Name: FIRDOSE KOUSER

Resume Number: 24408873

Day1….. Computer Architecture Software.

Write a differnces between System software and Application software with an examples in computer architecture software devices n utilities.

System Software v/s Application Software:

System Software:

Definition: System software manages and supports the operation of a computer system, providing a platform for application software to run on. It interacts directly with computer hardware and provides essential services for the functioning of the system.

Examples:

Operating Systems: Windows, macros, Linux, Unix.

Role: Manages hardware resources, provides a user interface, and supports the execution of applications.

Device Drivers: Software that enables communication between the operating system and hardware devices like printers, graphics cards, and storage devices.

Role: Facilitates the interaction between hardware components and the operating system, ensuring proper functionality and performance.

Utility Programs: Tools such as disk management utilities (disk cleanup, defragmentation), antivirus software, system diagnostic tools.

Role: Helps manage, optimize, and protect the system and its resources.

Shells and Command-Line Interfaces : Bash: Commonly used in Unix-like operating systems for scripting and command execution.

PowerShell: A task automation framework from Microsoft with a command-line shell and

Application Software:

Definition: Application software is designed to perform specific tasks or provide services for end-users, typically built on top of system software. It interacts with system resources through APIs provided by the operating system.

Examples:

Word Processing Software: Microsoft Word, Google Docs, LibreOffice Writer.

Role: Used for creating and editing documents.

Graphic Design Software: Adobe Photoshop, CorelDRAW, GIMP.

Role: Used for editing and creating digital graphics and designs.

Web Browsers: Chrome, Firefox, Safari.

Role: Used to access and interact with websites and online content.

Games: Fortnite, Minecraft, League of Legends.

Role: Entertainment software for gaming purposes.

Web Applications: Gmail: A free email service provided by Google, accessible through a web browser.

Role: YouTube: A video-sharing platform where users can upload, view, and share videos.

Key Differences:

Purpose: System software provides foundational services and manages hardware resources to facilitate the operation of the computer system. Application software serves specific user needs or tasks, such as document editing, graphic design, or entertainment.

Dependency: System software is essential for the operation of the computer system and must be present for any application software to run. Application software, while enhancing user functionality, depends on the underlying system software and hardware for execution.

Day2

Software architecture styles write a case study for clients-server architecture and service oriented architecture

Case Study: Client-Server and Service-Oriented Architecture

Client-Server Architecture

Overview:

The Client-Server architecture is a distributed application structure that partitions tasks between clients (requesters of services) and servers (providers of services). Clients initiate communication sessions with servers, which await incoming requests.

Scenario: E-Commerce Platform

Business Requirement: An e-commerce platform needs to handle a large volume of customer transactions, including browsing products, placing orders, and processing payments.

Implementation:

Client-Side:

- Web and mobile applications for customer interaction.
- User interfaces for browsing products, managing shopping carts, and checking out.

Server-Side:

- Application servers to handle business logic.
- Database servers to store product information, user data, and transaction records.

Architecture Diagram:

- [Client-Side]

- Web Browser

- Mobile App

- [Server-Side]

- Application Server (Handles business logic, user sessions)

- Database Server (Stores product details, user data, transactions)

Benefits:

- Centralized control over application data and logic.
- Easy to update and maintain server-side components.
- Scalability by adding more servers.

Challenges:

- Server can become a bottleneck under heavy load.
- Requires robust server security to protect against attacks.

Service-Oriented Architecture (SOA)

Case Study:

Overview:

SOA is an architectural pattern where services are provided to the other components by application components, through a communication protocol over a network. It emphasizes reusability, scalability, and interoperability.

Scenario: Financial Services Integration

Business Requirement:

A financial institution wants to integrate multiple independent services, such as customer account management, loan processing, and fraud detection, into a cohesive system.

Implementation:

1. **Services:**
    - **Customer Account Service:** Manages customer details and account activities.
    - **Loan Processing Service:** Handles loan applications, approvals, and disbursements.
    - **Fraud Detection Service**: Monitors transactions for potential fraudulent activity.
2. **Communication:**
    - Services communicate over a network using standard protocols (e.g., HTTP, SOAP, REST).
3. **Service Registry:**
    - A registry to manage and discover available services.

