1. What is DevOps ?

Devops is a software development methodology which improves the collaboration between developers and operations team using various automation tools. These automation tools are implemented using various stages which are a part of the Devops Lifecycle

Developer:

A developer's job is to develop applications and pass his code to the operations team.

Operations:

The operations team job is to test the code, and provide feedback to developers in case of bugs. If all goes well, the operations team uploads the code to the build servers.

WHY DevOps??

Old Scenario:

The developer used to run the code on his system, and then forward it to operations team.

The operations when tried to run the code on their system, it did not run!

The operations then marked this code as faulty, and used to forward this feedback to

the developer.

But, the code runs fine on the developer's system and hence he says "It is not my fault!".

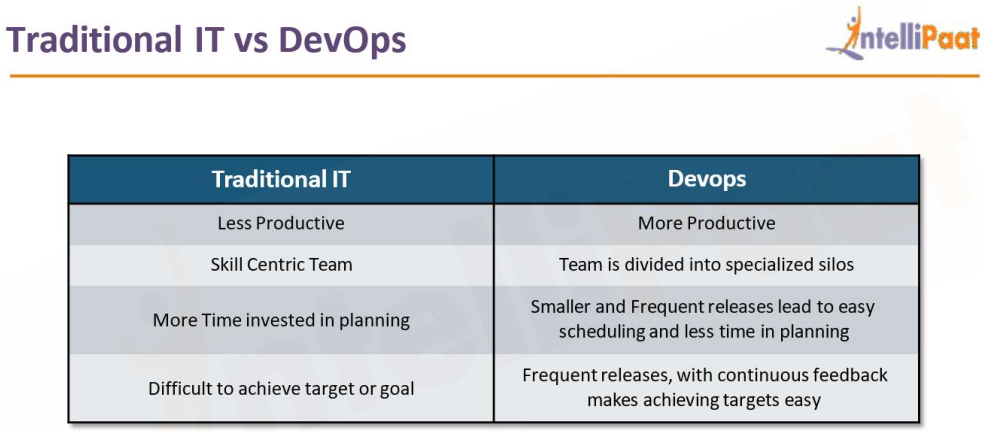
This led to a lot of back and forth between the developer and the operations team, hence impacted efficiency.

This problem was solved using Devops!

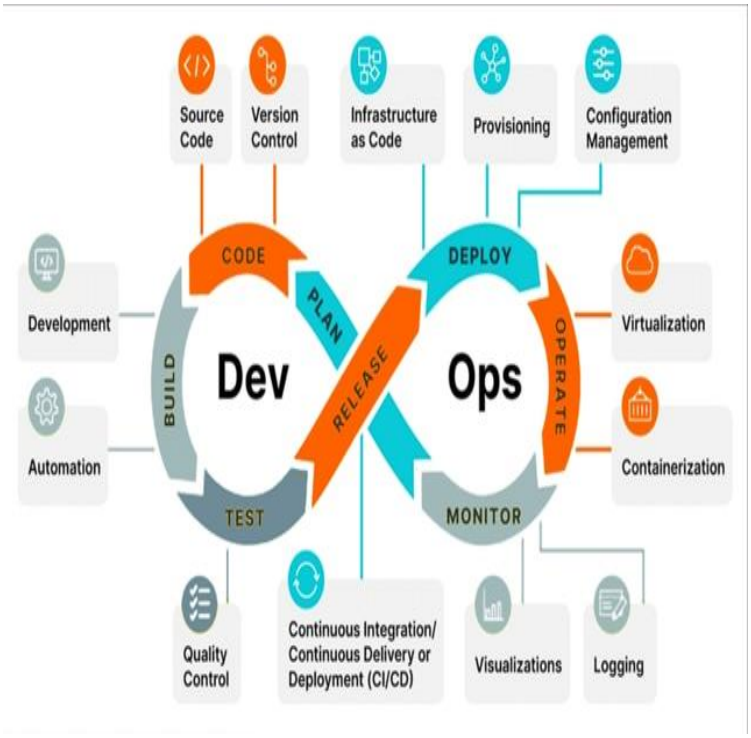
1. **Faster Releases:**
    - *Without DevOps*: Development and operations teams work separately. Developers write code and throw it over to operations, causing delays in testing and deployment.
    - *With DevOps*: Both teams collaborate from the start, using automation to build, test, and deploy code quickly. Releases happen faster and more frequently.

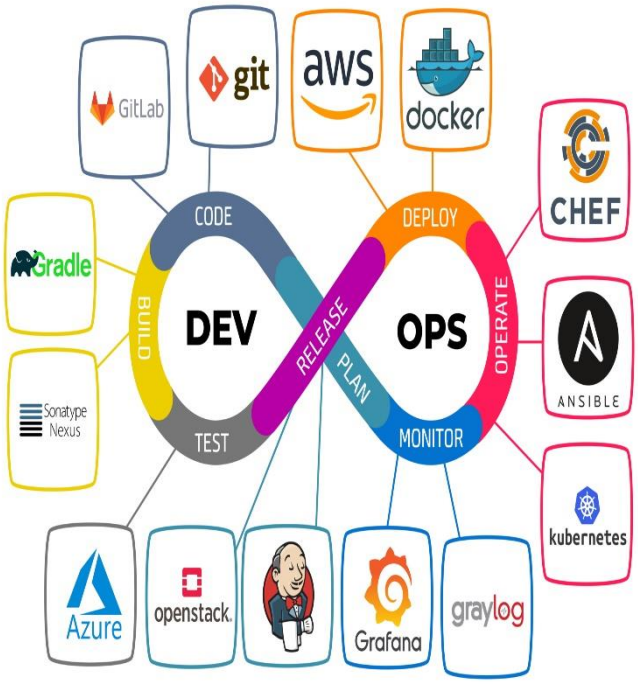
2. **Bug Fixing and Updates:**
    - *Without DevOps*: A bug fix might take days or weeks to go through the cycle of code, test, and deploy, slowing down the release.
    - *With DevOps*: Automated testing and integration allow quick detection and fixes of bugs, leading to faster deployment of updates.
3. **Consistency and Reduced Errors:**
    - *Without DevOps*: Manual deployments can lead to inconsistent environments and configuration errors.
    - *With DevOps*: Automated deployments ensure that all environments (development, testing, production) are consistent, reducing the risk of errors.

