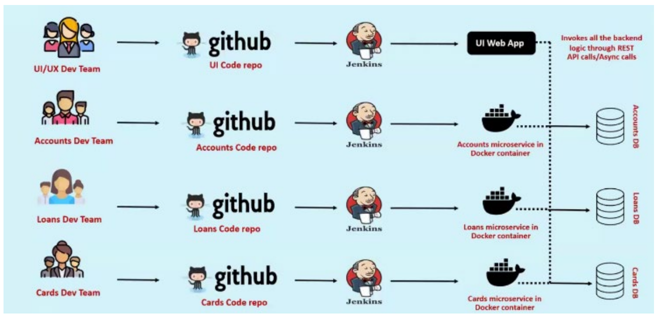
- Easy to develop and deploy: Smaller services are easier to build and maintain.
- Increased agility: Independent services can be updated without affecting the whole system.
- Scalable: Individual services can be scaled as needed.
- Business domain-driven: Aligns technical development with business needs.

Cons

- Complexity: Managing multiple services can be challenging.
- Infrastructure overhead: Requires tools for service management and communication.
- Security risks: More services mean more potential vulnerabilities.

When to Use Microservices Architecture?

- Large-scale applications where multiple teams need to work independently on different parts of the system.
- Applications that require high scalability and independence between business functions.
- Organizations that need to be agile and want to release features frequently.
- Systems with complex business logic that can be broken down into smaller, self-contained services.



Architectural Pattern

An Architectural Pattern is a general and reusable solution to a recurring design problem in software architecture. Unlike architectural styles, which define the overall structure of a system, architectural patterns provide specific guidance on how to implement functionalities within that structure.

Architectural patterns help in **organizing components, defining interactions, and improving** maintainability by ensuring scalability, flexibility, and separation of concerns. These patterns have a significant impact on the codebase, influencing either:

- Horizontally: how to structure components within a layer.
- Vertically: how a request moves through different layers in the architecture.

Examples of Architectural Patterns

1) Three-Tier Architecture

Definition: The Three-Tier Pattern organizes an application into three logical layers, each with a distinct responsibility. This separation improves maintainability and scalability.

Components:

- Presentation Tier: Handles user interface and interactions (e.g., web page, mobile UI).
- Business Logic Tier: Processes business rules, workflows, and computations.
- Data Tier: Manages databases and data storage.

Example:

A web-based e-commerce system where the front-end UI (React/Angular) interacts with the business logic (Java, .NET, Node.js) and retrieves data from a database (MySQL, PostgreSQL).

2. Microkernel (Plug-in Architecture)

Definition: The Microkernel Pattern separates the core system (microkernel) from specialized plug-ins or extensions. The core provides essential functionality, while plug-ins add features dynamically.

Components:

- Core System (Microkernel): Manages fundamental operations.
- Plug-ins: Extend functionality without modifying the core system.

Example:

Operating Systems like Linux or Windows, where users can install drivers and applications as plug-ins.
Integrated Development Environments (IDEs) like Eclipse or Visual Studio, which support different programming languages and tools as plug-ins.

3. Event-Driven Architecture

Definition: In Event-Driven Architecture (EDA), components communicate asynchronously by producing and consuming events, making the system more scalable and decoupled.

Components:

- Event Producers: Generate events (e.g., user clicks a button, a file is uploaded).
- Event Consumers: React to events and execute specific tasks.
- Event Broker (Message Queue): Manages event flow (e.g., Kafka, RabbitMQ).

Example:

Stock Trading Systems where events (buy/sell orders) trigger actions across different systems.

IoT (Internet of Things) where sensors send data to cloud applications, triggering automated responses.

Microservices Architecture

Definition: Microservices Architecture breaks down an application into a collection of independent, loosely coupled services, each responsible for a specific business function. Services communicate through lightweight APIs (e.g., REST, gRPC, GraphQL).

Components:

- Individual Microservices: Each service handles a single business capability (e.g., user authentication, payments, notifications).
- API Gateway: Manages requests and routes them to appropriate microservices.
- Service Discovery & Load Balancing: Ensures availability and performance (e.g., Kubernetes, Consul).

Example:

Netflix, where different microservices handle user profiles, recommendations, video streaming, and billing independently.

E-commerce platforms like Amazon, where microservices handle orders, payments, shipping, and customer reviews separately.

Design Patterns

A Design Pattern is a reusable solution to a common problem that occurs within a particular context in software design. Unlike Architectural Patterns, which influence the entire system's structure, Design Patterns are more localized and impact specific sections of the codebase.

They help improve code organization, reusability, maintainability, and scalability by providing standard approaches to solving common software problems.