Architecture Diagram:

[Client Applications]

- Web Portal

- Mobile App

[Service Layer]

- Customer Account Service

- Loan Processing Service

- Fraud Detection Service

[Communication]

- HTTP/REST/SOAP

[Service Registry]

- Service Discovery

Benefits:

- Loose coupling between services allows for independent development and deployment.
- Reusability of services across different applications.
- Scalability by adding more instances of a service.

Challenges:

- Requires robust network communication and management.
- Complex service orchestration and governance.
- Security concerns with data transmitted over the network.

➢ https://www.s-cube-network.eu/results/complete-solution-for-automative-supply-chain-case-study/case-study-files/SOA_Case%20Study_Solution_Overview.pdf

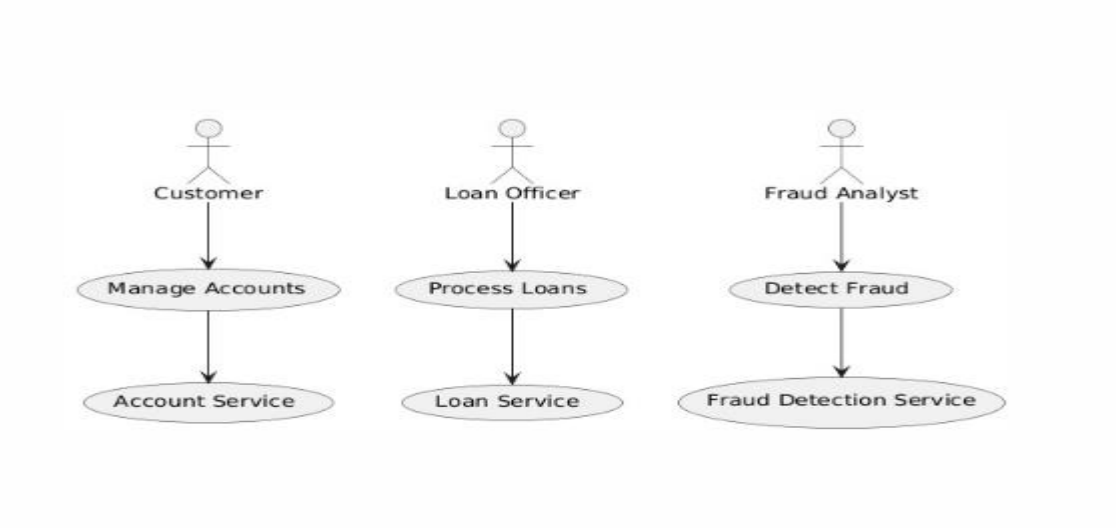
Service-Oriented Architecture (SOA)

Case Study: Scenario:b Financial Services Integration (**UML DIAGRAM**)

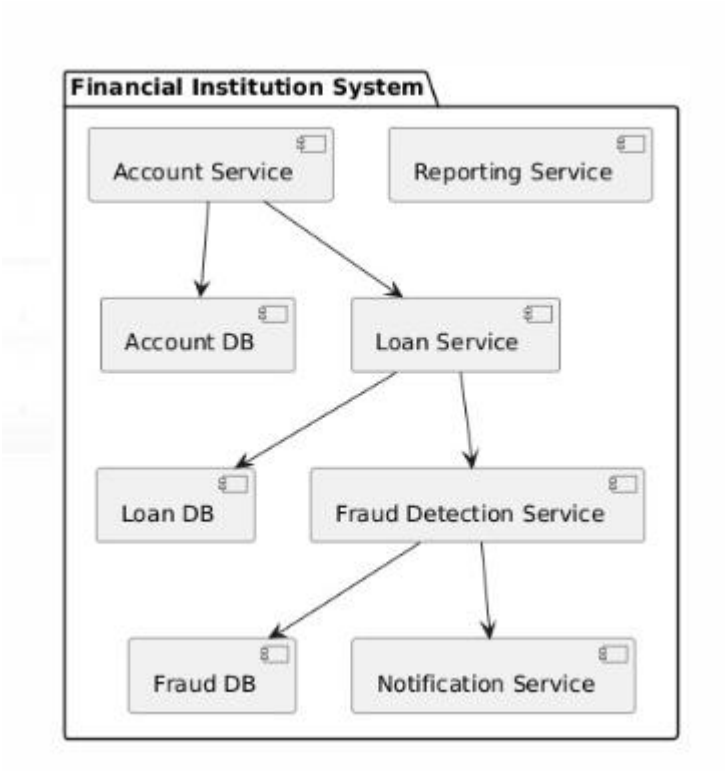
Business Requirement:

A financial institution wants to integrate multiple independent services, such as customer account management, loan processing, and fraud detection, into a cohesive system.