Traditional IT Vs Devops:

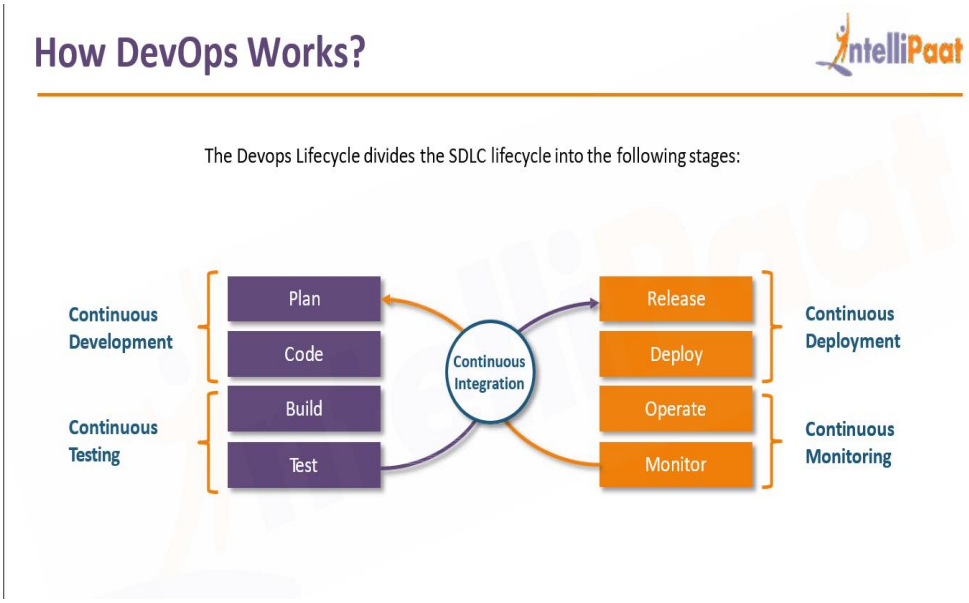


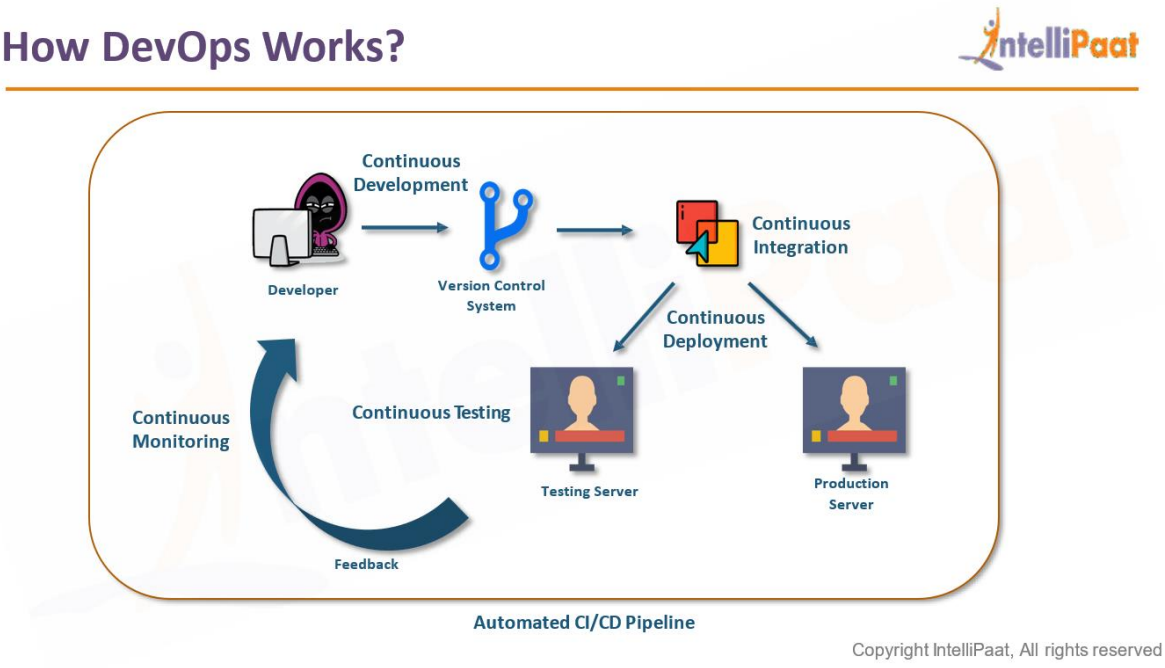
Traditional IT	Devops
Less Productive	More Productive
Skill Centric Team	Team is divided into specialized silos
More Time invested in planning	Smaller and Frequent releases lead to easy scheduling and less time in planning
Difficult to achieve target or goal	Frequent releases, with continuous feedback makes achieving targets easy





DevOps Lifecycles:





1.**Continous Development:**

This stage involves committing code to version control tools such as Git or SVN for maintaining the different versions of the code, and tools like Ant, Maven, Gradle for building/ packaging the code into an executable file that can be forwarded to the QAs for testing.

