Name :- Deepthi mashetti

Day 1 :

Computer Architecture

Softwares :- 1. System software

2. Application Software

System Software:

- This acts as an interface between the system and the applications.
- It is the platform that allows the various application software to run on the system.
- System Software is generally developed in low-level languages. This is so that the interaction between the software and hardware can be simplified and made more compatible.
- Is working is more automated. Once a system is turned on, the system software starts working
- The system software are installed at the time of installing the operating system. A computer device cannot work without its presence.
- It is an independent software. Once this is installed the computer will work.
- Since a device cannot work without a system software, the user has to have it installed in their devices.
- Example: System Software includes Android, Mac Operating system, MS Windows, etc.

. Application Software :

- This is designed directly from the user perspective.
- These are independent applications which can be download and installed in the system.

- Each application has a specific purpose and thus is developed with high-level languages so that the purpose can be fulfilled.
- User action is required to start application software. These applications can only be work when the user commands the system to do so.
- They have minimum involvement in the processing and functioning of the computer device.
- The application software can be installed as and when the user requires them.
- This is a dependent software. Applications can only be downloaded when the operating system is installed.
- These are designed to be user interactive, thus the application software can be removed as and when required by the user.
- Examples of Application Software includes Word Processor, games, media player,Device Drivers etc.

DAY 2 :

Architectural styles

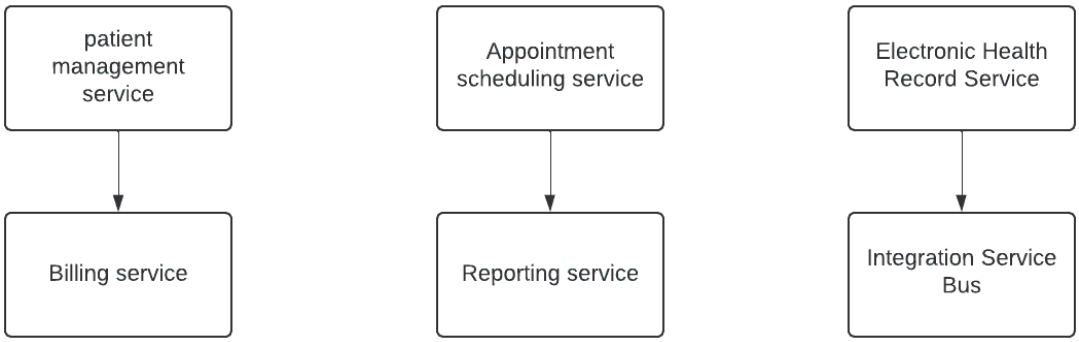
- Layered
- Client server
- Micro Services
- Event Drivers
- Service Oriented Architecture (SOA)

Service Oriented Architecture (SOA) :

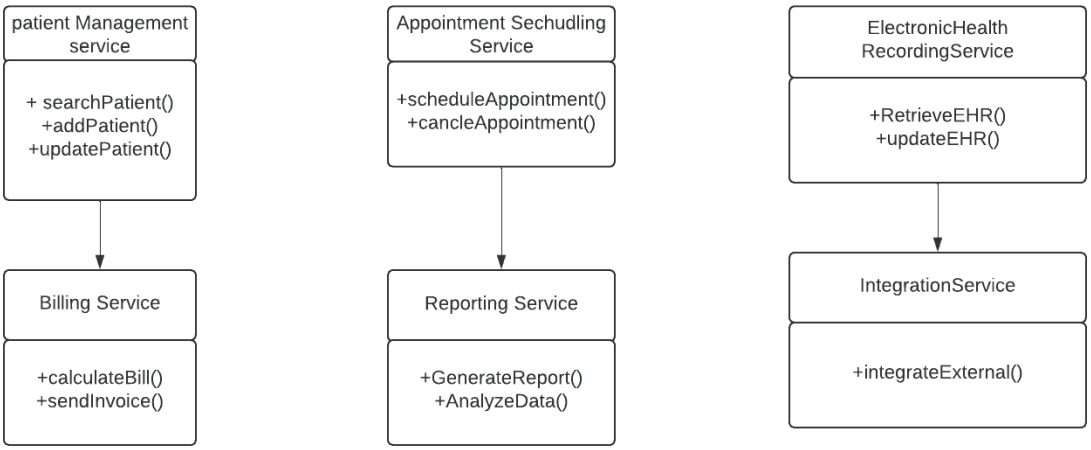
Case Study: Modernizing a Healthcare Information System with SOA.