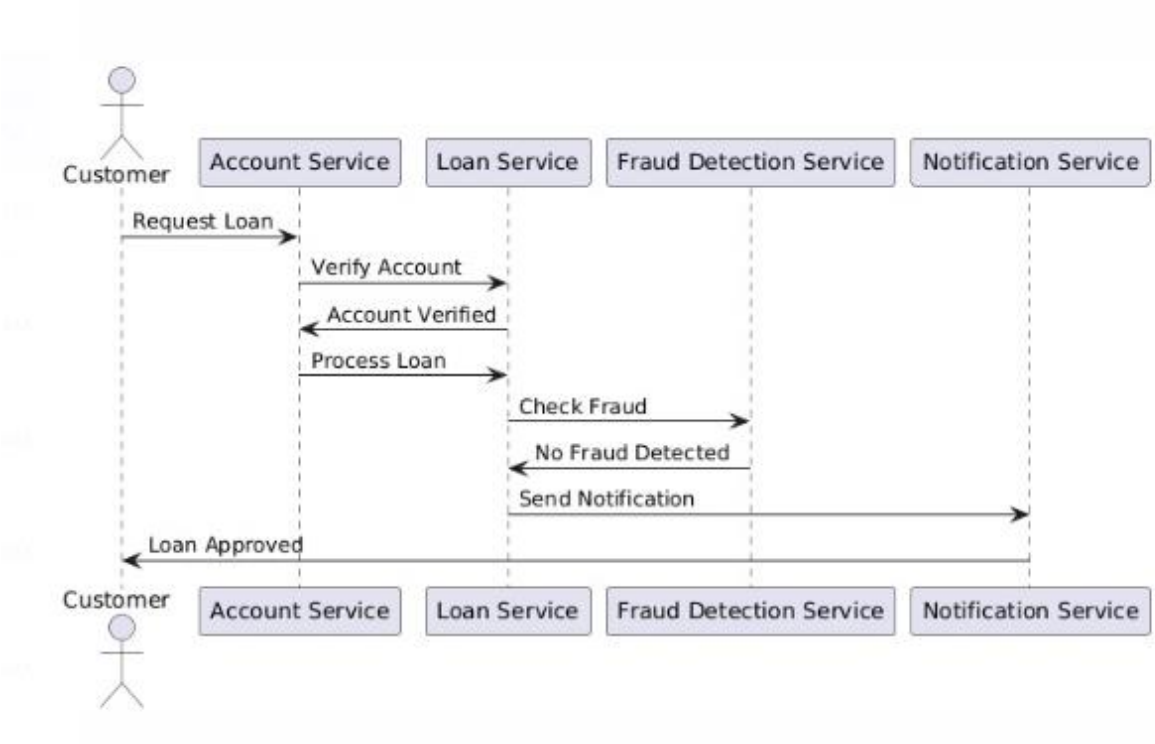
1. **Use Case Diagram: Shows the interactions between users and the system.**



