DAY01

**SOFTWARE AND ITS TYPES**

Software refers to a set of instructions, data, or programs used to operate computers and execute specific tasks. It is the non-tangible component of computers, distinct from hardware, which refers to the physical components of a computer system. Software can be broadly categorized into two main types:

**System Software**:

This type of software is designed to manage and control the hardware components of a computer, allowing other software and users to interact with the hardware effectively.

Examples include:

**Operating Systems (OS)**:

The operating system is the most prominent example of system software that acts as an interface between the user and system hardware. It is a group of software that handles the execution of programs and offers general services for the application that runs over the computer. There are various types of operating systems available in the market, such as embedded operating systems, real-time OS, distributed OS, single or multi-user operating system, mobile, Internet, and various others.Such as Windows, macOS, Linux, and Android.

**Device Drivers**:

In computing, the device driver is a type of software that operates or controls some specific hardware devices linked to your system. They provide a software interface to hardware devices allowing computer operating systems and other applications to fetch hardware functions without knowing the exact specifications of the hardware. Software that allows the OS to communicate with hardware peripherals like printers, graphics cards, and network adapters.

**Firmware**:

In electronic systems and computing, firmware is a type of permanent software embedded in the system's ROM (read-only memory) to provide low-level control for some particular system device hardware. It is a set of instructions that are stored permanently on your computer's hardware device. Low-level software embedded in hardware devices, providing essential control and operation functions.

**Application Software**:

Application programs or software applications are end-user computer programs developed primarily to provide specific functionality to the user. The applications programs assist the user in accomplishing numerous tasks such as doing online research, completing notes, designing graphics, managing the finances, watching a movie, writing documents, playing games, and many more. Therefore, many software applications are designed and developed every year by companies as per the demand and requirements of the potential users. The application software can either be designed for a general-purpose or specially coded as per the requirements of business cooperation.

Today there are varieties of application software available in the market. Given below are some of the popular examples:

**Word processor**

Word processor applications are globally used for documentation, making notes, and typing data. It also helps the end-users store and format data. They also enable the users to print their documents.

Some examples of Word Processor software's are as follows:

* MS Word (Microsoft)
* iWork-Pages (Apple)
* Corel WordPerfect
* Google Docs

**Database software**

Database software is used to create, manage, modify and organize a massive amount of data quickly retrieved. Another name for database software is Database Management System (DBMS). Such software helps companies in their data organization. Common examples of Database Software's are:

* Oracle
* MS Access
* SQLite
* Microsoft SQL Server
* FileMaker
* dBase
* MariaDB
* MySQL

**Multimedia software**

This software enables the users to play, create or record images, music, and video files. Different graphic designing companies widely use multimedia software to make animation, images, posts, packaging, marketing creative, gif, or even video editing. Due to their popularity and increasing demand, every software product development corporation has massive avenues in creating and upgrading them.

Common examples of Database Software's are given below:

* Adobe Photoshop
* Windows Movie Maker
* Adobe Illustrator
* Picasa
* Windows Media Player
* Corel Draw

**Web software**

These are a type of software that is globally used to browse the Internet. Web browsers help the users in positioning as well as fetching data across the web. Common examples of web browsers are given below:

* Chrome
* Mozilla Firefox
* Microsoft Internet Explorer
* Opera
* Microsoft Edge
* UC Browser
* Apple Safari

However, there also occurs another classification of the software that exists on the basis of their availability and shareability**.** The classification is given below:

**Free ware**

As the name suggests, Freeware software is available free of cost for an unlimited time. Any user can easily download their respective software from the Internet and start using them instantly without paying any charges or fees. Software development companies mostly design and develop freeware software as a strategy to reach out to more people. Typical examples of Freeware Software are as follows:

* Adobe Reader
* Zoom
* Skype
* ImgBurn
* Audacity
* Whatsapp
* Anydesk

**Shareware**

Shareware software is readily available on the Internet to download on a fixed trial basis. It is distributed freely with a set time limit, and at the end of the trial period, the user is asked either to pay the fee or uninstall the software. Some shareware, mainly including the gaming softwares, have a fixed trial based on the counts an application is opened rather than the number of days it has been installed on the system.

Give below are some of the popular examples for Shareware Software:

* Adobe Acrobat
* Adobe Photoshop
* AnyDVD
* PHP Debugger
* WinZip

**Open source**

People usually get confused with freeware and open-source, but both are different. Though both the software are available on the Internet free of cost with the only difference that open source software is available online along with their source code. It means the user can change, transform, and even can add additional features to them. Based on their services, they can be chargeable as well free of cost.

Give below are some of the popular examples for open-source Software:

* Mozilla Firefox
* MySQL
* Thunderbird
* OpenOffice
* ClamWinantivirus

DAY 2

## **ARCHITECTURE CASE STUDY**

**SOA**

Service-oriented architecture (SOA) is a method of software development that uses software components called services to create business applications. Each service provides a business capability, and services can also communicate with each other across platforms and languages

Service-oriented architecture (SOA) is a method of software development that uses software components called services to create business applications. Each service provides a business capability, and services can also communicate with each other across platforms and languages.