DevOps tools used:

Git is a distributed version-control system for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for source-code management in software development, but it can be used to keep track of changes in any set of files

2.**Continous Integration:**

This stage is the heart of the entire DevOps lifecycle. It is a software development practice in which the developers require to commit changes to the source code more frequently. This may be on a daily or weekly basis. Then every commit is built, and this allows early detection of problems if they are present. Building code is not only involved compilation, but it also includes **unit testing, integration testing, code review**, and **packaging**.

The four main stages of continuous integration are:

▪ Getting the Source Code from SCM: Fetching code from source control management systems.

▪ Building the Code: Compiling the code into executable units (e.g., WAR, JAR files).

▪ Code Quality Review: Ensuring the new code meets quality standards.

▪ Storing Build Artifacts: Saving the compiled outputs for deployment.

DevOps Tools:

Jenkins is an open-source automation server written in Java.

Jenkins helps to automate the non-human part of the software development process, with continuous integration

and facilitating technical aspects of continuous delivery.

3.**Continous Deployment:**

Automatically deploys tested code to production environments

Configuration management tools (e.g., **Puppet**, **Chef**, Ansible) automate deployment tasks and ensure consistency across different environments.

Containerization tools (e.g**., Docker,** Vagrant) maintain consistency in development, testing, and production environments by replicating the same dependencies and packages, reducing errors in production.

4.Continous Testing:

The stage deals with automated testing of the application pushed by the developer. If there is an error, the message is sent back to the integration tool, this tool in turn notifies the developer of the error. If the test was a success, the message is sent to Integration tool which pushes the build on the production server Involves automated testing to validate functionality and detect bugs early.

o Tools like **Selenium**, TestNG, and JUnit automate testing, saving time and effort and generating reports to simplify error evaluation.

o Continuous Integration tools (e.g., Jenkins) can automate the testing phase to ensure new code integrates well with existing code.

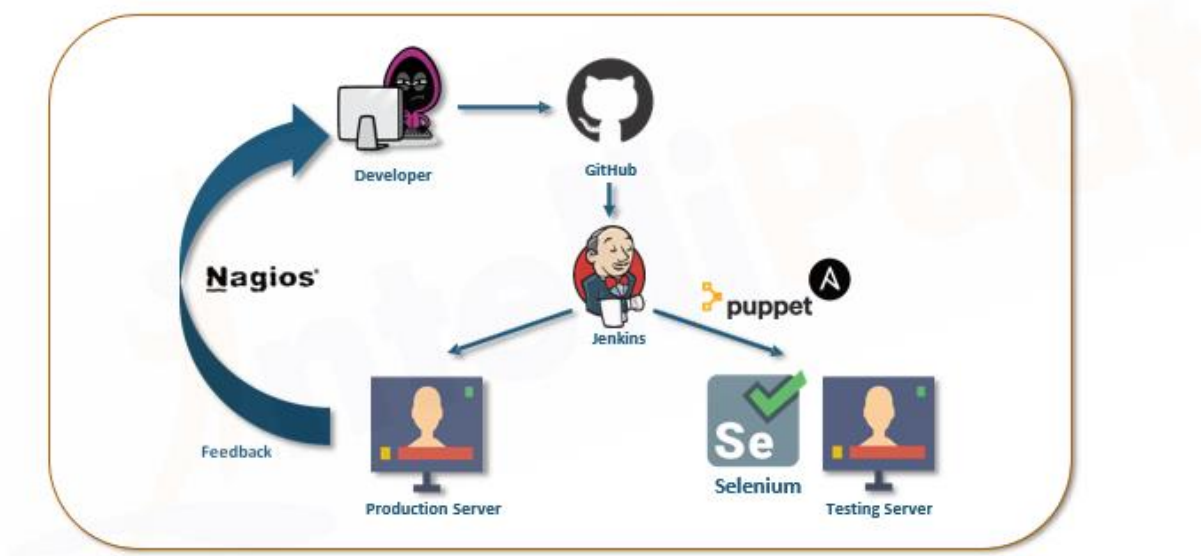
5.**Continous Monitoring:**

The stage continuously monitors the deployed application for bugs or crashes. It can also be setup to collect user feedback.

The collected data is then sent to the developers to improve the application

Tools:

Nagios is an open-source devops tool which is used for monitoring systems, networks and infrastructure. It also offers monitoring and alerting services for any configurable event.



##

2. Version Control System (VCS)

A Version Control System (VCS) is a tool used to manage changes in source code or files over time. It enables multiple people to collaborate on a project, tracks revisions, and allows rolling back to previous versions if necessary. There are two main types of version control systems: **local** and **distributed**.