https://pdfs.semanticscholar.org/a513/e1b39847faff37e34d6538d5e6459dde3925.pdf

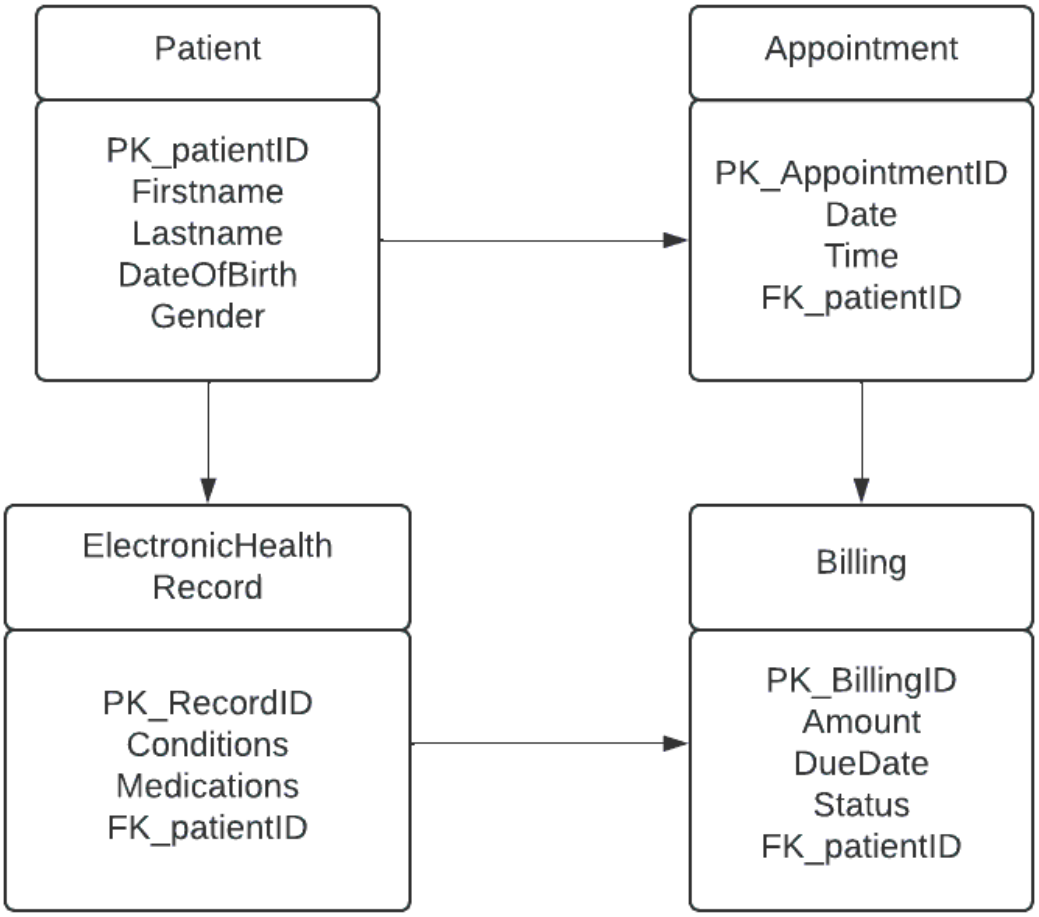
UML Component Diagram:-



UML Class Diagram :



SQL Diagram :



It is a huge collection of services in which services communicate with each other. This framework is used for designing a software system using a software architecture that views all components as services. The communication is provided by SOA generally used for data passing it could communicate two or more services and complete some activity. It provides various microservices as function which has following –

1. Loosely coupled

2. Reusable

3. Composable

4. Autonomic

5. Standardized

Advantages :-

- **Loosely coupled :** SOA promotes loose coupling between services, which means that services are not dependent on each other. This results in increased flexibility, scalability, and maintainability of the system.
- **Reusablilty** : Services are designed to be reusable, so they can be used in multiple applications.
- **Scalability:** SOA allows for easy scaling of applications because services can be added or removed without affecting the other parts of the system.
- **Flexibility:** SOA enables organizations to change and adapt to new business requirements easily, as services can be replaced or updated without affecting the rest of the system.
- **Interoperability:** SOA allows for the integration of different technologies and platforms, making it easier to connect different systems and applications.
- **Cost-effective:** SOA can be cost-effective in the long run because it reduces development time and makes it easier to maintain applications.