**Principle of SOA**

**Standardized service contract**

Services adhere to a standard communications agreement, as defined collectively by one or more service description documents within a given set of services.

**Service reference autonomy**

The relationship between services is minimized to the level that they are only aware of their existence.

**Service location transparency**

Services can be called from anywhere within the network that it is located no matter where it is present.

**Service longevity**

Services should be designed to be long lived. Where possible services should avoid forcing consumers to change if they do not require new features, if you call a service today you should be able to call the same service tomorrow.

**Service abstraction**

The services act as black boxes, that is their inner logic is hidden from the consumers.

**Service autonomy**

Services are independent and control the functionality they encapsulate, from a Design-time and a run-time perspective.

**Service statelessness**

Services are stateless, that is either return the requested value or give an exception hence minimizing resource use.

**Service granularity**

A principle to ensure services have an adequate size and scope. The functionality provided by the service to the user must be relevant.

**Service normalization**

Services are decomposed or consolidated (normalized) to minimize redundancy. In some, this may not be done. These are the cases where performance optimization, access, and aggregation are required.[17]

**Service composability**

Services can be used to compose other services.

**Service discovery**

Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.

**Service reusability**

Logic is divided into various services, to promote reuse of code.

**Service encapsulation**

Many services which were not initially planned under SOA, may get encapsulated or become a part of SOA.

**Benefits of SOA**

Service-oriented architecture (SOA) has several benefits over the traditional monolithic architectures in which all processes run as a single unit. Some major benefits of SOA include the following:

**Faster time to market**

Developers reuse services across different business processes to save time and costs. They can assemble applications much faster with SOA than by writing code and performing integrations from scratch.

**Efficient maintenance**

It’s easier to create, update, and debug small services than large code blocks in monolithic applications. Modifying any service in SOA does not impact the overall functionality of the business process.

**Greater adaptability**

SOA is more adaptable to advances in technology. You can modernize your applications efficiently and cost effectively. For example, healthcare organizations can use the functionality of older electronic health record systems in newer cloud-based applications.

**Components of SOA**

There are four main components in service-oriented architecture (SOA).

**Service**

Services are the basic building blocks of SOA. They can be private—available only to internal users of an organization—or public—accessible over the internet to all. Individually, each service has three main features.

**Service implementation**

The service implementation is the code that builds the logic for performing the specific service function, such as user authentication or bill calculation.

**Service contract**

The service contract defines the nature of the service and its associated terms and conditions, such as the prerequisites for using the service, service cost, and quality of service provided.

**Service interface**

In SOA, other services or systems communicate with a service through its service interface. The interface defines how you can invoke the service to perform activities or exchange data. It reduces dependencies between services and the service requester. For example, even users with little or no understanding of the underlying code logic can use a service through its interface.

**Service provider**

The service provider creates, maintains, and provides one or more services that others can use. Organizations can create their own services or purchase them from third-party service vendors.

**Service consumer**

The service consumer requests the service provider to run a specific service. It can be an entire system, application, or other service. The service contract specifies the rules that the service provider and consumer must follow when interacting with each other. Service providers and consumers can belong to different departments, organizations, and even industries.

**Service registry**

A service registry, or service repository, is a network-accessible directory of available services. It stores service description documents from service providers. The description documents contain information about the service and how to communicate with it. Service consumers can easily discover the services they need by using the service registry.

**Limitation of SOA**

**Limited scalability**

System scalability is significantly impacted when services share many resources and need to coordinate to perform their functionality.

**Increasing interdependencies**

Service-oriented architecture (SOA) systems can become more complex over time and develop several interdependencies between services. They can be hard to modify or debug if several services are calling each other in a loop. Shared resources, such as centralized databases, can also slow down the system.

**Single point of failure**

For SOA implementations with an ESB, the ESB creates a single point of failure. It is a centralized service, which goes against the idea of decentralization that SOA advocates. Clients and services cannot communicate with each other at all if the ESB goes down.

**SOA CASE STUDY**

**SOA in an E-Commerce Platform**

**Problem Statement**

An e-commerce company is facing significant challenges with its monolithic architecture. The system is difficult to scale, maintain, and update. Frequent deployments cause downtime, and the development team is struggling to introduce new features without impacting the existing functionalities. The company decides to transition to a Service-Oriented Architecture (SOA) to improve scalability, flexibility, and maintainability.

**Objective**

To re-architect the e-commerce platform using SOA principles, enabling independent development, deployment, and scaling of services.

**Approach**

**Identify Core Services:**

Decompose the monolithic application into core services. For the e-commerce platform, the key services identified are:

**User Service:**

Manages user authentication, authorization, and profile management.

**Product Service:**

Handles product catalog, inventory, and product details.

**Order Service:**

Manages order processing, tracking, and history.

**Payment Service:**

Handles payment processing and transaction management.

**Notification Service:**

Manages email, SMS, and push notifications.

**Design Service Contracts:**

Define the interfaces and communication protocols for each service. Use RESTful APIs for inter-service communication.

Example API for Product Service:

GET /products - Retrieve a list of products.

POST /products - Add a new product.

PUT /products/{id} - Update product details.

DELETE /products/{id} - Remove a product.

