Unit 4

Service Oriented Architecture(SOA)

Definition:

- SOA is a software development approach where applications are built using modular **services**, each providing a discrete business capability. Services can communicate across platforms and languages, allowing **reuse** in multiple systems or combining independent services to perform complex tasks

Example:

- Instead of implementing authentication separately for multiple systems, a single authentication service can be created and reused. Similarly, patient registration in a healthcare organization can be managed by one shared service across multiple systems.

Benefits of SOA

- Faster Time to Market
  - Services can be reused across different business processes, saving development time and costs.
  - Applications can be assembled quickly without writing all code from scratch.
- Efficient Maintenance
  - Small, independent services are easier to create, update, and debug than large monolithic code blocks.
  - Modifying a service does not affect the overall business process, reducing risk.
- Greater Adaptability
  - SOA allows applications to adapt efficiently to technological changes.
  - Legacy systems can be integrated with newer applications cost-effectively, e.g., older electronic health record systems reused in cloud-based healthcare applications.

Basic principles of service-oriented architecture

- Although there are no strict standards for implementing SOA, the following principles are common across most implementations:
- **Interoperability**
  - Each service provides a **description document** detailing its functionality and usage terms.
  - Services can be used by **any client system**, regardless of platform or programming language (e.g., C# and Python services can coexist).
  - **Changes in one service do not affect others**, as services interact indirectly.
- **Loose Coupling**
  - Services should have **minimal dependency** on external resources (data models, systems).
  - Services should be **stateless**, not retaining information from past sessions or transactions.
  - This ensures that **modifying a service doesn't disrupt clients or other services**.
- **Abstraction**
  - Service users **do not need to know internal code logic**.
  - Services appear as a **"black box"**; users interact via **service contracts** and documentation specifying how to use them.
- **Granularity**
  - Services should have an **appropriate scope**, ideally encapsulating a **single business function**.
  - Multiple services can be combined to create **composite services** for complex operations.

Web Services in Cloud Computing

- **Web Services** in cloud computing are **standardized software components or APIs** that enable **communication and data exchange between different applications over the Internet**.
- They act as a **bridge between client applications and cloud resources**, allowing clients to request and receive services **regardless of platform, language, or operating system**.
- **In simple terms:**
  - A **web service** is a **software function accessible via web protocols (like HTTP)** that enables **interoperable machine-to-machine communication** over a network — especially over the **Internet or cloud infrastructure**.

- **Key characteristics:**
  - Platform independent (Java, .NET, Python, etc.)
  - Language neutral (XML, JSON data)
  - Uses open standards (HTTP, XML, JSON, WSDL, SOAP, REST)
  - Interoperable and reusable
- **Examples in Cloud Computing:**
  - Google Cloud Translate API (REST-based web service)
  - AWS Simple Queue Service (SQS) API (REST-based)
  - Legacy enterprise ERP web services (SOAP-based)

- **Types of Web Services**
- There are mainly **two types** of web services used in cloud computing:
  - **SOAP-based Web Services**
  - **RESTful Web Services**

SOAP

- **SOAP (Simple Object Access Protocol)** is a protocol-based messaging framework widely used for enabling communication between distributed applications in cloud computing environments.
- It is platform- and language-independent, and it uses XML as the message format to ensure interoperability across heterogeneous systems.
- SOAP follows a strict client–server model and is particularly suited for enterprise-level cloud applications that require high security, reliability, and well-defined contracts between services.
- A SOAP message consists of an XML envelope containing an optional header, a body with the actual data, and an optional fault element for error handling