Types of VCS:

1. **Local Version Control Systems:**
    - In a local VCS, all the versions of a project are stored on a single local system (e.g., a developer's computer). It is limited to single users and lacks collaboration features.
    - **Example:** Revision Control System (RCS).
2. **Centralized Version Control Systems (CVCS):**
    - In a CVCS, a single central server stores all the versioned files. Developers "check out" files from the server, make changes, and then "check in" the changes.
    - **Advantages:**
        - Easier collaboration.
        - All team members can access the latest version.
        - Admin can control user access.
    - **Disadvantages:**
        - Single point of failure (if the server crashes, everything is lost).
    - **Example:** Subversion (SVN).
3. **Distributed Version Control Systems (DVCS):**
    - In a DVCS, every contributor has a full copy of the project repository, including its entire history. Developers can commit changes locally and later synchronize their copy with others.
    - **Advantages:**
        - Faster operations (e.g., commit, diff).

        - No single point of failure; each contributor has a backup.
        - Better support for branching and merging.
    - **Example:** Git, Mercurial.

Core Features of VCS:

1. **Version Tracking:**
    - VCS tracks changes in files, providing a history of modifications and allowing the restoration of previous versions.
2. **Branching and Merging:**
    - Branching enables creating independent versions of the project to develop features, fix bugs, or experiment.
    - Merging integrates changes from different branches into a single branch.
3. **Collaboration:**
    - VCS allows multiple developers to work on the same project simultaneously without overwriting each other's changes. This is managed through branches and merging.
4. **Conflict Resolution:**
    - When multiple people work on the same file, conflicts may occur. VCS tools offer mechanisms to resolve conflicts, often by highlighting differences and letting users choose how to merge them.
5. **Backup and Recovery:**
    - Since VCS maintains a history of changes, it serves as a backup system. If a file is deleted or corrupted, you can restore an earlier version.
6. **Commit and Commit Messages:**
    - A commit in VCS is a save point in the project’s history. Developers can add descriptive messages that explain why the changes were made, helping others understand the project's evolution.

Advantages of Using VCS:

- **Improved Collaboration:** Team members can work simultaneously on the same codebase without stepping on each other's work.

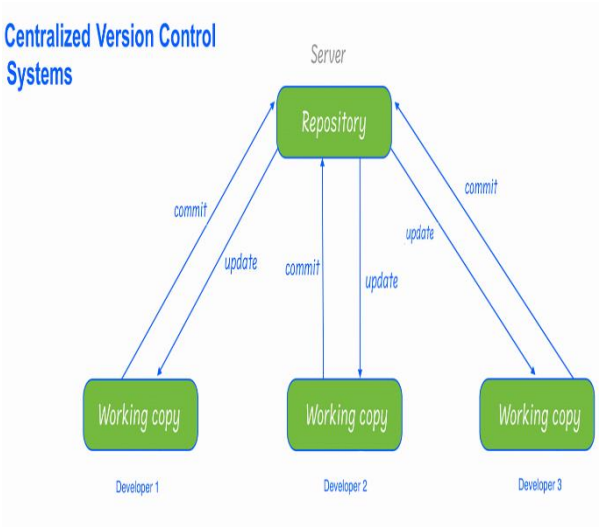
- **History and Accountability:** Every change is documented, so you can track who made changes and why.
- **Backup and Versioning:** VCS provides a safety net by allowing the rollback of code to previous versions.
- **Better Workflow:** VCS supports parallel development through branching, making it easier to develop new features or experiment without disrupting the main codebase.

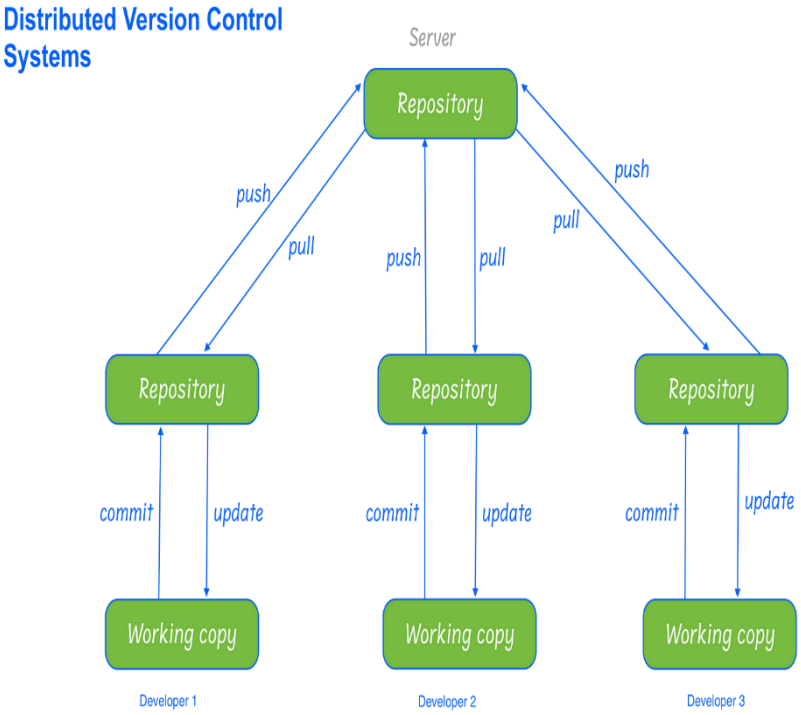
Popular VCS Tools:

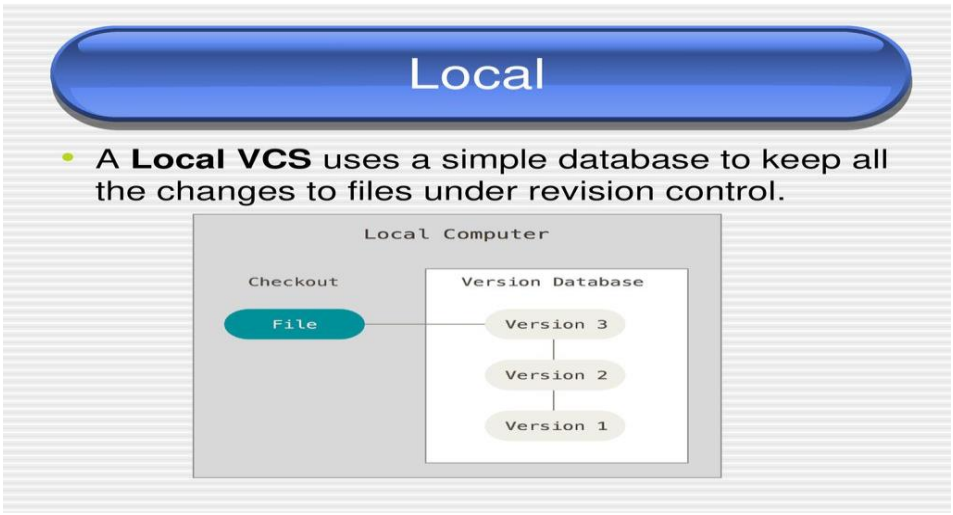
1. **Git:**
    - A widely used DVCS that supports distributed workflows, offers powerful branching and merging, and is used for both small and large projects.
    - **Example of Use:** Open-source projects like Linux Kernel.
2. **Subversion (SVN):**
    - A CVCS that stores everything in a central server but lacks the distributed feature of Git.
    - **Example of Use:** Corporate projects where centralized control is preferred.

Conclusion:

Version Control Systems are essential tools in modern software development, enabling collaboration, version tracking, and efficient project management. They provide structure to code management, prevent conflicts, and offer mechanisms for recovery and backup, ensuring smooth development even in large teams.


Centralized Version Control Systems
Server
Repository
commit
update
commit
update
commit
update
Working copy
Working copy
Working copy
Developer 1
Developer 2
Developer 3


Distributed Version Control Systems
Server
Repository
push
pull
push
pull
push
pull
Repository
Repository
Repository
commit
update
commit
update
commit
update
Working copy
Working copy
Working copy
Developer 1
Developer 2
Developer 3


Local
A Local VCS uses a simple database to keep all the changes to files under revision control.
Local Computer
Checkout
File
Version Database
Version 3
Version 2
Version 1

##

3. **What is Cloud Computing?**

Cloud computing is the delivery of computing services—such as servers, storage, databases, networking, software, and analytics—over the internet ("the cloud"). Instead of owning and maintaining physical hardware, organizations rent access to computing resources on-demand from cloud providers like Amazon Web Services (AWS), Microsoft Azure, or Google Cloud.

- **Example:** Instead of hosting your own servers to store data, you can use cloud services to store, manage, and access your data from anywhere via the internet.

Characteristics of Cloud Computing

1. **On-demand self-service:**
    - Users can access computing resources (like storage, servers, etc.) as needed without needing human intervention from the service provider.
    - **Example:** You can start using cloud storage at any time without asking for permission.
2. **Broad network access:**
    - Cloud services are available over the internet and can be accessed by various devices such as smartphones, laptops, and tablets.
    - **Example:** You can access files stored in Google Drive from your laptop or phone.
3. **Resource pooling:**
    - Cloud providers use a multi-tenant model where resources (like servers) are shared among many users, but each user's data is kept private and secure.
    - **Example:** Multiple businesses use the same cloud infrastructure, but their data remains separate.
4. **Rapid elasticity:**

- Cloud services can scale up or down quickly based on demand. If you need more storage or computing power, you can get it instantly.
- **Example:** If an app has a sudden spike in traffic, cloud resources can automatically scale to handle it.

5. **Measured service:**
   - Cloud systems automatically control and optimize resource usage, charging customers only for the resources they use (pay-as-you-go model).
   - **Example:** You pay for exactly how much storage or processing power you use, like a utility bill.

Cloud Deployment Models

1. **Public Cloud:**
   - Services are provided over the internet to the general public. Anyone can rent cloud services from a provider.
   - **Example:** Amazon Web Services (AWS), Google Cloud.
2. **Private Cloud:**
   - Cloud infrastructure is used exclusively by one organization, either hosted internally or by a third party. It offers more security and control.
   - **Example:** A company hosting its own private servers that only its employees can access.
3. **Hybrid Cloud:**
   - A mix of both public and private clouds, where data and applications can be shared between them. It provides flexibility by allowing businesses to keep sensitive data in a private cloud and use the public cloud for less critical tasks.
   - **Example:** A company storing sensitive customer data in a private cloud but using a public cloud for other non-sensitive tasks.
4. **Community Cloud:**
   - The cloud infrastructure is shared among several organizations with common concerns (e.g., security, compliance), often from a specific community.

- **Example:** Universities sharing a cloud to store and manage educational data.