**Choose Technology Stack:**

Programming Languages: Java for core services, Node.js for lightweight services.

Database: MySQL for transactional data, Redis for caching, and Elasticsearch for search capabilities.

Message Broker: RabbitMQ for asynchronous communication between services.

Containerization: Docker for containerizing services.

Orchestration: Kubernetes for managing containerized services.

.

**Testing and Monitoring:**

Implement comprehensive testing strategies, including unit tests, integration tests, and end-to-end tests.

Set up monitoring and logging tools (e.g., Prometheus, Grafana, ELK stack) to track service performance and diagnose issues.

**Outcome**

**Scalability:**

Services can be scaled independently based on demand. For example, the Product Service can be scaled separately during high traffic on product pages.

Flexibility: New features can be added to specific services without affecting others, reducing deployment risks.

**Maintainability:**

Smaller, well-defined services are easier to maintain and update. Teams can work on different services simultaneously without stepping on each other's toes.

Resilience: Failures in one service do not necessarily bring down the entire system. For instance, if the Notification Service fails, other core services like Order Processing continue to function.

**Challenges and Mitigation**

Inter-Service Communication:

Latency and network issues were mitigated using asynchronous messaging with RabbitMQ.

**Data Consistency:**

Ensuring data consistency across distributed services was addressed using eventual consistency patterns and implementing sagas for managing long-running transactions.

**Security:**

Implemented robust authentication and authorization mechanisms using OAuth2 and JWT tokens.

**Conclusion**

Transitioning to SOA significantly improved the e-commerce platform's performance, scalability, and maintainability. The company can now innovate rapidly, respond to market changes effectively, and provide a better user experience.

### **MICROSERVICE ARCHITECTURE**

**Definition:**

Microservice architecture is a design approach where an application is structured as a collection of small, autonomous services, each responsible for a specific business function. These services communicate over a network, typically using HTTP/REST or messaging queues.

**Key Characteristics:**

* **Independence:** Each microservice can be developed, deployed, and scaled independently.
* **Single Responsibility:** Each service is responsible for a specific piece of functionality.
* **Decentralized Data Management:** Each service manages its own database or data store.
* **Inter-Service Communication:** Services communicate with each other through APIs.

### **Benefits of Microservice Architecture**

1. **Scalability:** Individual components can be scaled independently based on demand.
2. **Flexibility in Technology:** Different services can use different technologies, languages, and frameworks best suited for their function.
3. **Improved Fault Isolation:** Failures in one service do not necessarily impact others, enhancing overall system resilience.
4. **Faster Deployment:** Smaller, independent services can be developed, tested, and deployed more quickly.
5. **Team Autonomy:** Different teams can work on different services independently, enhancing productivity and reducing bottlenecks.

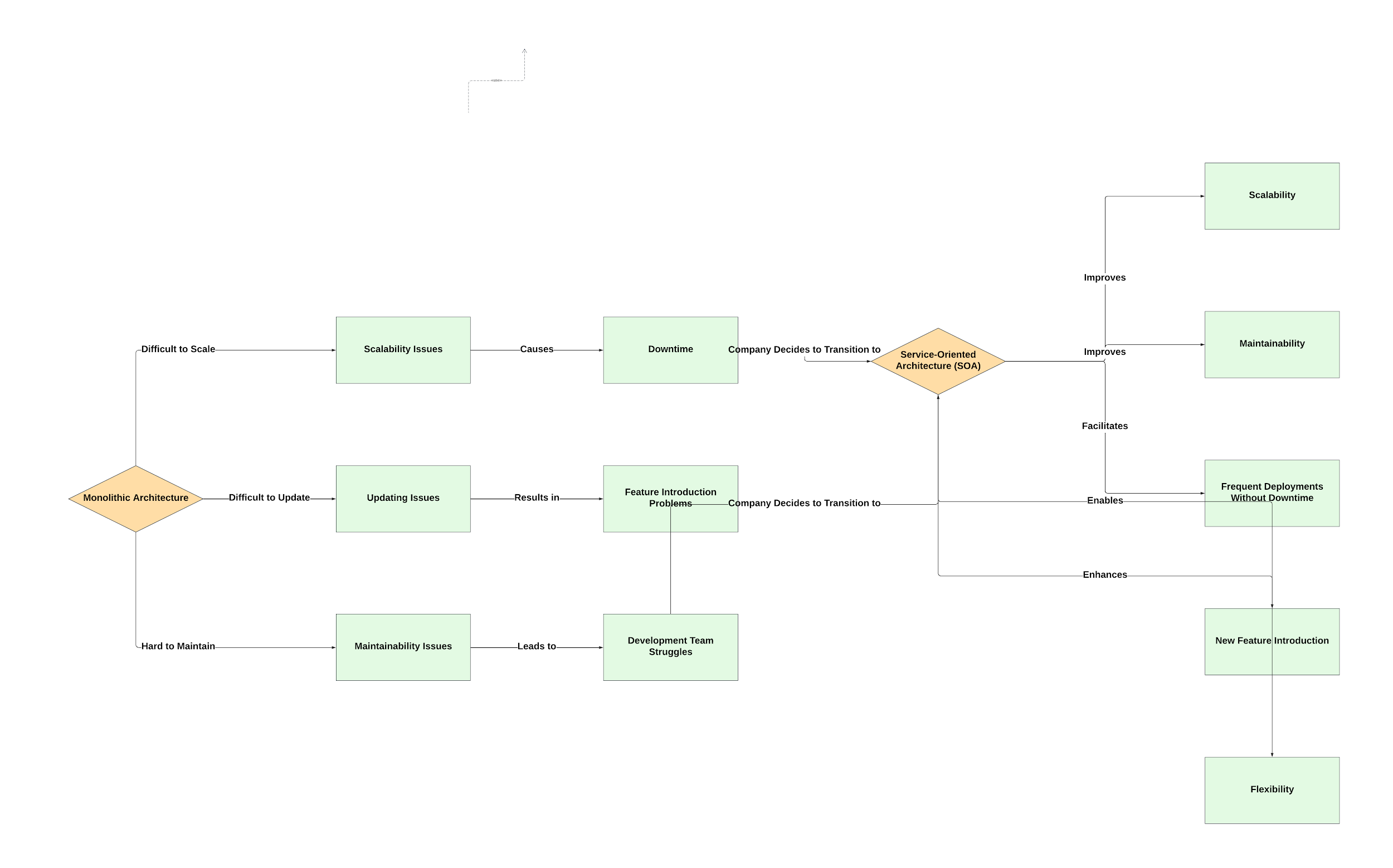
### **Pros and Cons of Microservice Architecture**