Disadvantage :-

- **Complexity:** SOA can be complex to design, implement, and manage, especially when dealing with large-scale systems and multiple services. This requires specialized skills and resources and can result in increased development time and costs**.**
- **Performance:** Services may introduce latency and overhead, which can negatively impact the performance of the system. This requires careful design and optimization of services and infrastructure.
- **Security:** Because services are accessible over the network, security becomes a concern, and additional measures must be taken to ensure that data is secure. This requires careful design and implementation of security measures.

- **Testing:** Testing services can be complex, and ensuring that all services work together correctly can be challenging. This requires specialized testing tools and techniques and can result in increased testing time and costs.
- **Governance:** Proper governance is required to ensure that services are designed, developed, and deployed according to established standards and best practices. This requires a comprehensive governance framework and a culture of compliance and accountability.

MicroService Architecture :

It takes large number of services and breaks down into small services or shareable components. It also called a monolith in which all functionality is placed into a single process. This approach is used for developing application and communication with different approaches which may write in different programming language and data storage. Here are some characteristic functions of MSA –

1. Business capabilities
2. Products
3. Smart End Point
4. Automation
5. Evolutionary

Advantage :-

- **Scalability:** Microservices can be easily scaled horizontally, allowing organizations to handle increased traffic and user requests without affecting the performance of the system
- **Agility:** Microservices enable organizations to quickly adapt to changing business requirements by allowing teams to develop and deploy services independently of each other.
- **Resilience:** Microservices are designed to be fault-tolerant, meaning that if one service fails, it does not affect the rest of the system.

- **Reusability:** Microservices can be reused across multiple applications, reducing development time and costs and promoting consistency across the organization.
- **Technology Diversity:** Microservices allow organizations to use different technologies and platforms for different services, enabling teams to choose the best technology for each service.
- **Continuous Delivery:** Microservices can be deployed and updated independently of each other, enabling continuous delivery and reducing time-to-market.
- **Easy Maintenance:** Microservices are easier to maintain than monolithic applications because each service is smaller and has fewer dependencies.
- **Increased Collaboration:** Microservices enable teams to work independently and collaborate effectively, resulting in increased productivity and faster development cycles.

Disadvantage :

- **Complexity:** Microservices introduce a higher level of complexity compared to monolithic applications. This complexity can result in increased development time and costs and requires specialized skills and resources.
- **Distributed System:** MSA involves building a distributed system, which can result in increased latency, network overhead, and communication complexity.
- **Data Management:** Managing data across different services can be challenging, and organizations must implement effective data management and synchronization strategies.
- **Testing:** Testing microservices can be complex and time-consuming, and organizations must implement effective testing strategies to ensure that all services work together correctly.
- **Deployment and Infrastructure Management:** Deploying and managing microservices can be challenging, especially when dealing with a large number of services. Organizations must have effective deployment and infrastructure management strategies in place.
- **Security:** Because microservices are distributed, security becomes a concern, and additional measures must be taken to ensure that data is secure.

- **Governance:** Proper governance is required to ensure that services are designed, developed, and deployed according to established standards and best practices.

DAY 3

SOA presentation

DAY 4

MVC (Model-View-Controller) and its variants

- The MVC full form, is a design pattern that divides your application into three primary parts: a model, a view, and a controller.
- The controller manages the flow of data between these two components.

Model :-

- The model part of the MVC architecture contains all relevant information about what should be stored in the database or other places where it would be needed later on during the development or testing of your codebase.