Cloud Service Models

1. **Infrastructure as a Service (IaaS):**
    - Provides virtualized computing resources like servers, storage, and networks. It's the most basic cloud service.
    - **Example:** Renting virtual machines from AWS or Azure where you can install and run your own software and applications.
2. **Platform as a Service (PaaS):**
    - Provides a platform allowing developers to build and deploy applications without worrying about the underlying infrastructure.
    - **Example:** Google App Engine, which allows developers to deploy apps without managing the hardware.
3. **Software as a Service (SaaS):**
    - Delivers fully functional software over the internet that users can access on a subscription basis.
    - **Example:** Gmail, Google Docs, and Microsoft Office 365, where you use software via a web browser without installing it on your computer.

Conclusion:

Cloud computing is transforming how businesses use and manage their IT infrastructure. It offers flexibility, cost savings, and scalability by allowing companies to rent computing resources over the internet rather than owning physical hardware. Different deployment models and service models allow organizations to choose the best fit for their needs, from basic infrastructure (IaaS) to complete software solutions (SaaS).

##

4. **What is AWS (Amazon Web Services)?**
   AWS (Amazon Web Services) is a cloud computing platform provided by **Amazon**. It offers a wide range of cloud services, including **computing power, storage, databases, networking, machine learning, and analytics**, available on a pay-as-you-go basis. AWS enables businesses and developers to run applications, store data, and scale infrastructure without having to invest in expensive physical hardware.

   Principles AWS Works On:
   Here write characteristics of above cloud computing question

---

5. **What are Microservices?**

Microservices, or the **microservice architecture**, is a software development approach where an application is broken down into small, loosely coupled services. Each service performs a specific business function, operates independently, and can be developed, deployed, and scaled without affecting the entire system. These services communicate through APIs, typically over HTTP.

- Instead of building a monolithic application where all the functionality is tightly integrated into a single codebase.
- Microservices break down the application into smaller, loosely coupled services.

How Do Microservices Work?

1. **Independent Services:**
   - Each microservice is responsible for a single business function (e.g., user authentication, product catalog functions). These services can be developed and maintained independently by different teams, creating a **modular structure**.
   - **Example:** In an e-commerce app, the "User Account" service manages login and registration, while the "Product Catalog" service handles

displaying items for sale. If a new feature is needed for user profiles, it can be updated independently without affecting the catalog or payment systems.

2. **Modular Structure:**
    - Microservices allow developers to divide an application into smaller, manageable modules. This modular approach makes the system easier to understand, modify, and extend. Each module or service is responsible for a specific function, making updates and changes simpler and less risky.
    - **Example:** The "Cart Service" only handles the shopping cart, while the "Order Service" deals with finalizing the order. If the cart functionality needs improvement, it can be worked on without touching the order processing system.
3. **Communication via APIs:**
    - Microservices communicate with each other over APIs (like REST or gRPC), ensuring that the services remain independent while working together to complete business tasks.
    - **Example:** The payment service can make API calls to the inventory service to confirm available stock before processing a transaction.
4. **Autonomous Deployment and Continuous Improvement:**
    - Microservices enable **continuous improvement** because each service can be updated or enhanced independently. This allows developers to release new features or bug fixes quickly, without waiting for updates to the entire application.
    - **Example:** If the login service requires a security update, the development team can roll out the changes for that service only, without affecting the rest of the system.
5. **Decentralized Data Management:**
    - Each microservice can use its own database or storage, providing flexibility in how data is managed. This also enables the use of different types of databases for different services, a practice known as **polyglot persistence**.
    - **Example:** A service managing customer orders may use a relational database, while a service for user preferences might use a NoSQL database for faster data retrieval.

6. **Scalability:**
    - Microservices can be scaled independently. If one service experiences higher demand, only that service can be scaled, ensuring efficient resource usage and cost savings.
    - **Example:** During a high-traffic sale, the checkout microservice can be scaled up to handle increased order volume, while other services like user profiles remain unchanged.

Conclusion:

Microservices offer a flexible, scalable, and modular approach to software development. The architecture allows for **continuous improvement** by enabling frequent updates to individual services. It also promotes a **modular structure**, making applications easier to build, maintain, and scale. This makes microservices an ideal solution for large, complex systems that require regular updates and enhancements.

---

6. Components of Microservices Architecture:

1. **API Gateway:**
    - The API Gateway is the entry point for all requests coming from users. It directs the requests to the right microservice and handles tasks like checking if the user is allowed to access the service, balancing traffic, and monitoring the services.
2. **Service Discovery:**
    - This component helps microservices find each other. Instead of manually entering addresses, services register themselves, and when another service needs to communicate with them, it automatically finds the right one.
3. **Database per Service:**
    - Each microservice has its own database, meaning it controls its own data. This ensures each service can operate independently and doesn't interfere with other services' data.

4. **Inter-Service Communication:**
    - Microservices communicate with each other using simple methods like sending requests over the web (HTTP/REST) or using messaging systems. This allows them to work together without being too tightly connected.
5. **Load Balancer:**
    - The load balancer splits incoming requests evenly between multiple copies of a service, preventing any one service from becoming overwhelmed with too much traffic.
6. **Containers (e.g., Docker):**
    - Containers package each microservice with everything it needs to run, making it easy to deploy the same service across different environments without worrying about setup problems.
7. **Monitoring and Logging:**
    - Monitoring tracks the health and performance of services, while logging keeps records of what services are doing. This helps identify and fix issues quickly when something goes wrong.
8. **CI/CD Pipelines:**
    - Continuous Integration and Continuous Deployment pipelines automatically test, build, and deploy microservices. This makes it easier and faster to update services with new features or bug fixes.
9. **Security:**
    - Each service manages its own security, such as checking if users are allowed to access it and securing communication between services to protect sensitive information.
10. **Service Mesh:**

- A service mesh handles the communication between microservices. It manages things like traffic control, security, and making sure everything runs smoothly, without requiring developers to write extra code for these functions.