**Pros:**

* **Modular Development:** Easier to understand, develop, and maintain small modules.
* **Continuous Deployment:** Facilitates continuous integration and deployment practices.
* **Resilience:** Fault isolation ensures the entire system doesn’t go down if a single service fails.
* **Polyglot Persistence:** Use different storage technologies based on service requirements.

**Cons:**

* **Complexity:** Increased complexity in managing multiple services, dependencies, and network communications.
* **Data Management:** Challenges in maintaining data consistency across services.
* **Latency:** Network communication between services can introduce latency.
* **Testing:** Requires sophisticated testing strategies to ensure all services work together correctly.
* **Deployment:** Continuous deployment and monitoring of many services can be challenging.



### 

### 

### 

### 

### 

### 

### **Case Study: Transitioning from Monolithic to Microservice Architecture**

**Problem Statement:**

A large e-commerce platform is experiencing issues with scalability, deployment speed, and fault isolation. The monolithic architecture causes the entire application to be redeployed for any change, slowing down the release process and increasing the risk of downtime.

**Transition Strategy:**

1. **Identify Services:** Break down the monolithic application into distinct business functions, such as user management, product catalog, order processing, and payment services.
2. **Develop Microservices:** Develop these functions as independent microservices, each with its own database and API.
3. **Implement API Gateway:** Use an API gateway to handle requests and route them to the appropriate microservice.
4. **Continuous Integration and Deployment (CI/CD):** Set up CI/CD pipelines for each microservice to automate testing and deployment.
5. **Monitoring and Logging:** Implement centralized monitoring and logging to track the performance and health of each microservice.
6. **Gradual Migration:** Gradually migrate functionality from the monolithic application to the new microservices, ensuring minimal disruption.

**Challenges Faced:**

* **Service Communication:** Ensuring reliable and efficient communication between services using RESTful APIs and message queues.
* **Data Consistency:** Managing distributed transactions and data consistency across multiple services.
* **Infrastructure Management:** Handling the increased complexity of infrastructure, including container orchestration and network configurations.
* **Security:** Ensuring robust security measures are in place for inter-service communication and data handling.

**Outcomes:**

* **Improved Scalability:** The ability to scale individual services independently based on demand.
* **Faster Deployment:** Reduced deployment time and frequency of updates, enabling faster feature releases.
* **Enhanced Resilience:** Improved fault isolation, reducing the impact of individual service failures on the overall system.
* **Increased Flexibility:** The ability to use different technologies for different services, optimizing performance and resource utilization.

Microservice architecture offers significant benefits but also introduces complexities that require careful planning and management. By addressing these challenges, organizations can achieve greater agility, scalability, and resilience in their software systems.

DAY04

**MVC**

The MVC design pattern is a software architecture pattern that separates an application into three main components: Model, View, and Controller, making it easier to manage and maintain the codebase. It also allows for the reusability of components and promotes a more modular approach to software development.

**Variants of MVC**

**MVVM (Model-View-ViewModel):**

* Model:

Similar to MVC, it represents the data and business logic.

* View:

The user interface which displays the data.

* ViewModel:

An abstraction of the view exposing public properties and commands. It handles the interaction between the view and the model, allowing for a more maintainable and testable approach, especially useful in data-binding contexts like WPF or Angular.

**MVP (Model-View-Presenter):**

* Model:

Represents the data and business logic.

* View:

Displays the data and receives user input.

* Presenter:

Acts as an intermediary between the view and the model. It retrieves data from the model, updates the view, and handles user input. Unlike MVC, the view in MVP is more passive, relying on the presenter for all updates and logic.