View :-

- The view part of the MVC architecture contains HTML code that describes how to display something on the screen which can then be accessed by controllers in order to respond to user requests and perform actions based on user input received through actions passed down from higher levels.

Controller :-

- The controller part of the MVC architecture is responsible for accepting requests from users and passing them on to appropriate views.
- The MVC controller also handles any tasks that need to be performed before or after displaying a view, such as validating input or storing data.

- The MVC controller is responsible for coordinating interactions between models and views.

MVC Features :-

- The MVC framework is well-suited for applications that involve complex interactions between multiple objects and views. Some of the MVC framework features are:
- Reduced complexity
- Increased testability and maintainability
- Better separation of concerns

The MVC pattern subsequently evolved, giving rise to variants such **as hierarchical model–view–controller (HMVC), model–view–adapter (MVA), model–view–presenter (MVP), model–view–viewmodel (MVVM),** and others that adapted MVC to different contexts.

model–view–presenter (MVP) :-

Model–view–presenter (MVP) is a derivation of the model–view–controller (MVC) architectural pattern, and is used mostly for building user interfaces. In MVP, the presenter assumes the functionality of the "middle-man". In MVP, all presentation logic is pushed to the presenter.

hierarchical model–view–controller (HMVC) :-

Hierarchical Model View Controller (HMVC) is an extension on the traditional Model view controller (MVC) architecture. It's main purpose is for use in web applications. The HMVC came about as a solution to scalability problems present in applications which used MVC**.**

model–view–adapter (MVA) :-

MVA is an architectural pattern that is also known as the “mediating controller MVC” or MVC with the mediator pattern applied. This pattern has all traffic pass through a “mediator” which is the Adapter in this case.

model–view–viewmodel (MVVM) :-

It is an architectural pattern which addresses the uneven scattering of functionality in an MVC by shifting the interface logic away from the View and presenting the view with bindings for direct access to model state or transforms/logic for model state.

Software design pattern :-

Software design patterns are established solutions to common design problems that software developers face. They are templates or best practices for designing software architectures and solving issues in a way that is both effective and reusable. Here's a detailed overview of some of the most important design patterns, categorized into three main types: Creational, Structural, and Behavioral.

1. Creational Design Patterns

Creational patterns focus on how objects are created. They abstract the instantiation process, making it more flexible and efficient.

Singleton

Purpose: Ensures that a class has only one instance and provides a global point of access to it.

Example: A configuration manager that reads configuration settings from a file.

-Example Code (Java):

java

```
public class Singleton {
        private static Singleton instance;
        private Singleton() {}
        public static Singleton getInstance() {
                if (instance == null) {
                        instance = new Singleton();
                }
                return instance;
        }
}
```

Factory Method

Purpose: Defines an interface for creating objects, but allows subclasses to alter the type of objects that will be created.

Example: A document creation application where the type of document (Word, PDF, etc.) is decided at runtime.

Example Code (Java):

java

```
abstract class Document
{
```

```
    abstract void create();
}

class WordDocument extends Document {
    @Override
    void create() {
        System.out.println("Creating a Word document.");
    }
}

class DocumentFactory {
    public Document createDocument(String type) {
        if (type.equals("Word")) {
            return new WordDocument();
        }
        // Additional types can be added here
        return null;
    }
}
```

Abstract Factory

Purpose: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Example: Creating user interfaces with different themes (light mode, dark mode).

Example Code (Java):

java

```
interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}
```

```
class WinFactory implements GUIFactory {
    public Button createButton() { return new WinButton(); }
    public Checkbox createCheckbox() { return new WinCheckbox(); }
}

class MacFactory implements GUIFactory {
    public Button createButton() { return new MacButton(); }
    public Checkbox createCheckbox() { return new MacCheckbox(); }
}
```

Builder

Purpose: Separates the construction of a complex object from its representation so that the same construction process can create different representations.

Example: Building a complex meal with different combinations of dishes.