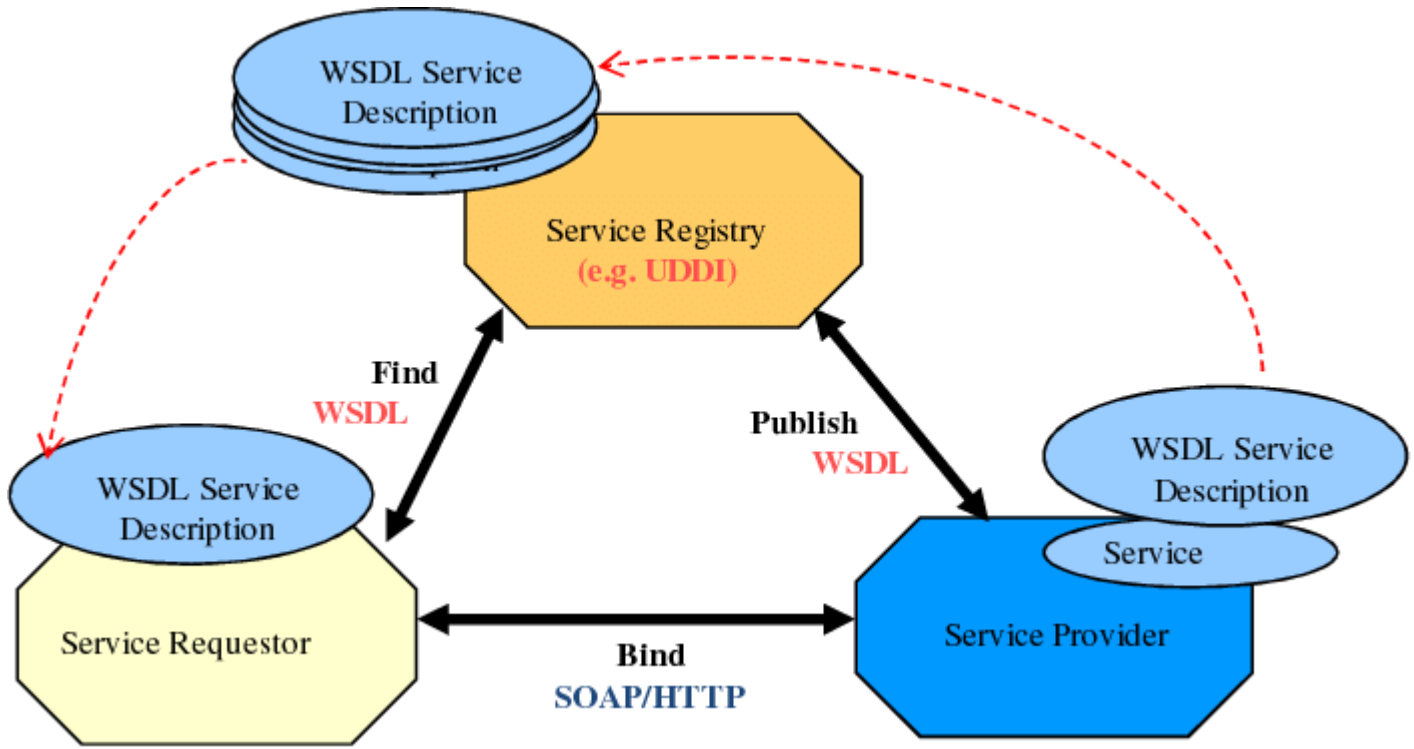
- **Key Features of SOAP**
  - **Protocol-based** (unlike REST which is just an architectural style)
  - **Language independent** (can be used with Java, .NET, Python, etc.)
  - **Platform independent** (runs on Windows, Linux, etc.)
  - **Uses XML** for message formatting (platform-neutral, human-readable)
  - **Extensible** (can be extended with security, transactions, routing)
  - Works over **multiple transport protocols**: HTTP, SMTP, JMS, TCP
  - Follows **client-server model**

- The SOAP protocol architecture includes multiple layers:
  - the service transport layer (which uses HTTP, SMTP, JMS, or TCP to transmit messages),
  - the XML messaging layer (which encodes and decodes messages),
  - the SOAP processing layer (which handles the envelope, header, and body rules),
  - the service description layer (which uses WSDL—Web Services Description Language—to define the service operations, data types, and protocols),
  - the service discovery layer (which uses UDDI—Universal Description, Discovery, and Integration—to publish and locate web services).

- **SOAP Architecture** defines the structured framework through which SOAP-based web services operate in cloud computing environments.
- It consists of several layers and components that work together to enable secure, reliable, and platform-independent communication between distributed systems.
- At the core is the **SOAP message**, which is an XML-based document composed of an **Envelope** (the root element identifying the message as SOAP), an optional **Header** (carrying meta-information such as authentication and routing data), a **Body** (containing the actual request or response data), and an optional **Fault** element (used for reporting errors).
- These messages are transmitted over a **transport protocol layer** such as HTTP, SMTP, JMS, or TCP, which is responsible for physically delivering the messages between client and server.
- Above the transport layer is the **XML messaging layer**, which handles the encoding and decoding of the XML data.