**HMVC (Hierarchical Model-View-Controller):**

* Extends MVC by allowing nested MVC triads (sub-MVCs), making it suitable for complex applications.
* Each component (Model, View, Controller) can contain its own MVC components, leading to a hierarchical structure.

**PAC (Presentation-Abstraction-Control):**

* Presentation:

The user interface layer, similar to the view in MVC.

* Abstraction:

Represents the data and business logic, similar to the model in MVC.

* Control:

Manages the flow of the application, acting as an intermediary between presentation and abstraction layers, similar to the controller.

**Advantages of MVC**

* Separation of Concerns
* Parallel Development
* Reusability
* Maintainability
* Scalability
* Flexibility
* Enhanced User Experience
* Facilitates Testing
* Reduced Code Duplication
* Improved Security

# **SOFTWARE DESIGN PATTERN**

In software engineering, a design pattern describes a relatively small, well-defined aspect (i.e. functionality) of a computer program in terms of how to write the code.Using a pattern is intended to leverage an existing concept rather than re-inventing it. This can decrease the time to develop software and increase the quality of the resulting program.

**Categories of Design Patterns**

Design patterns are generally categorized into three main types:

**Creational Patterns:**

Deal with object creation mechanisms, trying to create objects in a manner suitable for the situation.

**Singleton:**

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

**Factory Method:**

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

**Abstract Factory:**

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

**Builder:**

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

**Prototype:**

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

**Structural Patterns:**

Deal with object composition or how classes and objects can be combined to form larger structures.

**Adapter:**

Allows incompatible interfaces to work together by converting the interface of a class into another interface that a client expects.

**Decorator:**

Adds behavior or responsibilities to an object dynamically without altering its structure.

**Proxy:**

Provides a surrogate or placeholder for another object to control access to it.

**Composite:**

Composes objects into tree structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions of objects uniformly.

**Facade:**

Provides a simplified interface to a complex subsystem, making it easier to use.

**Behavioral Patterns**:

Deal with object interaction and responsibility distribution among objects.

**Observer**:

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**Strategy**:

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.

**Command**:

Encapsulates a request as an object, thereby allowing users to parameterize clients with queues, requests, and operations.

**Iterator**:

Provides a way to access elements of a collection object sequentially without exposing its underlying representation.

**State**:

Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

**Template Method**:

Defines the skeleton of an algorithm in a method, deferring some steps to subclasses.

### **Benefits of Using Design Patterns**

**Reusability**:

Patterns provide reusable solutions that can be applied across different projects and contexts, reducing redundancy.

**Maintainability**:

Well-defined patterns make it easier to understand and maintain code.

**Scalability**:

Patterns help in designing systems that can scale more easily as they grow in complexity.

**Best Practices**:

Patterns encapsulate best practices, avoiding common pitfalls and promoting standard approaches to problem-solving.

**Communication**:

Patterns provide a common language for developers, making it easier to convey ideas and solutions.

### **Applying Design Patterns**

1. **Identify the Problem**: Understand the problem and the context in which it occurs.
2. **Select the Appropriate Pattern**: Choose a pattern that best matches the problem context.
3. **Customize the Pattern**: Adapt the pattern to fit the specific needs of the project.
4. **Implement the Solution**: Write the code following the chosen pattern.

**CLOUD COMPUTING AND SERVICES**

**CLOUD COMPUTING**

Cloud computing is the on-demand delivery of computing services such as servers, storage, databases, networking, software, and analytics

**Types of Cloud Computing**

* Software as a service (SaaS)
* Platform as a service (PaaS)
* Infrastructure as a service (IaaS)
* Anything/Everything as a service (XaaS)
* Function as a Service (FaaS)

**Software as a Service(SaaS)**

Software-as-a-Service (SaaS) is a way of delivering services and applications over the Internet. Instead of installing and maintaining software, we simply access it via the Internet, freeing ourselves from the complex software and hardware management. It removes the need to install and run applications on our own computers or in the data centers eliminating the expenses of hardware as well as software maintenance.

**Platform as a Service(PaaS)**

PaaS is a category of cloud computing that provides a platform and environment to allow developers to build applications and services over the internet. PaaS services are hosted in the cloud and accessed by users simply via their web browser.