Example Code (Java):

```
java
class Meal {
    private String mainCourse;
    private String drink;
    public void setMainCourse(String mainCourse) { this.mainCourse = mainCourse; }
    public void setDrink(String drink) { this.drink = drink; }
}
abstract class MealBuilder {
    protected Meal meal = new Meal();
    public abstract void buildMainCourse();
    public abstract void buildDrink();
    public Meal getMeal() { return meal; }
}
class VegMealBuilder extends MealBuilder {
```

```
        public void buildMainCourse() { meal.setMainCourse("Vegetarian Burger"); }
        public void buildDrink() { meal.setDrink("Lemonade"); }
    }
```

Prototype

Purpose: Creates new objects by copying an existing object, known as the prototype.

Example: Copying objects with default settings for a new configuration.

Example Code (Java):

java

```
interface Prototype {
    Prototype clone();
}

class ConcretePrototype implements Prototype {
    @Override
    public Prototype clone() {
        return new ConcretePrototype();
    }
}
```

2. Structural Design Patterns

Structural patterns focus on how objects and classes are composed to form larger structures.

Adapter (or Wrapper)

Purpose: Allows incompatible interfaces to work together.

Example: Adapting a legacy system interface to a new system.

Example Code (Java):

java

```
interface Target {
    void request();
```

```
    }
class Adaptee {
        void specificRequest() {
            System.out.println("Specific request.");
        }
    }
class Adapter implements Target {
        private Adaptee adaptee;
        public Adapter(Adaptee adaptee) { this.adaptee = adaptee; }
        public void request() { adaptee.specificRequest(); }
    }
```

Decorator

Purpose: Adds new functionality to an object without altering its structure.

Example: Adding scroll bars to a window.

Example Code (Java):

java

```
interface Window {
        void draw();
    }
class SimpleWindow implements Window {
        public void draw() {
            System.out.println("Drawing a simple window.");
        }
    }
abstract class WindowDecorator implements Window {
        protected Window decoratedWindow;
        public WindowDecorator(Window decoratedWindow) { this.decoratedWindow =
decoratedWindow; }
        public void draw() { decoratedWindow.draw(); }
    }
```

```
class ScrollableWindow extends WindowDecorator {
        public ScrollableWindow(Window decoratedWindow) { super(decoratedWindow); }
        public void draw() {
            super.draw();
            System.out.println("Adding scroll bars.");
        }
    }
```

Composite

Purpose: Allows clients to treat individual objects and compositions of objects uniformly.

Example: A file system where files and directories are treated similarly.

Example Code (Java):

java

```
interface Component {
        void operation();
    }
class Leaf implements Component {
        public void operation() {
            System.out.println("Leaf operation.");
        }
    }
class Composite implements Component {
        private List<Component> children = new ArrayList<>();
        public void add(Component component) { children.add(component); }
        public void operation() {
            for (Component child : children) {
                child.operation();
            }
        }
    }
```

Facade

Purpose: Provides a simplified interface to a complex subsystem.

Example: A simplified API for a complex library.

Example Code (Java):

```
java
class SubsystemA {
    void operationA() { System.out.println("Subsystem A operation."); }
}
class SubsystemB {
    void operationB() { System.out.println("Subsystem B operation."); }
}
class Facade {
    private SubsystemA a = new SubsystemA();
    private SubsystemB b = new SubsystemB();
    public void performOperation() {
        a.operationA();
        b.operationB();
    }
}
```

Bridge

Purpose: Decouples an abstraction from its implementation so that the two can vary independently.

Example: Drawing different shapes (circle, square) in different colors.

Example Code (Java):

```
java
interface DrawingAPI {
    void drawCircle(int x, int y, int radius);
}
class ConcreteDrawingAPI1 implements DrawingAPI {
    public void drawCircle(int x, int y, int radius) {
        System.out.println("Drawing API 1: Circle at (" + x + ", " + y + ") with radius " + radius);
    }
```

```
    }
class Circle {
        private int x, y, radius;
        private DrawingAPI drawingAPI;
        public Circle(int x, int y, int radius, DrawingAPI drawingAPI) {
            this.x = x; this.y = y; this.radius = radius; this.drawingAPI = drawingAPI;
        }
        public void draw() {
            drawingAPI.drawCircle(x, y, radius);
        }
    }
```

Proxy

Purpose: Provides a surrogate or placeholder for another object.

Example: A proxy that manages access to a resource-heavy object.