Session - 04

How to Build Microservices: Overcoming Traditional Challenges with Spring Boot

Challenges with Traditional Approaches:

- Monolithic Deployment: Traditional deployment involves bundling the entire application (e.g., WAR or EAR files) and deploying it to a web or application server like Tomcat or WildFly. This is fine for monolithic applications, but microservices require more flexible, independent deployment.
- Scalability: Microservices often need to be deployed, scaled, and maintained independently. Packaging and deploying each microservice using traditional methods would be time-consuming and impractical.

Solution: Spring Boot for Microservices

- The key to overcoming these challenges is Spring Boot, a Java-based framework that simplifies the development of microservices.

How Spring Boot Helps:

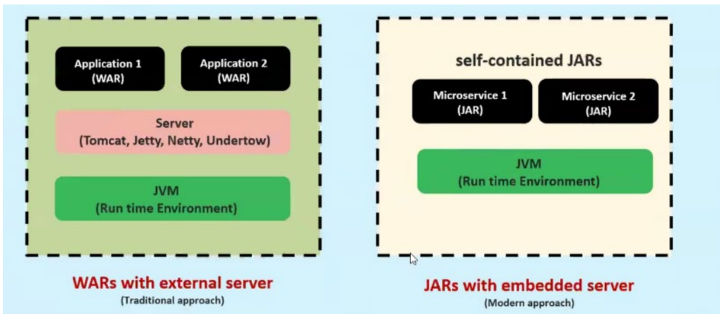
- Standalone Applications: Spring Boot packages each microservice as a self-contained JAR file, eliminating the need for an external application server.
- Embedded Servers: Microservices come with embedded servers (Tomcat/Jetty), making them independent and easy to deploy.
- Microservice Features:
  - REST APIs with Spring MVC.
  - Service Discovery using Eureka.
  - Configuration Management with Spring Cloud Config.
  - Distributed Tracing using Spring Cloud Sleuth.
- Cloud-Native: Spring Boot microservices are cloud-ready and work well with Docker and Kubernetes for containerization.

Why Spring Boot for Microservices?

Spring Boot simplifies microservice development by enabling self-contained, executable JARs, eliminating the need for external servers.

Key Benefits:

- **Built-in Features & Integrations** – Auto-configuration, dependency injection, and cloud support.
- **Embedded Servers** – Comes with Tomcat, Jetty, or Undertow, removing the need for separate installations.
- **Production-Ready** – Provides metrics, health checks, and externalized configurations out of the box.
- **Quick Setup with Starters** – Pre-configured starter dependencies for databases, queues, etc.
- **Cloud-Native Support** – Works seamlessly with Kubernetes, containers, and cloud providers for easy deployment.


Application 1 (WAR)
Application 2 (WAR)
Server (Tomcat, Jetty, Netty, Undertow)
JVM (Run time Environment)
WARs with external server
(Traditional approach)
self-contained JARs
Microservice 1 (JAR)
Microservice 2 (JAR)
JVM (Run time Environment)
JARs with embedded server
(Modern approach)

Implementing REST Services

REST (Representational state transfer) services are one of the most often encountered ways to implement communication between two web apps. REST offers access to functionality the server exposes through endpoints a client can call.

Below are the different use cases where REST services are being used most frequently these days,

REST (Representational State Transfer) is a widely used architectural style for enabling communication between web applications. REST services expose business functionalities through **endpoints** that clients can call using standard **HTTP methods**.

CRUD Operations with HTTP Methods

REST services primarily support CRUD (Create, Read, Update, Delete) operations for managing data in a storage system. The following HTTP methods are used for these operations:

- Create → POST (Adds new resources)
- Read → GET (Retrieves resources)
- Update → PUT/PATCH (Modifies existing resources)
- Delete → DELETE (Removes resources)

For industrial-level implementations, the **CQRS (Command Query Responsibility Segregation)** design pattern is often used to separate read and write operations, improving scalability and maintainability.

Session – 07

What is Cloud Computing?

Cloud computing is the delivery of computing services—such as servers, storage, databases, networking, software, and analytics—over the internet ("the cloud") to offer faster innovation, flexible resources, and economies of scale.

Key Features of Cloud Computing

On-demand self-service: Users can provision computing resources (e.g., storage, servers) automatically without human intervention.

Example: AWS users can launch virtual machines anytime without needing IT staff approval.

Broad network access: Services are accessible over the internet via multiple devices like laptops, mobiles, and tablets.

Example: Google Drive allows users to access their files from any internet-connected device.

Resource pooling: Cloud providers allocate and share computing resources among multiple customers dynamically.

Example: Microsoft Azure uses multi-tenancy to serve multiple companies from the same hardware securely.