**Infrastructure as a Service**

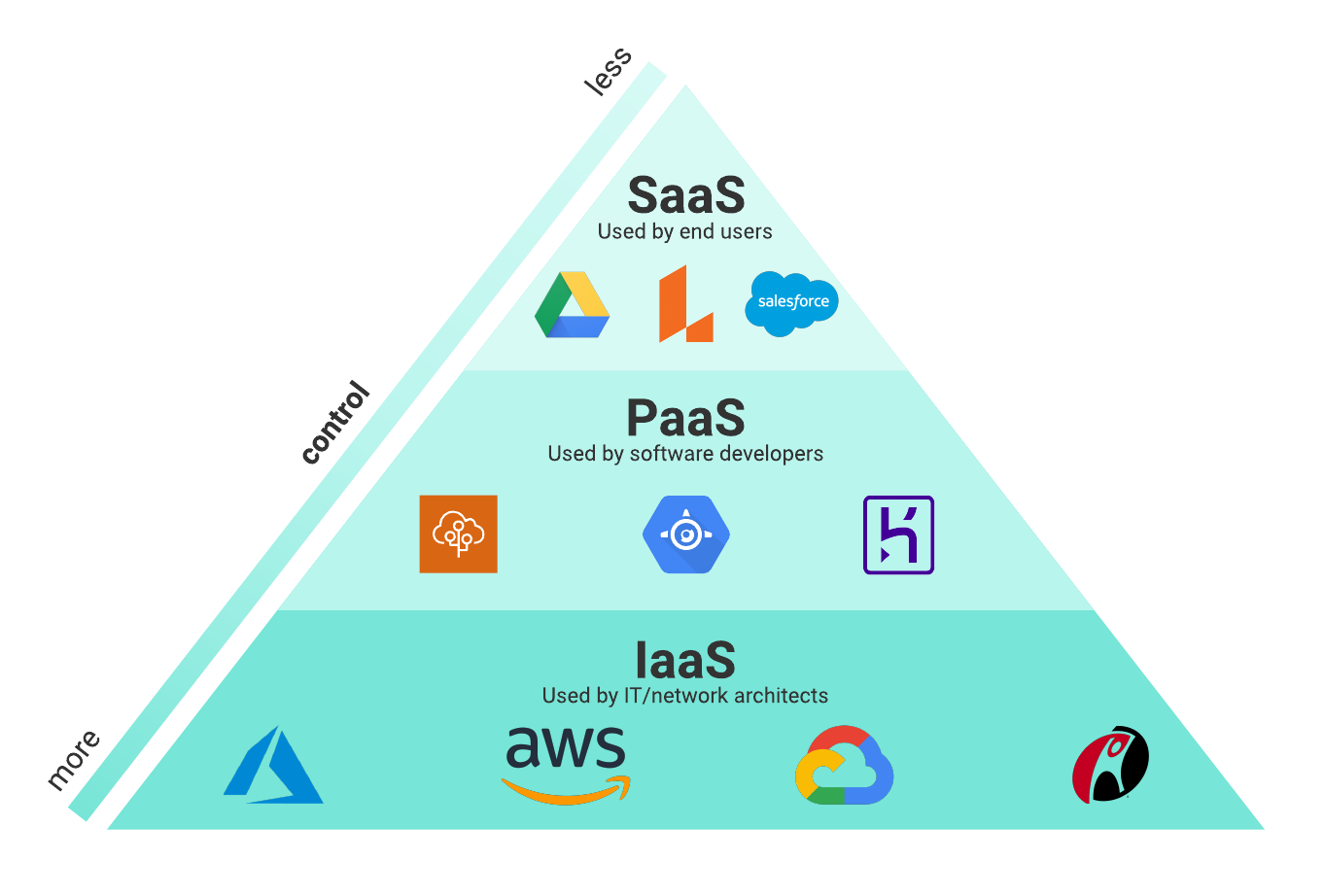
Infrastructure as a service (IaaS) is a service model that delivers computer infrastructure on an outsourced basis to support various operations. Typically IaaS is a service where infrastructure is provided as outsourcing to enterprises such as networking equipment, devices, database, and web servers. It is also known as Hardware as a Service (HaaS). IaaS customers pay on a per-user basis, typically by the hour, week, or month. Some providers also charge customers based on the amount of virtual machine space they use.

**Anything as a Service(XaaS)**

It is also known as Everything as a Service. Most of the cloud service providers nowadays offer anything as a service that is a compilation of all of the above services including some additional services.

**Function as a Service (FaaS)**

FaaS is a type of cloud computing service. It provides a platform for its users or customers to develop, compute, run and deploy the code or entire application as functions. It allows the user to entirely develop the code and update it at any time without worrying about the maintenance of the underlying infrastructure.



**DOCKER**

**What is Docker?**

Docker is an open-source platform that enables developers to build, deploy, run, update and manage containers.

**Docker architecture, terms and tools**

Docker uses a client/server architecture. The following is a breakdown of the core components associated with Docker, along with other Docker terms and tools.

**Docker host:**

A Docker host is a physical or virtual machine running Linux (or another Docker-Engine compatible OS).

**Docker Engine:**

Docker engine is a client/server application consisting of the Docker daemon, a Docker API that interacts with the daemon, and a command-line interface (CLI) that talks to the daemon.

**Docker daemon:**

Docker daemon is a service that creates and manages Docker images, by using the commands from the client. Essentially the Docker daemon serves as the control center for Docker implementation.

**Docker client:**

The Docker client provides the CLI that accesses the Docker API (a REST API) to communicate with the Docker daemon over Unix sockets or a network interface. The client can be connected to a daemon remotely, or a developer can run the daemon and client on the same computer system.

**Docker objects:**

Docker objects are components of a Docker deployment that help package and distribute applications. They include images, containers, networks, volumes, plug-ins and more.

**Docker containers:**

Docker containers are the live, running instances of Docker images. While Docker images are read-only files, containers are live, ephemeral, executable content. Users can interact with them, and administrators can adjust their settings and conditions by using Docker commands.

**Docker images:**

Docker images contain executable application source code and all the tools, libraries and dependencies the application code needs to run as a container. When a developer runs the Docker image, it becomes one instance (or multiple instances) of the container.