Example Code (Java):

```
    java
    interface Image {
        void display();
    }
class RealImage implements Image {
        private String filename;
        public RealImage(String filename) { this.filename = filename; }
        public void display() { System.out.println("Displaying " + filename); }
    }
class ProxyImage implements Image {
        private RealImage realImage;
        private String filename;
        public ProxyImage(String filename) { this.filename = filename; }
        public void display() {
            if (realImage == null) {
```

```
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}
```

3. Behavioral Design Patterns

Behavioral patterns focus on communication between objects and how responsibilities are distributed.

Chain of Responsibility

Purpose: Passes a request along a chain of potential handlers until one of them handles it.

Example: A help desk where requests are escalated through different levels.

Example Code (Java):

java

```
abstract class Handler {
    private Handler next;
    public void setNext(Handler next) { this.next = next; }
    public void handleRequest(int request) {
        if (next != null) {
            next.handleRequest(request);
        }
    }
}
class ConcreteHandlerA extends Handler {
    public void handleRequest(int request) {
        if (request < 10) {
            System.out.println("Handler A handled request " + request);
        } else {
            super.handleRequest(request);
        }
```

cloud computing :-

Comprehensive Guide to Cloud Computing

Cloud computing is a revolutionary technology that enables the delivery of computing services over the internet. It offers scalable resources on demand, ranging from computing power and storage to advanced services like AI and big data analytics. This guide will cover everything you need to know about cloud computing, including its types, key services, architectures, best practices, and real-world use cases.

1. What is Cloud Computing?

Cloud computing provides a range of IT resources and services over the internet. Instead of owning and maintaining physical hardware and software, users access computing resources on a pay-as-you-go basis.

Key Characteristics of Cloud Computing

| Characteristic | Description |

| On-Demand Self-Service | Users can provision computing capabilities as needed without human intervention from service providers. |

| Broad Network Access | Services are available over the network and can be accessed from various devices (smartphones, tablets, PCs). |

| Resource Pooling | Providers pool computing resources to serve multiple consumers using a multi-tenant model. |

| Rapid Elasticity | Resources can be scaled up or down quickly based on demand. |

| Measured Service | Resource usage is measured, and users are billed based on their consumption. |

Benefits of Cloud Computing

Cost Efficiency: Reduces upfront capital expenditures and offers a pay-as-you-go model.

Scalability: Easily scale resources up or down based on demand.

Flexibility: Access resources and services from anywhere at any time.

Automatic Updates: Cloud providers handle software updates and maintenance.

Disaster Recovery: Cloud solutions often include backup and disaster recovery options.

2. Types of Cloud Computing Services

Cloud computing offers various services, which can be categorized into three main types:

2.1 Service Models

| Model | Description
|

| Infrastructure as a Service (IaaS) | Provides virtualized computing resources over the internet, including servers, storage, and networking. |

| Platform as a Service (PaaS) | Offers hardware and software tools over the internet, typically for application development. |

| Software as a Service (SaaS) | Delivers software applications over the internet on a subscription basis. |

IaaS Example Providers

|Provider | Description |

| Amazon Web Services (AWS) | Offers services like EC2, S3, and VPC.
|

| Microsoft Azure | Provides VMs, Blob Storage, and Virtual Networks.
|

| Google Cloud Platform (GCP) | Includes Compute Engine, Cloud Storage, and VPC.
|

PaaS Example Providers

| Provider | Description |

| Heroku | Provides a platform for building and running apps. |

| Google App Engine| A fully managed PaaS for app deployment and scaling.
|

| Microsoft Azure App Services | Offers web apps, mobile backends, and RESTful APIs.
|

SaaS Example Providers

| Provider | Description |

| Google Workspace| Includes Gmail, Docs, Drive, and Calendar. |

| Salesforce | Provides CRM solutions and business apps. |

| Office 365 | Offers productivity tools like Word, Excel, and Outlook. |

2.2 Deployment Models

| Model | Description |

| Public Cloud | Services are offered over the public internet and shared among multiple organizations. |

| Private Cloud | Services are maintained on a private network and used exclusively by a single organization. |

| Hybrid Cloud | A combination of public and private clouds, allowing data and applications to be shared between them. |

| Community Cloud | Shared infrastructure for a specific community of organizations with common concerns. |