Rapid elasticity: Resources can be scaled up or down based on demand in real-time.

Example: E-commerce platforms like Amazon scale their servers during Black Friday sales to handle high traffic.

Measured service: Cloud usage is tracked, monitored, and billed based on actual consumption (pay-as-you-go model).

Example: Google Cloud charges users only for the storage and processing power they use.

Benefits of Cloud Computing

Cost-efficient

Reduces capital investment in hardware and maintenance costs.

Example: Startups use AWS instead of buying expensive servers.

Scalable and flexible

Businesses can scale their resources up/down as per demand.

Example: Spotify scales its infrastructure when launching in new countries.

Enhanced collaboration

Teams can work together from different locations with shared cloud resources.

Example: Google Docs allows multiple users to edit a document in real-time.

Automatic updates and maintenance

Cloud providers manage updates and security patches, reducing IT workload.

Example: Microsoft 365 receives automatic software updates without user intervention.

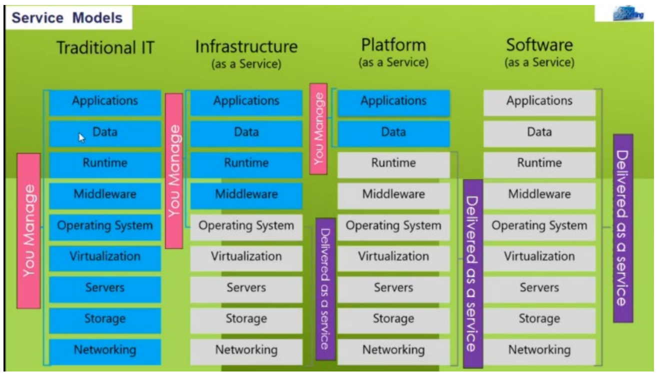
Disaster recovery and backup

Cloud providers offer backup and recovery solutions to prevent data loss.

Example: Dropbox provides automatic cloud backups for businesses.

Types of Cloud Services Model

Cloud service models define how cloud resources are provided to users. The three primary models are:



1. **IaaS (Infrastructure as a Service)**

Provides virtualized computing resources over the internet, such as servers, storage, and networking.

Example: AWS EC2 allows businesses to rent virtual machines instead of buying physical servers.

2. **PaaS (Platform as a Service)**

Offers a platform for developers to build, test, and deploy applications without managing infrastructure.

Example: Google App Engine enables developers to build and deploy web applications without worrying about the underlying servers.

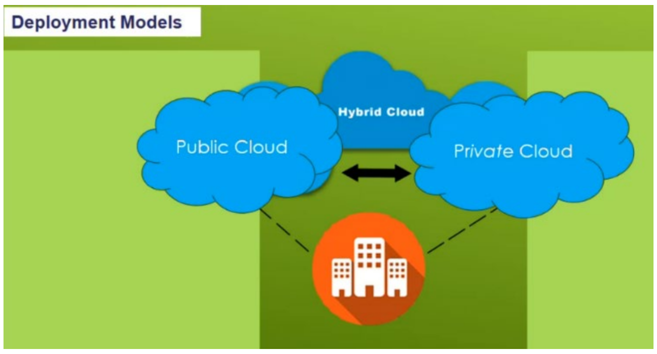
3. **SaaS (Software as a Service)**

Delivers software applications over the internet on a subscription basis.

Example: Gmail and Microsoft Office 365 provide email and productivity tools via the cloud.

Deployment Models of Cloud Computing

Deployment models define how cloud resources are managed, accessed, and shared across organizations. The four main models are:



1. **Public Cloud**

Cloud services are offered over the internet and shared among multiple organizations.

Example: Google Cloud and AWS provide cloud services to the public.

2. **Private Cloud**

Cloud infrastructure is dedicated to a single organization, ensuring higher security and control.

Example: Banks like JPMorgan Chase use private clouds for sensitive financial data processing.

3. **Hybrid Cloud**

Combines public and private cloud solutions, allowing businesses to use both for different purposes.

Example: A hospital might store patient records in a private cloud while running non-sensitive applications on a public cloud.

4. **Community Cloud**

Shared infrastructure for organizations with common interests or compliance requirements.

Example: Government agencies within the same country sharing a cloud for regulatory compliance.

Cloud Migration Design Patterns

Cloud migration design patterns define strategies for moving applications from on-premises infrastructure to the cloud. The choice of pattern depends on business goals, application complexity, and cloud benefits.

1. **Lift & Shift (Rehosting)**

Definition:

- Moving applications to the cloud with minimal or no modifications.
- Often used for quick migration without changing application architecture.