Building Docker images from scratch is possible, but most developers pull them down from common repositories. Developers can create multiple Docker images from a single base image and will share their stack's commonalities.

Docker images are made up of layers, and each layer corresponds to a version of the image. Whenever a developer makes changes to an image, a new top layer is created, and this top layer replaces the previous top layer as the current version of the image. Previous layers are saved for rollbacks or to be reused in other projects.

Each time a container is created from a Docker image, yet another new layer called the container layer is created. Changes made to the container—like adding or deleting files—are saved to the container layer, and these changes only exist while the container is running.

This iterative image-creation process increases overall efficiency since multiple live container instances can run from a single base image. When they do so, they use a common stack.

**Docker build:**

Docker build is a command that has tools and features for building Docker images.

**Dockerfile:**

Every Docker container starts with a simple text file containing instructions for how to build the Docker container image. Dockerfile automates the process of creating Docker images. It's essentially a list of CLI instructions that Docker Engine will run to assemble the image. The list of Docker commands is vast but standardized: Docker operations work the same regardless of contents, infrastructure or other environment variables.

**Docker documentation:**

Docker documentation, or Docker docs, refers to the official Docker library of resources, manuals and guides for building containerized applications.

**Docker Hub:**

Docker Hub6 is the public repository of Docker images, calling itself the world's largest library and community for container images7. It holds over 100,000 container images sourced from commercial software vendors, open source projects and individual developers. Docker Hub includes images produced by Docker, Inc., certified images belonging to the Docker Trusted Registry and thousands of other images.

All Docker Hub users can share their images at will. They can also download predefined base images from the Docker filesystem as a starting point for any containerization project.

Other image repositories exist, including GitHub8. GitHub is a repository hosting service well known for application development tools and as a platform that fosters collaboration and communication. Users of Docker Hub can create a repository (repo) that can hold many images. The repository can be public or private and linked to GitHub or BitBucket accounts.

**Docker Desktop:**

Docker Desktop is an application for Mac or Windows that includes Docker Engine, Docker CLI client, Docker Compose, Kubernetes and others. It also provides access to Docker Hub.

**Docker registry:**

A Docker registry is a scalable, open-source storage and distribution system for Docker images. It enables developers to track image versions in repositories by using tagging for identification. This tracking and identification are accomplished by using Git, a version control tool.

**Docker plug-ins:**

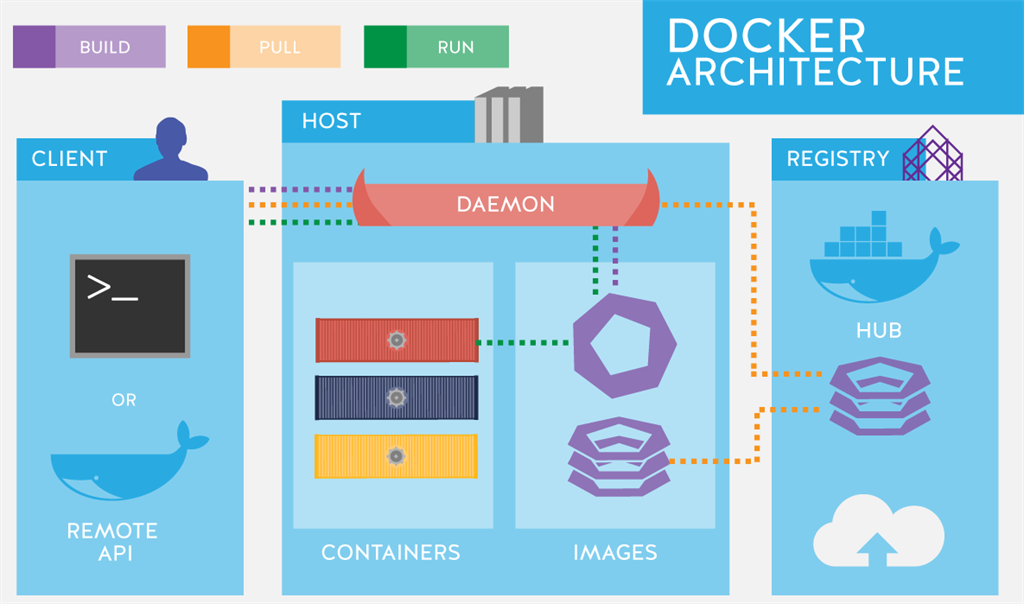
Developers use plug-ins to make Docker Engine even more functional. Several Docker plugins supporting authorization, volume and network are included in the Docker Engine plug-in system; third-party plug-ins can be loaded as well.

**Docker extensions:**

Docker extensions enable developers to use third-party tools within Docker Desktop to extend its functions. Extensions for developer tools include Kubernetes app development, security, observability and more.

**Docker Compose:**

Developers can use Docker Compose to manage multicontainer applications, where all containers run on the same Docker host. Docker Compose creates a YAML (.YML) file that specifies which services are included in the application and can deploy and run containers with a single command.



**KUBERNETES**

**What is Kubernetes?**

Kubernetes, also known as k8s or kube, is an open source container orchestration platform for scheduling and automating the deployment, management and scaling of containerized applications.