- The **SOAP processing layer** ensures proper interpretation of the envelope, header, and body according to SOAP standards.
- SOAP services are formally described using **WSDL (Web Services Description Language)** in the **service description layer**, which defines the service operations, input/output data types, and communication protocols, acting as a contract between the client and the service provider.
- Additionally, the **UDDI (Universal Description, Discovery, and Integration)** registry serves as the **service discovery layer**, where services can be published and searched by potential clients.
- Together, these layers create a well-defined architecture that supports interoperability, extensibility, and security, making SOAP suitable for complex enterprise and cloud-based applications that require strict message structure, guaranteed delivery, and standardized service contracts

- The core components and interactions within a SOAP web service architecture include:
- **Service Provider:**
  - This is the application or system that offers a particular web service. It exposes its functionalities through a WSDL (Web Services Description Language) file.
- **Service Requester (Client):**
  - This is the application or system that wants to consume the web service provided by the service provider. It uses the WSDL to understand how to interact with the service.
- **Service Registry (Optional):**
  - A UDDI (Universal Description, Discovery and Integration) registry can be used to publish and discover web services. Service providers can register their services in the UDDI, and service requesters can search the UDDI to find suitable services.


WSDL Service Description
Service Registry (e.g. UDDI)
Find WSDL
Publish WSDL
WSDL Service Description
Service Requestor
WSDL Service Description
Service
Service Provider
Bind SOAP/HTTP

- SOAP Message Structure:
- SOAP messages are XML documents structured with a specific format:
- **Envelope:**
  - The root element of every SOAP message, defining the XML namespace and containing the entire message.
- **Header (Optional):**
  - Contains application-specific information that might need to be processed by intermediate SOAP nodes along the message path, such as authentication, transaction management, or routing information.
- **Body (Mandatory):**
  - Contains the actual payload of the message, including the data and the method to be invoked on the service provider.
- **Fault (Optional):**
  - A sub-element within the Body used to report errors that occur during the processing of the SOAP message.