Example:

- A company moving its on-premise web servers to AWS EC2 instances without modifying the application.
- *Netflix initially migrated to AWS using Lift & Shift before optimizing further.*

Best For:

- Legacy applications
- Quick migration with minimal cost

- Reducing data center dependency

2. **Rearchitect**

Definition:

- Redesigning an application to leverage cloud-native features such as microservices, serverless, and containers.

Example:

- A monolithic e-commerce platform breaking into microservices and deploying on AWS Lambda & Kubernetes for scalability.
- *Airbnb rearchitected its infrastructure to improve resilience and handle high traffic.*

Best For:

- Improving performance and scalability
- Long-term cloud optimization
- Adopting cloud-native features

3. **Rework (Re-platforming)**

Definition:

- Making slight modifications to an application to take advantage of cloud optimizations without full redesign.

Example:

- Moving a traditional database from on-premises to a managed cloud database like Amazon RDS while keeping the application code the same.
- *Spotify reworked its infrastructure by migrating databases to Google Cloud for better performance.*

Best For:

- Applications needing slight optimizations
- Balancing cost and cloud benefits
- Reducing maintenance effort

4. **Refactor**

Definition:

- Modifying application code and architecture to fully utilize cloud benefits such as serverless computing and autoscaling.

Example:

- Converting a traditional API running on a virtual machine to a serverless function using AWS Lambda.
- *Capital One refactored its banking applications to run on cloud-native architectures for cost savings and agility.*

Best For:

- Applications needing high scalability and resilience
- Optimizing cost and performance
- Enabling DevOps and automation

Pattern	**Changes to Application**	**Cloud Benefits**	**Complexity**	**Example Use Case**
Lift & Shift	None or minimal	Low	Low	Moving VMs to AWS EC2
Rearchitect	Full redesign	High	High	Monolith to microservices
Rework	Some modifications	Medium	Medium	On-prem database to cloud database
Refactor	Code & architecture changes	Very High	High	Traditional API to serverless

What is Serverless Computing?

Serverless computing is a cloud execution model where developers build applications without managing servers. The cloud provider automatically handles infrastructure setup, scaling, and maintenance.

Key Characteristics:

- **Event-driven:** Resources are only used when a function is triggered.
- **Auto-scaling:** The cloud provider automatically scales resources up or down.
- **No server management:** Developers focus on writing code, while the provider manages infrastructure.
- **Pay-per-use:** Billing is based on actual execution time instead of pre-allocated resources.

Example:

- **AWS Lambda**: Runs a function only when triggered by an event, such as a file upload to S3.
- **Google Cloud Functions**: Executes code in response to HTTP requests or database changes.
- **Azure Functions**: Used for automating workflows, such as processing IoT sensor data.

Best For:

- Event-driven applications (e.g., chatbots, notifications)
- Real-time data processing (e.g., image recognition, analytics)
- Microservices-based architectures

While **overlapping with PaaS**, serverless computing is **fully event-driven**, meaning resources are only used when needed, making it cost-efficient and highly scalable.

Why Businesses are Moving to the Cloud?

Businesses are migrating to the cloud for scalability, cost efficiency, and security. Key drivers include:

1. **Software and Hardware Refresh Cycle**

- On-premises infrastructure requires periodic upgrades, which are costly and time-consuming.
- Cloud providers handle hardware and software updates automatically.
- **Example:** A company moving from aging physical servers to AWS to avoid costly replacements.

2. **Capacity Requirement**

- Businesses need scalable resources to handle fluctuating workloads.
- Cloud offers **on-demand scalability** without over-provisioning.
- **Example:** An e-commerce platform scaling its servers during Black Friday sales.

3. **Security**

- Cloud providers offer **advanced security** features like encryption, access controls, and automated threat detection.
- Cloud providers comply with **industry security standards** (ISO, SOC 2, GDPR).
- **Example:** A healthcare organization using Azure to secure patient data under HIPAA compliance.

4. **End of Life Events**

- Legacy systems become obsolete, requiring **migration to modern solutions**.
- Cloud offers **continuous innovation** with updated services.

- **Example:** A company moving from Windows Server 2012 (end of life) to AWS cloud-based services.

5. **Compliance**

- Cloud providers offer **built-in regulatory compliance** for industries like finance, healthcare, and government.
- Reduces the burden of maintaining compliance manually.
- **Example:** A financial institution using Google Cloud for GDPR and PCI DSS compliance.

6. **Business Acquisitions**

- Mergers require **integrating IT systems** quickly and efficiently.
- Cloud enables **seamless data sharing** across newly acquired entities.
- **Example:** A global enterprise consolidating IT operations after acquiring multiple businesses using hybrid cloud.

Containerization

What is Containerization?