**What does Kubernetes do?**

Kubernetes schedules and automates container-related tasks throughout the application lifecycle, including the following.

**Deployment**

Deploy a specified number of containers to a specified host and keep them running in a wanted state.

**Rollouts**

A rollout is a change to a deployment. Kubernetes lets you initiate, pause, resume or roll back rollouts.

**Service discovery**

Kubernetes can automatically expose a container to the internet or to other containers by using a domain name system (DNS) name or IP address.

**Storage provisioning**

Set Kubernetes to mount persistent local or cloud storage for your containers as needed.

**Load balancing**

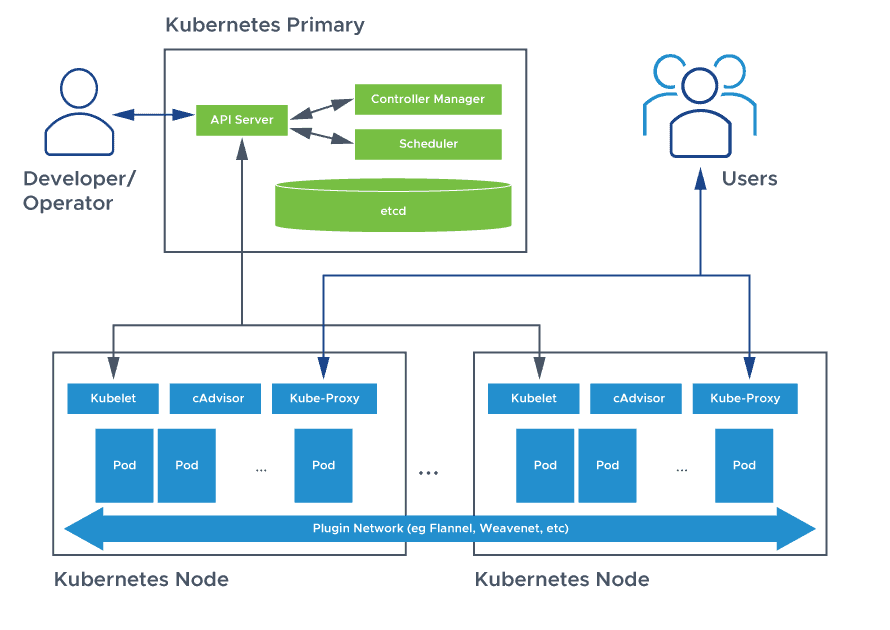
Based on CPU usage or custom metrics, Kubernetes load balancing can distribute the workload across the network to maintain performance and stability.

**Autoscaling**

When traffic spikes, Kubernetes autoscaling can spin up new clusters as needed to handle the additional workload.

**Self-healing for high availability**

When a container fails, Kubernetes can restart or replace it automatically to prevent downtime. It can also take down containers that don’t meet your health check requirements.



**Kubernetes architecture and components**

Deploying Kubernetes involves clusters, the building blocks of Kubernetes architecture. Clusters are made up of nodes, each representing a single compute host, either a physical machine (bare metal server) or a VM.

Kubernetes architecture consists of two main parts: the control pane components and the components that manage individual nodes.

A node consists of pods. These are groups of containers that share the same computing resources and the same network. They are also the unit of scalability in Kubernetes. If a container in a pod is gaining more traffic than it can handle, Kubernetes will replicate the pod to other nodes in the cluster.

The control plane automatically handles scheduling the pods across the nodes in a cluster.

**Control plane components**

Each cluster has a master node that handles the cluster’s control plane. The master node runs a scheduler service that automates when and where the containers are deployed based on developer-set deployment requirements and available computing capacity.

The main components in a Kubernetes cluster are the kube-apiserver, etcd, kube-scheduler, kube-controller-manager and cloud-controller-manager:

**API server:**

The application programming interface (API) server in Kubernetes exposes the Kubernetes API (the interface used to manage, create and configure Kubernetes clusters) and serves as the entry point for all commands and queries.

**etcd:**

The etcd is an open source, distributed key-value store used to hold and manage the critical information that distributed systems need to keep running. In Kubernetes, the etcd manages the configuration data, state data and metadata.

**Scheduler:**

This component tracks newly created pods and selects nodes for them to run on. The scheduler considers resource availability and allocation restraints, hardware and software requirements, and more.

**Controller-manager:**

A set of built-in controllers, the Kubernetes controller-manager runs a control loop that monitors the shared state of the cluster and communicates with the API server to manage resources, pods or service endpoints. The controller-manager consists of separate processes that are bundled together to reduce complexity and run in one process.

**Cloud-controller-manager:**

This component is similar in function to the controller-manager link. It links to a cloud provider’s API and separates the components that interact with that cloud platform from those that only interact within the cluster.

**Node components**

Worker nodes are responsible for deploying, running and managing containerized applications:

**Kubelet:**

Kubelet is a software agent that receives and runs orders from the master node and helps to ensure that containers run in a pod.

**Kube-proxy:**

Installed on every node in a cluster, the kube-proxy maintains network rules on the host and monitors changes in services and pods.