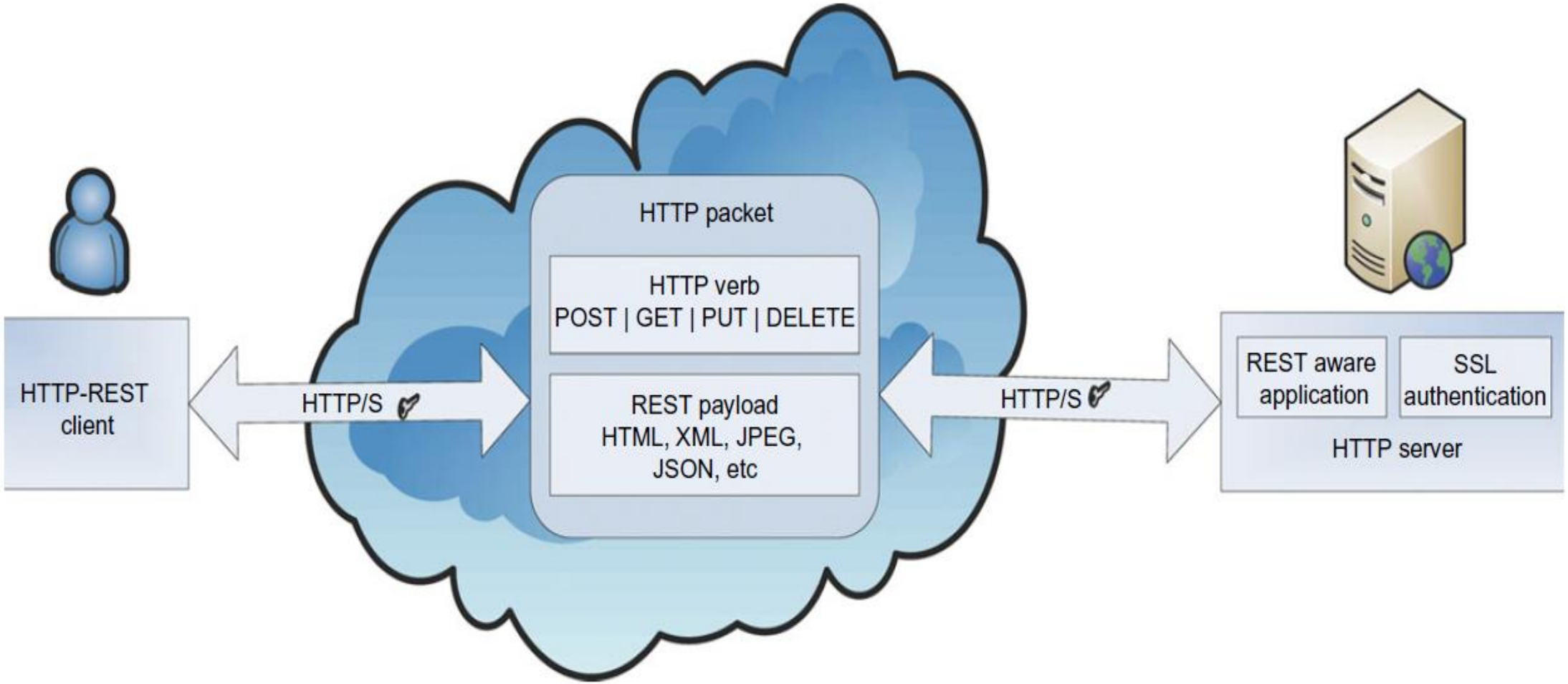
- SOAP Communication Flow:
  - The service requester formulates a SOAP request message, typically an XML document, based on the WSDL.
  - This SOAP message is sent to the service provider using a transport protocol like HTTP, HTTPS, or SMTP.
  - The service provider receives the SOAP message, processes the request, and performs the required operations.
  - The service provider then constructs a SOAP response message, also an XML document, containing the results of the operation.
  - This response is sent back to the service requester using the same transport protocol.

```
<SOAP-ENV:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <SOAP-ENV:Header>
  <authHeader xmlns="http://example.com/auth">
   <username>user123</username>
   <password>pass123</password>
  </authHeader>
 </SOAP-ENV:Header>
 <SOAP-ENV:Body>
  <getCustomerRequest xmlns="http://example.com/customers">
   <customerId>123</customerId>
  </getCustomerRequest>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Copy

REST (Representational State Transfer)

- REST is a **software architecture style** for **distributed systems**, especially **distributed hypermedia systems** like the World Wide Web.
- It has become popular among enterprises such as **Google, Amazon, Yahoo!**, and social networks like **Facebook and Twitter** due to its **simplicity** and ease of being **published and consumed by clients**.
- REST was introduced by **Roy Thomas Fielding**, one of the principal authors of the HTTP specification, in his **2000 doctoral dissertation**. It was developed alongside the **HTTP/1.1 protocol**.
- The REST architectural style is based on **four main principles**:
  - **Resource Identification through URIs**
  - **Uniform, Constrained Interface**
  - **Self-Descriptive Messages**
  - **Stateless Interactions**


HTTP packet
HTTP verb
POST | GET | PUT | DELETE
REST payload
HTML, XML, JPEG,
JSON, etc
HTTP-REST
client
HTTP/S
HTTP/S
REST aware
application
SSL
authentication
HTTP server

1. **Resource Identification through URIs:**

- The RESTful web service exposes a set of resources which identify targets of interaction with its clients.
- In REST, the key abstraction of information is a **resource**, which can represent anything that can be named, such as a document, image, user profile, or a temporal service.
- A resource is a conceptual mapping to a set of entities.
- Each resource is identified by a unique URI, usually expressed as a URL. This URI provides a global addressing space, enabling easy interaction between distributed components.
- Facilitates service discovery. URIs can be bookmarked, linked, and exchanged, making resources easily accessible.
- **Example:** A user object in a RESTful system could have a URI like https://example.com/users/123.
    - Here, 123 uniquely identifies the specific user resource.

2. Uniform, Constrained Interface:

- Interaction with RESTful web services is done via the HTTP standard, client/server cacheable protocol.
- Resources are manipulated using a fixed set of four CRUD (create, read, update, delete) verbs or operations: PUT, GET, POST, and DELETE.
- PUT creates a new resource, which can then be destroyed by using DELETE. GET retrieves the current state of a resource. POST transfers a new state onto a resource.
- Ensures that all clients interact with resources in a consistent, predictable way.
- Simplifies client-server communication and allows caching.
- Example:
  - GET /users/123 – retrieves details of user 123.
  - PUT /users/123 – updates information of user 123.
  - DELETE /users/123 – deletes the user resource

3. Self-Descriptive Message:

- A REST message includes enough information to describe how to process the message. This enables intermediaries to do more with the message without parsing the message contents.
- In REST, resources are decoupled from their representation so that their content can be accessed in a variety of standard formats (e.g., HTML, XML, MIME, plain text, PDF, JPEG, JSON, etc.).
- REST provides multiple/alternate representations of each resource.
- Metadata about the resource is available and can be used for various purposes, such as cache control, transmission error detection, authentication or authorization, and access control.

- **Example:**

  A GET /users/123 request may return JSON:

  ```
  {
    "id": 123,
    "name": "Alice",
    "email": "alice@example.com",
    "last_modified": "2025-09-21T10:00:00Z"
  }
  ```

  Metadata like last_modified and HTTP headers help with caching and concurrency control.

4. Stateless Interactions:

- The REST interactions are “stateless” in the sense that the meaning of a message does not depend on the state of the conversation.
  - Stateless meaning the server does not retain client context between requests. Each request contains all information needed to process it.
- Stateless communications improve
  - Scalability: Servers can handle more clients as they do not store session data.
  - Reliability: Easier to recover from partial failures, since each request is independent.
  - Visibility: Monitoring systems can interpret a request without needing prior context.
- However, stateless interactions may decrease network performance(increases network overhead) as repeated data may need to be sent in each request.
- Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction.

- RESTful web services provide a lightweight infrastructure where services can be built with minimal tools, making them inexpensive and easy to adopt.
- Developers can test these services directly from a web browser without creating specialized client-side software, reducing development effort.
- Being stateless, RESTful services support scalability and can handle a large number of clients efficiently through caching, clustering, and load balancing.

- RESTful web services can be considered an alternative to SOAP stack or “big web services”, because of their simplicity, lightweight nature, and integration with HTTP.
- Using URIs and hyperlinks, REST enables resource discovery without the need for centralized repositories, although efforts like the Web Application Description Language (WADL) have been introduced to describe RESTful services in XML and improve discoverability.
- Despite its advantages, REST is not a formal standard but rather an architectural style, and it faces challenges such as URI length limitations in GET requests.
- To help developers, Java frameworks like Restlet and JAX-RS (JSR-311) have emerged, providing APIs, annotations, and tools to map URIs and HTTP methods to Java objects while supporting multiple content types such as HTML, XML, and JSON.
- Jersey, the reference implementation of JAX-RS, further extends these capabilities, enabling flexible and scalable develop

- A practical example of RESTful web services in high-performance computing is the Amazon Simple Storage Service (S3) interface, which provides scalable internet-based data storage accessible anytime through simple web requests.
- In S3, the core entities are objects (data with metadata), stored in buckets—containers that organize storage, manage access control, and handle usage reporting.
- Each bucket and object is uniquely identified by a URI of the form https://s3.amazonaws.com/{bucket}/{object}.
- RESTful operations on these resources are performed using standard HTTP methods:
  - GET (retrieve bucket lists, object data, or metadata),
  - PUT (create buckets or upload objects),
  - DELETE (remove buckets or objects), and
  - HEAD (retrieve object metadata).
- Every operation involves sending a standard HTTP request (method, URI, headers, optional body) and receiving an HTTP response (status code, headers, optional body).
- Authentication is handled through an Authorization header, which includes the AWS Access Key ID, Secret Key, and a generated signature; requests are processed only if the signature matches.
- This simplicity, combined with REST's lightweight design, makes S3 easy to integrate into composite applications (mashups) where services like Flickr images can be overlaid on Google Maps.
- Thus, Amazon S3 demonstrates how REST principles enable scalable, secure, and flexible data storage and retrieval over the web

MicroServices

- Microservices is an **architectural style** where a large application is built as a collection of small, independent, and loosely coupled services.
- Each service focuses on a single business function, runs in its own process, and communicates with other services through lightweight protocols (mostly HTTP/REST or gRPC, sometimes messaging queues like Kafka or RabbitMQ).
- Unlike monolithic architecture, where all functionalities are bundled into one codebase, microservices **divide the system into modules** that can be developed, deployed, scaled, and maintained independently.

Key characteristics of microservices architecture:

1. Independent Services

- Each microservice is developed, deployed, and managed separately. This independence ensures that changes in one service (e.g., payment gateway) do not affect others (e.g., product catalog). It also allows different teams to work on different services simultaneously, improving productivity.