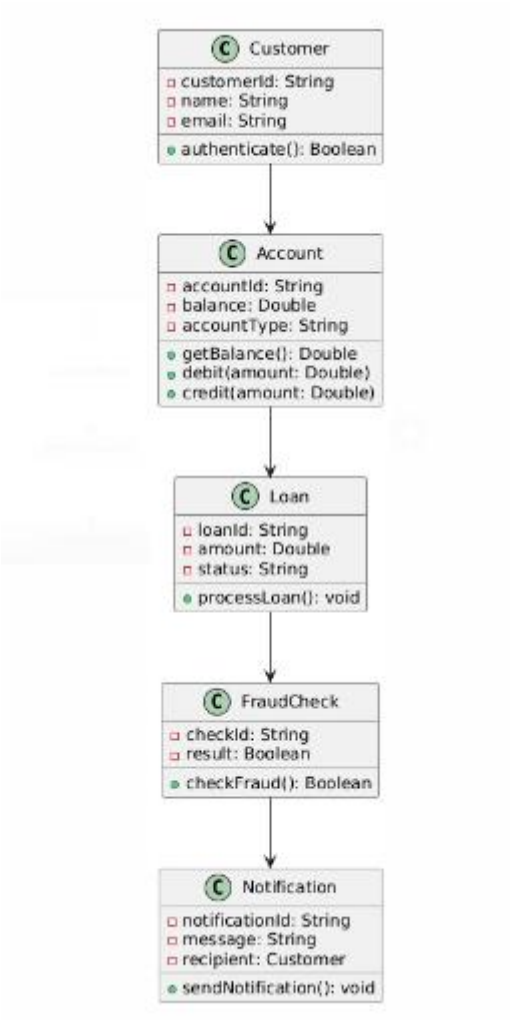
2. **Component Diagram: Depicts the various services and their interconnections.**



3. **Sequence Diagram: Illustrates the flow of messages between services.**



4. **Class Diagram: Represents the static structure of the system, including classes and their relationships.**



DAY3

PPT PRESENTATION LINK: SOA(FINANICAL SERVICE INTRAGATION)

- **https://docs.google.com/presentation/d/1Jpr_erPvLpkdRLfwPc1NccAwoyLpRG8p/edit?usp=sharing&ouid=113960673034916271967&rtpof=true&sd=true**

DAY4

Write a note on MVC (Model-View-Controller) and its variants details

The Model-View-Controller (MVC) architectural pattern is a design pattern used in software engineering to separate the internal representations of information from the ways that information is presented to and accepted by the user. Here's a breakdown of MVC and its variants:

MVC (Model-View-Controller)

- Model: Represents the data and the business logic of the application. It directly manages the data, logic, and rules of the application.
- View: Represents the user interface elements that display the data. It renders the model into a form suitable for interaction.
- Controller: Acts as an intermediary between the Model and the View. It listens to the input from the View, processes it (possibly altering the Model), and updates the View accordingly.

Variants of MVC

1. MVVM (Model-View-ViewModel):

- Model: Same as in MVC, handling the business logic and data.
- View: Same as in MVC, displaying the data.
- ViewModel: A layer that manages the state of the View and interacts with the Model. It provides a way to bind data between the View and the Model, often using data-binding techniques.
- Use Case: MVVM is commonly used in modern UI development frameworks such as WPF, Silverlight, and Angular.

2. MVP (Model-View-Presenter):

- Model: Handles the data and business logic.
- View: Displays the data but is more passive compared to MVC. It has an interface through which the Presenter interacts with it.
- Presenter: Similar to the Controller but holds more logic. It interacts with both the Model and the View, updating the View based on changes in the Model.
- Use Case: MVP is often used in Android app development.

3. MVA (Model-View-Adapter):

- Model: Represents the data and business logic.
- View: Responsible for the presentation layer.
- Adapter: Acts as a mediator that adapts the data from the Model to the View. This can be useful when the data format used by the Model is not directly compatible with what the View expects.
- Use Case: Useful in scenarios where there are different types of Views that need to display the same data.

4. PAC (Presentation-Abstraction-Control):

- Presentation: Manages the user interface.
- Abstraction: Handles the core functionality and data.
- Control: Mediates the interaction between the Presentation and Abstraction.
- Use Case: Used in complex systems with multiple layers of abstraction and presentation.

5. HMVC (Hierarchical Model-View-Controller):

- A variation of MVC that allows for nested MVC triads, meaning that a Controller can have multiple sub-controllers, each with their own Model and View.
- Use Case: Useful for large applications with complex and hierarchical user interfaces.

Understanding Software Design Patterns

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:

1.Creational, 2.Structural, and 3. 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):

```
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):

```
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):

```
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):

```
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):

```
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):

```
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):

```
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):

```
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();
        }
    }
}
```

- **Façade:**
- Purpose: Provides a simplified interface to a complex subsystem.

- Example: A simplified API for a complex library.

- Example Code (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):

```
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):

```
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):

```
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);
        }
```

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. |

2.2 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. |

2.3 **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.

2.4** 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.