Containerization is a lightweight virtualization technology that allows applications to be packaged with their dependencies and run consistently across different environments. It ensures that an application runs the same way, whether in development, testing, or production.

Why Use Containerization?

- **Portability** – Run anywhere (Local, Cloud, On-Premise).
- **Lightweight** – Uses fewer resources than VMs.
- **Fast Deployment** – Spin up apps in seconds.
- **Scalability** – Easily scale services up or down.
- **Consistency** – Works identically across environments.

How Does It Work?

1. A container image is created with everything an application needs (code, libraries, dependencies).
2. The image is deployed as a container instance using a container runtime (e.g., Docker).
3. Containers can be orchestrated using tools like Kubernetes for scaling and automation.

Popular Containerization Tools

- **Docker** – Most widely used container platform
- **Kubernetes** – Manages & orchestrates containers at scale
- **Podman** – Alternative to Docker, rootless containers
- **OpenShift** – Enterprise Kubernetes platform

How to Containerize an Application?

Containerizing an application involves **packaging** it with all its dependencies into a **container image** so it can run consistently across different environments.

Build a Container Image (Before Running a Container)

The first step in containerization is to **create a container image**. This image contains everything the application needs to run, such as:

- Application code
- Libraries & dependencies
- Configuration files
- Runtime environment (e.g., JDK for Java apps)
- OS-level dependencies

Once built, this image can be used to create **containers**, which are **instances** of the image running in an isolated environment.

What is a Container Image?

A **container image** is a **lightweight, standalone, and executable package** that includes everything needed to run an application. It is **immutable**, meaning it does not change after creation.

Key Characteristics:

- **Read-Only**: The base image remains unchanged; containers run from it.
- **Layered Structure**: Built using multiple layers (base OS, dependencies, application code).
- **Portable**: Can run across different machines, cloud platforms, or Kubernetes clusters.
- **Version-Controlled**: Stored and managed in container registries like Docker Hub, AWS ECR, or Azure Container Registry.

Purpose of Building an Image

Why Do We Need to Build an Image?

- **Portability** → Run the same application across different environments.
- **Consistency** → Avoid "works on my machine" problems.
- **Scalability** → Easily deploy across multiple servers or cloud providers.
- **Security** → Isolate applications to prevent conflicts between dependencies.
- **Efficiency** → Lightweight compared to virtual machines, reducing resource usage.

Methods for Building a Container Image

There are multiple ways to build a container image for an application. Below are the most common methods:

1. **Using a Dockerfile (Traditional Approach)**

A **Dockerfile** is a script that defines how to build a container image step by step.

A script that defines how to manually build an image step-by-step using Docker commands. Requires Docker installed.

Example: Containerizing a Spring Boot App Using a Dockerfile

1. **Create a Dockerfile** in the root directory of your project.
2. **Define the base image** and specify application dependencies.
3. **Copy application files** and configure how the container should run.

2. **Using JIB (Java Image Builder - Alternative for Spring Boot Apps)**

JIB is a **Google tool** that builds **optimized** container images without requiring Docker or a Dockerfile. It directly creates the image and pushes it to a registry.

A tool that builds optimized container images directly from Maven or Gradle without requiring Docker. Best for Java apps.

How to Use JIB?

1. Add JIB to Your Maven pom.xml
2. Build and Push the Image

Advantages of JIB:

- **Faster builds** (avoids full rebuilds).
- **No need for Docker daemon**.
- **Optimized image layers** for caching and performance.

3. **Using Spring Boot with Built-In spring-boot:build-image (Spring Boot Buildpacks)**

Spring Boot provides an **out-of-the-box** way to build OCI-compliant images using **Buildpacks**.

A Spring Boot built-in feature that automates image creation using Cloud Native Buildpacks, without a Dockerfile.

How to Use Spring Boot to Build an Image?

1. Add the Spring Boot Plugin to pom.xml
2. Build the Image
3. Run the Container

Advantages of Spring Boot Buildpacks:

- No need for a **Dockerfile**.
- Automatically selects the best base image.
- Secure and optimized images.