2. Single Responsibility (Business Capability Focused)

- Each microservice is designed around a specific business function or domain. For example, in an e-commerce system, separate microservices might handle *orders*, *inventory*, and *shipping*. This makes the system modular, easier to understand, and more maintainable

3. Decentralized Data Management

- Unlike monolithic systems where a single database is shared, each microservice manages its own database or data storage. This ensures loose coupling and gives flexibility to use the best-suited database (SQL, NoSQL, etc.) for the service. However, maintaining data consistency across services becomes a challenge and often requires patterns like *event sourcing* or *sagas*.

4. Technology Heterogeneity

- Teams can choose different programming languages, frameworks, or databases for different services. For example, one service could be written in Java with MySQL, while another uses Python with MongoDB. This flexibility allows teams to use the right tool for the job.

5. Decentralized Governance and DevOps Alignment

- Microservices align with DevOps and Agile principles. Each service can be continuously integrated, tested, and deployed without waiting for the whole system to be rebuilt. This accelerates innovation and reduces deployment risks.

6. Resilience and Fault Isolation

- If one microservice fails (e.g., recommendation engine), the entire system does not go down, unlike monolithic systems. Circuit breakers, retries, and failover mechanisms can be implemented to keep the system resilient.

7. Scalability

- Each microservice can be scaled independently based on demand. For instance, the "search" service in an e-commerce platform may need to handle millions of requests, while the "payment" service handles fewer. Independent scaling reduces costs and optimizes resource utilization.

8. API-Driven Communication

- Microservices communicate through lightweight APIs (typically REST, gRPC, or messaging systems like Kafka/RabbitMQ). This ensures interoperability and allows services to be deployed across multiple environments, including cloud and on-premises.

9. Automated Deployment and CI/CD

- Microservices are usually deployed via containers (e.g., Docker, Kubernetes), enabling automation, rapid delivery, and easier rollback in case of failures. CI/CD pipelines streamline the process of building, testing, and deploying each service.

10. Observability (Monitoring & Logging)

- Since microservices are distributed, effective monitoring, centralized logging, and tracing (using tools like Prometheus, ELK stack, or Jaeger) are critical to track performance, errors, and inter-service communication.

Application Architecture in Cloud

- Application architecture in the cloud refers to how applications are designed, structured, and deployed to leverage cloud computing services (IaaS, PaaS, SaaS). It defines the components, their interactions, and how resources (compute, storage, networking, databases, etc.) are managed in a scalable, secure, and cost-effective way.
- Unlike traditional on-premises systems, cloud applications leverage distributed resources, elastic scaling, and managed services provided by **cloud service providers (CSPs)** like AWS, Azure, and GCP.
- It ensures:
    - **Scalability** (handle sudden traffic spikes)
    - **Resilience** (recover from failures automatically)
    - **Security** (protect data & apps)
    - **Cost-effectiveness** (pay-as-you-go model)

Core Layers of Cloud Application Architecture

(a) Presentation Layer (Client-Side / Frontend)

- **Role**: User-facing interface (web browsers, mobile apps).
- **Cloud Integration**:
  - CDN (Content Delivery Network) for faster delivery (e.g., CloudFront, Azure CDN).
  - Load balancers distribute traffic (e.g., AWS ELB, Azure Load Balancer).
- **Examples**: React/Angular web apps, Android/iOS apps.

(b) Application Layer (Business Logic)

- **Role**: Handles core logic, computations, and workflows.
- **Cloud Integration**:
  - **Microservices** deployed in **containers** (Docker, Kubernetes).
  - **Serverless Functions** (AWS Lambda, Azure Functions) for event-driven tasks.
  - Auto-scaling groups to scale up/down based on demand.
- **Example**: A checkout service in an e-commerce platform.

c) Data Layer

- **Role**: Store and manage data.
- **Cloud Options**:
  - **Relational Databases**: AWS RDS, Azure SQL, Cloud SQL.
  - **NoSQL Databases**: DynamoDB, Cosmos DB, Firestore.
  - **Storage**:
    - Object storage (S3, Blob Storage) for images, videos, backups.
    - Block storage for VM disks.
  - **Caching**: Redis, Memcached for faster access.
- **Example**: Orders database + Product catalog in DynamoDB.

(d) Integration & Messaging Layer

- **Role**: Connects components for communication.
- **Cloud Options**:
  - **Message Queues**: SQS, RabbitMQ, Pub/Sub.