7. **Role of Microservices in DevOps (Headings Only)**

1. Faster Development and Deployment
2. Scalability and Flexibility
3. Independent Teams and Ownership
4. Continuous Improvement
5. Automation and CI/CD Pipelines
6. Resilience and Fault Isolation
7. Monitoring and Logging

Expand on your own

---

8. **Cloud computing Architecture & virtualisation Architecture**

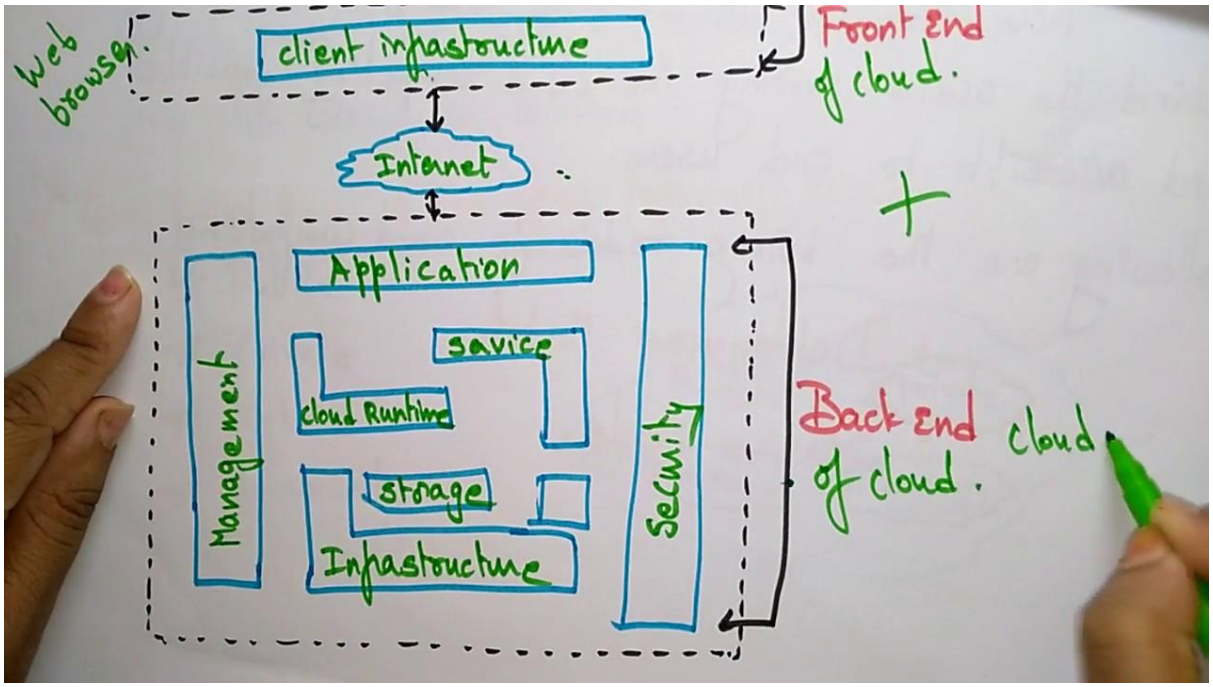
Write what is cloud computing and its characteristics.

Cloud Architecture

1. **Front-end Layer**:
    - **Clients**: Devices like desktops, laptops, tablets, and smartphones that access cloud services.
    - **User Interface**: Web browsers or applications that provide access to cloud resources.
2. **Back-end Layer**:
    - **Cloud Servers**: These include various servers that provide storage, applications, and services.
    - **Data Storage**: Databases, data lakes, or file storage systems that store user data and application information.
    - **Application Services**: Microservices and APIs that provide business logic and services to applications.
3. **Network Layer**:

- **Internet**: The connectivity that allows users to access cloud services.
- **Load Balancers**: Distribute network or application traffic across multiple servers to ensure reliability and performance.

4. **Management Layer**:
   - **Resource Management**: Tools for managing cloud resources, monitoring performance, and scaling services.
   - **Security**: Implements security measures like identity management, encryption, and compliance.
5. **Service Models**:
   - **IaaS (Infrastructure as a Service)**: Provides virtualized computing resources over the internet.
   - **PaaS (Platform as a Service)**: Offers a platform allowing developers to build, deploy, and manage applications.
   - **SaaS (Software as a Service)**: Delivers software applications over the internet, on a subscription basis.