Best Method to Choose

Method	Pros	Cons
Dockerfile	Full control over the image, works with any language	Manual maintenance required
JIB	No need for Docker, fast incremental builds	Java-only solution

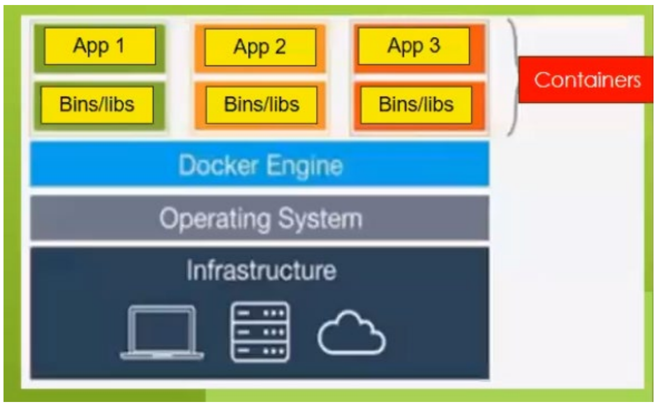
Spring Boot Buildpacks	No need for a Dockerfile, automated optimizations	Works best with Spring Boot apps

Conclusion

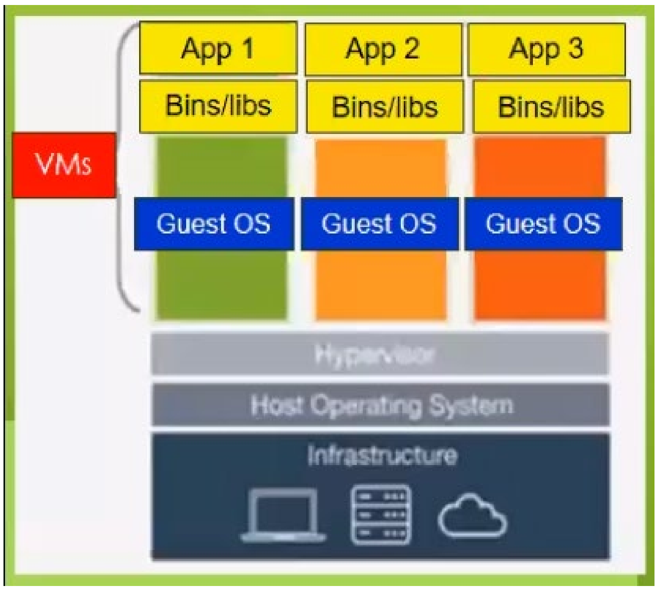
- Use **Dockerfile** for **full control** and **custom images**.
- Use **JIB** if you're working with **Java and want faster builds**.
- Use **Spring Boot Buildpacks** for **Spring Boot applications** with **minimal effort**.

Difference between Containerization and Virtualization

Containerization: A lightweight virtualization method that packages an application and its dependencies into a single container, sharing the host operating system's kernel while ensuring the application runs consistently across different environments.



Virtualization: A technology that creates virtual instances of hardware or operating systems, allowing multiple virtual machines (VMs), each with its own OS, to run on a single physical machine.



Feature	Containerization	Virtualization
Definition	Lightweight, portable application containers.	Emulates entire machines with OS and hardware.
Abstraction	Abstracts at the OS level (shares OS kernel).	Abstracts at the hardware level.
Size	Small and fast (MBs).	Large and slow (GBs).
Startup Time	Very fast (seconds).	Slower (minutes).
Resource Usage	Efficient, shares host OS resources.	Heavy, each VM runs its own OS.
Isolation	Process-level isolation.	Strong isolation with separate OS instances.
Examples	Docker, Kubernetes.	VMware, VirtualBox, Hyper-V.
Best For	Microservices, scalable apps, CI/CD pipelines.	Running different OS types, legacy systems.

Docker

Build, Ship, Run, Any App Anywhere:
This phrase summarizes Docker's main purpose:

- **Build**: Create lightweight, portable containers that include everything needed to run an application (code, libraries, dependencies).
- **Ship**: Easily share and distribute containers across environments (development, testing, production) without compatibility issues.
- **Run**: Execute containers consistently on any system with Docker installed, whether it's a local machine, a cloud server, or a hybrid environment.
- **Any App Anywhere**: Docker ensures applications run the same way regardless of the underlying operating system or infrastructure, providing flexibility and scalability.

Key Features:

- Improved, user-friendly Linux container technology.
- Easy, human-readable container images built using Dockerfiles (recipes).
- Provides isolation of networking, memory, CPU, and file systems.
- Container images can be easily shared and extended.
- Uses a layered file system with *copy-on-write* for efficiency.
- Lightweight virtualization technology, faster than traditional VMs.
- Supports Linux, Windows, and Mac OS X.

Docker Image:

Contains everything needed to run software:

- Code, runtime (e.g., JVM), drivers, tools, scripts, libraries, and deployments.

Layering in Docker Image

Docker Image Layering:

- Docker images are built in layers, where each layer represents an instruction in the Dockerfile (e.g., installing software, copying files).
- Layers are stacked on top of each other to form the final image.

Copy-on-Write Mechanism:

- When a container runs, a thin *read-write layer* is added on top of the image's *read-only layers*.
- Changes made by the container (like file modifications) are stored in this top layer, while the base layers remain unchanged.