  - **API Gateway**: Secure access to microservices.
  - **Event Bus**: AWS EventBridge, Azure Event Grid.
- **Example**: Order placed → Notification service triggered → Payment service.

(e) Security & Identity Layer

- **Role**: Protect users, data, and applications.
- **Cloud Features**:
  - IAM (Identity and Access Management).
  - Authentication: OAuth2, SSO, Cognito, Azure AD.
  - Encryption: KMS (AWS), Key Vault (Azure).
  - WAF (Web Application Firewall) and DDoS protection.
- **Example**: Only admin users can update product details.

f) DevOps, Monitoring, and Management Layer

- **Role**: Continuous deployment, performance monitoring, cost management.
- **Cloud Tools**:
  - CI/CD: GitHub Actions, GitLab CI, AWS CodePipeline.
  - Monitoring: AWS CloudWatch, Azure Monitor, Prometheus + Grafana.
  - Infrastructure as Code: Terraform, CloudFormation.
- **Example**: Auto-deploying new microservice versions with zero downtime.

Cloud Application Architecture Models

- Cloud application architecture models describe the different ways applications can be designed, structured, and deployed in cloud environments. Each model has its own characteristics, benefits, and use cases depending on scalability requirements, performance goals, and complexity of the system.

1. Monolithic Architecture

- In a **monolithic cloud architecture**, the application is built as a single, unified codebase where all features—such as user interface, business logic, and data management—are tightly coupled.
- Although simple to develop and deploy initially, monolithic applications become difficult to scale and maintain as they grow larger.
- When deployed in the cloud, a monolithic application is often hosted on virtual machines or containers, and scaling requires replicating the entire system rather than individual components.
- This model is best suited for small applications or legacy systems being migrated to the cloud with minimal changes.

2. Microservices Architecture

- The **microservices model** breaks down applications into a collection of small, independent services that communicate through APIs.
- Each service is responsible for a specific functionality, such as payment processing, inventory management, or authentication, and can be developed, deployed, and scaled independently.
- In the cloud, microservices are commonly deployed in containers (Docker, Kubernetes) or serverless platforms, making them highly flexible and resilient.
- This model offers agility, fault isolation, and better scalability, but it introduces complexity in service orchestration, monitoring, and inter-service communication.
- Companies like Netflix and Amazon successfully use microservices for high-scale applications.

3. Serverless Architecture

- In a **serverless architecture** (also known as Function-as-a-Service, or FaaS), the cloud provider manages the infrastructure and automatically scales the execution of small functions in response to events.
- Developers only need to write code, while the cloud platform handles provisioning, scaling, and resource allocation.
- This model allows organizations to pay only for the actual execution time of code, making it cost-efficient.
- It is ideal for event-driven applications, such as image processing, IoT data collection, or chatbots.
- However, serverless architectures may face challenges like cold start latency and limited execution time for complex workflows.

4. Event-Driven Architecture

- The **event-driven model** is designed around the concept of events and reactions, where services or functions are triggered by specific events such as user actions, database updates, or system alerts.
- It enables applications to be loosely coupled, responsive, and scalable.
- Cloud providers support this model through tools like AWS EventBridge, Azure Event Grid, and Google Pub/Sub.
- Event-driven architectures are particularly effective for real-time systems, such as fraud detection, notification services, or e-commerce order processing.
- This model ensures high flexibility but requires careful management of event flows and monitoring to avoid bottlenecks.

5. Hybrid and Multi-Cloud Architecture

- In **hybrid architecture**, part of the application runs on on-premises infrastructure while other components are deployed in the cloud, enabling organizations to balance security, compliance, and scalability.
- **Multi-cloud architecture**, on the other hand, distributes application components across multiple cloud providers (e.g., AWS + Azure + GCP) to reduce vendor lock-in and improve resilience.
- These models are particularly relevant for enterprises dealing with sensitive data, regulatory requirements, or global-scale operations.
- While they offer flexibility and risk reduction, they also increase operational complexity in terms of integration, monitoring, and governance.