3. Cloud Computing Architectures

3.1 Basic Architecture

| Component | Description |

| Cloud Provider | Company offering cloud services (e.g., AWS, Azure, Google Cloud).

| Cloud Users | Individuals or organizations that use cloud services. |

| Service Models | IaaS, PaaS, SaaS models providing different levels of service. |

| Infrastructure | Physical data centers and virtual resources like servers, storage, and networks. |

Basic Cloud Computing Architecture Diagram:

Source: Google Cloud

3.2 Components of Cloud Architecture

| Component | Description |

| Compute | Virtual machines, containers, or serverless functions. |

| Storage | File storage, block storage, and object storage solutions. |

| Networking | Virtual private clouds, load balancers, and DNS management.

|

| Databases | Relational databases, NoSQL databases, and data warehousing solutions.
|

| Security | Identity management, encryption, and security monitoring.
|

| Management | Tools for monitoring, billing, and orchestrating cloud resources.
|

3.3 Cloud Computing Models

| Model | Description
|

| Serverless| Running applications without managing servers. Examples: AWS Lambda, Azure Functions, Google Cloud Functions. |

| Containers | Encapsulating applications and dependencies in containers. Examples: Docker, Kubernetes. |

4. Cloud Computing Best Practices

4.1 Security Best Practices

| Practice | Description
|

| Use IAM Roles | Implement Identity and Access Management (IAM) roles for secure access controls.
|

| Encrypt Data | Encrypt data at rest and in transit to protect sensitive information.
|

| Regular Updates| Keep your systems and applications up-to-date with the latest security patches.
|

| Monitor Activity| Implement logging and monitoring to detect and respond to suspicious activities.
|

| Backup Data| Regularly backup data and ensure recovery procedures are in place.
|

4.2 Cost Management

| Practice | Description
|

| Right-Sizing Resources| Allocate resources based on actual needs to avoid over-provisioning and

reduce costs. |

| Use Reserved Instances | Commit to using resources for a longer term to receive discounts.
|

| Monitor Billing | Regularly review billing statements and set up alerts for unexpected charges.
|

| Optimize Storage Costs| Use cost-effective storage solutions and manage data lifecycle policies.

4.3 Performance Optimization

| Practice | Description
|

| Auto-Scaling | Implement auto-scaling to adjust resources based on demand.

| Load Balancing | Distribute workloads across multiple servers or instances for better performance and availability. |

| Optimize Applications| Fine-tune application performance for better efficiency.

| Monitor Performance | Use tools for performance monitoring and optimization.

4.4 Compliance and Governance

| Practice | Description
|

| Adhere to Regulations| Ensure compliance with industry regulations and standards.
|

| Implement Policies| Establish cloud governance policies for resource management and security.
|

| Audit Regularly | Perform regular audits to ensure compliance and security measures are effective.
|

5. Real-World Use Cases of Cloud Computing**

5.1 Web Hosting

Use Case: Hosting websites and web applications.

Example: Hosting an e-commerce site using AWS EC2 instances and S3 for static content.

5.2 Data Backup and Disaster Recovery

Use Case: Protecting data and ensuring business continuity.

Example: Using Google Cloud Storage for backups and Google Cloud Disaster Recovery services.

5.3 Application Development

Use Case: Building, testing, and deploying applications.

Example: Developing a mobile app with Azure App Services and integrating with Azure SQL Database.

5.4 Big Data Analytics

Use Case: Analyzing large datasets for insights and decision-making.

Example: Using AWS Redshift for data warehousing and analyzing user behavior.

5.5 Machine Learning and AI

Use Case: Building and deploying machine learning models.

Example: Using Google AI Platform to train and deploy machine learning models for predictive analytics.

5.6 IoT Solutions

Use Case: Managing and analyzing data from Internet of Things (IoT) devices.

Example: Using Azure IoT Hub to connect and manage IoT devices, and Azure Stream Analytics for data processing.

5.7 DevOps and CI/CD

Use Case: Automating the development and deployment pipelines.

Example: Using AWS CodePipeline for continuous integration and delivery.