Benefits of Layering:

- **Reusability**: Existing layers can be reused across multiple images, saving time and storage.
- **Efficiency**: Only the layers that change need to be updated, making builds faster.
- **Version Control**: Each layer can be cached, allowing easy rollbacks and tracking of changes.

Example:

- Base Layer: OS (Ubuntu)
- Next Layer: Install Java
- Next Layer: Add application code
- Top Layer: Configurations and scripts

How Docker Image Layering Helps Control Image Size

Docker uses a layered architecture and the copy-on-write strategy to prevent the Docker image size from increasing unnecessarily when modifications are made.

Key Concepts:

1. **Layered File System**:

- Each instruction in a Dockerfile creates a new layer.
- Layers are immutable (cannot be changed once created).

- All layers, except the top one, are read-only.

2. **Copy-on-Write (COW)**:

- When you modify a file in a running container, Docker doesn't modify the original file.
- Instead, Docker creates a copy of the file in the top **read-write layer** and applies changes there.
- The base image remains unchanged, saving space.

Cloud Native Development Best Practice

What is Cloud Native?

Cloud Native refers to designing, building, and running applications that fully leverage the benefits of cloud computing. These applications are optimized for scalability, flexibility, and resilience in cloud environments such as AWS, Azure, and Google Cloud.

Key Characteristics of Cloud Native Applications:

- **Microservices Architecture**: Applications are broken into small, independent services that can be developed, deployed, and scaled separately.
- **Containerized Deployment**: Use of containers (like Docker) for consistent and portable deployment across cloud environments.
- **Dynamic Orchestration**: Automated management of containers using tools like Kubernetes for scaling, load balancing, and self-healing.
- **Scalability**: Ability to scale services up or down based on demand, ensuring performance without over-provisioning resources.
- **Resilience**: Built-in fault tolerance and self-healing capabilities to ensure high availability even if parts of the system fail.
- **DevOps Practices**: Integration of continuous integration and continuous delivery (CI/CD) pipelines for frequent and reliable updates.

Benefits of Cloud Native Applications:

- **Faster Development and Deployment**: Automating development workflows and using lightweight containers accelerates delivery.
- **Cost Efficiency**: Pay-as-you-go cloud models reduce infrastructure costs.
- **Improved Performance**: Applications scale automatically with demand.
- **Flexibility**: Easily deployable across multiple cloud providers.
- **High Availability**: Designed for reliability with minimal downtime.

Cloud Native Applications vs Traditional Applications

Feature	Cloud Native Applications	Traditional Applications
Architecture	Microservices (modular, independent services)	Monolithic (single large codebase)
Deployment	Containerized (Docker, Kubernetes)	Deployed on physical servers or VMs
Scalability	Dynamic, automatic scaling based on demand	Manual scaling, often resource-intensive
Development	Agile, CI/CD pipelines for frequent releases	Waterfall, slower release cycles
Resource Usage	Efficient, pay-as-you-go cloud resource utilization	Fixed resources, often underutilized
Resilience	Built-in fault tolerance and self-healing	Single points of failure, harder to recover
Portability	Highly portable across cloud providers	Tied to specific hardware or infrastructure
Cost	Pay-as-you-go, operational cost model	High upfront cost, fixed infrastructure investment

Management	Automated through orchestration tools (Kubernetes)	Manual configuration and management
Updates	Continuous integration and delivery (CI/CD)	Scheduled, less frequent updates
Flexibility	Supports multi-cloud and hybrid environments	Limited flexibility, often single environment

Key Differences:

- **Cloud Native**: Optimized for cloud environments with high scalability, agility, and automation.
- **Traditional Apps**: Built for on-premise or fixed infrastructure with manual scaling and slower updates.

Coding Standards for Cloud Native Applications – The 15-Factor Principles

The **15-Factor App** methodology extends the original Twelve-Factor App principles to address modern cloud-native application requirements.

1. **Codebase**: A cloud-native application should have a **single codebase** that is version-controlled, typically using tools like Git. Multiple deployments of the application (e.g., dev, staging, production) should refer to the same codebase, with environment-specific configurations handled separately.

   Coding Standard:

- Keep a **single codebase** in a version control system (e.g., Git).
- Use branching strategies (e.g., Gitflow, trunk-based development) to manage different stages of development.
- Ensure that the codebase is **modular and maintainable**, avoiding duplication.

2. **Dependencies**: Cloud-native applications should explicitly declare their dependencies to ensure consistent environments and avoid conflicts. Dependencies must be isolated to prevent unnecessary reliance on the underlying system.

   Coding Standard:

- Use dependency management tools (e.g., **npm** for Node.js, **pip** for Python) to explicitly list and install dependencies.
- Isolate dependencies within the app, using virtual environments or containers.
- Avoid relying on system-wide libraries to prevent version conflicts.

3. **Config**: Environment-specific configuration (e.g., API keys, database URLs) should be stored in environment variables. This separates the configuration from the code, allowing the same codebase to run across different environments (e.g., local, staging, production).
   Coding Standard:

- Store sensitive data and environment-specific configuration in **environment variables**.
- Use tools like **dotenv** (for local development) and cloud configuration services for secure storage.
- Keep sensitive configurations, like passwords and API keys, secure by using encryption.

4. **Backing Services**: Backing services such as databases, message queues, and caching should be treated as attached resources. The application should not assume anything about the internal configuration or implementation of these services.
   Coding Standard:

- Treat external services like databases or caching as **plug-and-play** components, connected via environment variables.
- Ensure that the app can easily switch backing services without needing to change the code.

5. **Build, Release, Run**: The build, release, and run stages should be kept separate. The build stage is responsible for compiling code and preparing assets, the release stage for preparing configurations, and the run stage for execution.
   Coding Standard:

- Clearly separate build, release, and run processes to avoid confusion and accidental changes.
- Automate the deployment pipeline using CI/CD tools, ensuring that each stage is repeatable and automated.

6. **Processes**: Applications should be designed to be **stateless**, meaning no internal state is stored in the application itself. Any required state should be stored in external backing services (e.g., databases or caches).
   Coding Standard:

- Ensure that your application does not rely on in-memory state between requests.
- Use external systems for storing persistent state (e.g., a distributed database or object storage).

7. **Port Binding**: Cloud-native apps should **self-contain** and expose services via port binding. The application should not rely on an external web server (e.g., Apache or Nginx) to handle requests.
   Coding Standard:

- The app should **bind to a port** on startup and handle HTTP requests on its own (e.g., Express.js in Node.js, Flask in Python).
- Use environment variables to configure port binding for different environments.

8. **Concurrency**: Applications should scale by adding more **processes** rather than relying on vertically scaling the app (e.g., adding more memory or CPU to a single machine). The process model enables horizontal scaling in cloud environments.
   Coding Standard:

- Use process-based models (e.g., worker processes, application containers) for scaling.
- Ensure that each process is independently scalable and handles specific tasks.

9. **Disposability**: Applications should be **disposable**, meaning they can start and stop quickly. This is crucial for the dynamic scaling and fault tolerance inherent in cloud-native environments.
   Coding Standard:

- Ensure that your application starts up and shuts down **quickly**.
- Handle graceful shutdowns by closing connections, completing transactions, and cleaning up resources.

10. **Dev/Prod Parity**: The development, staging, and production environments should be as similar as possible to avoid **environment-related bugs** and ensure consistency across stages.
    Coding Standard:

- Use containerization (e.g., Docker) to replicate the production environment locally.
- Keep production, staging, and dev environments as close to identical as possible, especially regarding the services used.

11. **Logs**: Logs should be treated as **event streams** that can be consumed by external log aggregators. Logs are crucial for monitoring, debugging, and auditing.

Coding Standard:

- Emit logs as **structured data** (e.g., JSON) that can be easily consumed by log management tools.
- Avoid using logs for user-facing messages and focus on system state, errors, and events.

12. **Admin Processes**: Administrative or management tasks (e.g., database migrations, backup jobs) should be executed as **one-off processes** that are executed separately from the main application processes.

13. **API First**: Applications should be designed with an **API-first approach**, ensuring that all functionalities are exposed via APIs. This enables seamless integration between services and supports microservices architecture.
    Coding Standard:

- Design and document APIs before implementing them using tools like **Swagger/OpenAPI**.
- Ensure APIs follow RESTful principles or use gRPC for high performance.
- Maintain backward compatibility when updating APIs.

14. **Telemetry**: Implementing **telemetry** ensures that applications are monitored in real-time for performance, errors, and health metrics. This enables better observability and proactive issue resolution.
    Coding Standard:

- Integrate telemetry tools (e.g., **Prometheus**, **Grafana**, **ELK Stack**) to capture logs, metrics, and traces.
- Include structured logging, distributed tracing, and error reporting in your codebase.
- Use dashboards and alerts to monitor app health continuously.

15. **Authentication and Authorization**: Robust **authentication and authorization** mechanisms are essential for securing cloud-native applications, ensuring that only authorized users and services can access resources.
    Coding Standard:

- Implement secure authentication protocols (e.g., OAuth 2.0, OpenID Connect) and manage user identities securely.
- Use JWTs (JSON Web Tokens) for stateless authentication between microservices.
- Regularly audit and update security policies and access controls.