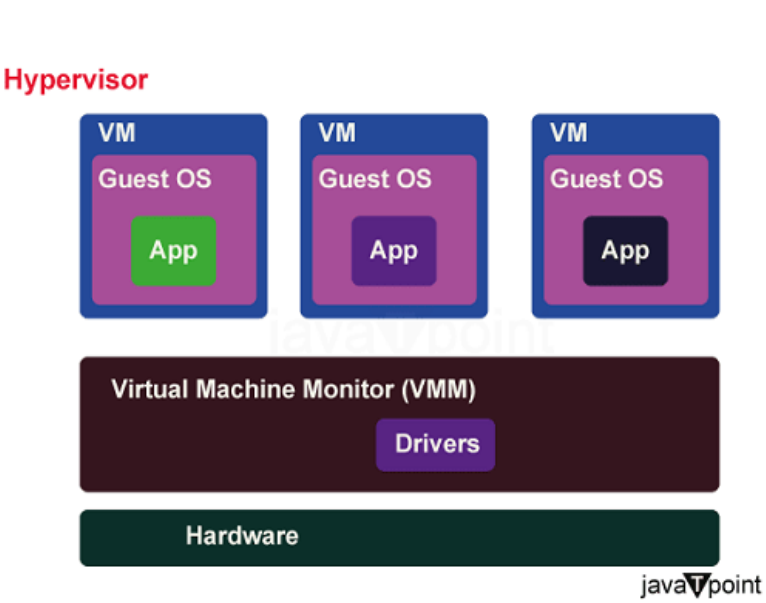


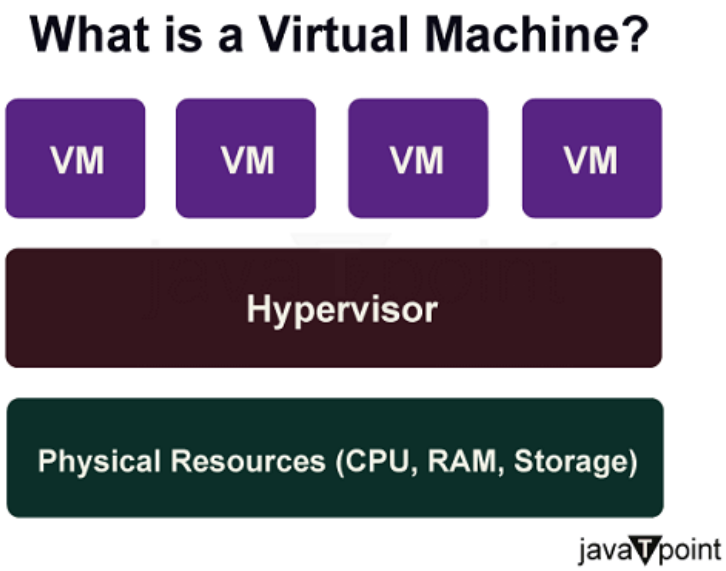
Virtualization is a technology that allows the creation of multiple simulated (virtual) environments or resources from a single physical system. It abstracts physical hardware to enable several virtual machines (VMs) to run independently on the same physical infrastructure.

Example: Using virtualization, a physical server can run multiple virtual machines, each with its own operating system, such as Windows and Linux.

Virtualization Architecture

1. **Physical Layer**:
    - **Physical Servers**: The actual hardware that hosts virtual machines (VMs).
2. **Hypervisor Layer**: A hypervisor is software that creates and manages virtual machines by abstracting the underlying physical hardware.
    - **Type 1 Hypervisor**: Runs directly on the hardware (bare-metal) and manages VMs (e.g., VMware ESXi, Microsoft Hyper-V).
    - **Type 2 Hypervisor**: Runs on top of an operating system and manages VMs (e.g., VMware Workstation, Oracle VirtualBox).
3. **Virtual Machine Layer**:
    - **Virtual Machines**: Abstractions of physical machines, each with its own OS and applications, sharing the underlying hardware resources.
4. **Management Layer**:
    - **Virtual Machine Manager (VMM)**: Tools for managing and orchestrating VMs, resource allocation, and monitoring performance.
5. **Network and Storage Virtualization**:
    - **Network Virtualization**: Abstracts networking hardware to create virtual networks that can be easily managed and scaled.
    - **Storage Virtualization**: Combines multiple physical storage devices into a single logical unit, allowing for efficient storage management.





---

9. What is Containerization ?

- **Containerization** is a technology that packages applications and their dependencies into containers. These containers encapsulate everything needed to run the application, including code, runtime, libraries, and system tools, ensuring consistency across different environments.

- It ensures consistency across different environments (development, testing, production).
- Containers run isolated from each other but share the host operating system.
- They start quickly and use resources efficiently.

Example

Docker is a common tool for creating and managing containers.

---

10. Docker

- Docker is a containerization platform that allows developers to package applications and their dependencies into lightweight, portable containers. These containers encapsulate everything needed to run the application, including code, runtime, libraries, and system tools, ensuring consistency across different environments.

What is Git?

Git is a distributed version control system used to track changes in source code during software development. It allows multiple developers to collaborate on a project by managing code changes, branching, merging, and maintaining a complete history of all changes.

Git vs GitHub

- **Git**:
    - A local tool for version control.
    - Manages your source code and tracks changes on your machine.
    - Used for branching, merging, and maintaining a project's history.
- **GitHub**:
    - A web-based platform that hosts Git repositories.
    - Allows for remote collaboration, code sharing, and project management.
    - Provides features like pull requests, issues tracking, and social coding (forking, following, stars).

Feature	Git	GitHub
Purpose	Version control on local system	Online hosting for Git repositories
Type	Command-line tool	Web-based platform
Usage	Track code changes	Collaborate, review, and store code online
Collaboration	Local, peer-to-peer	Centralized, remote collaboration

Git Workflow/Lifecycle

The Git workflow consists of several key stages that manage how code changes move through a project. Here's an overview of the typical Git lifecycle:

1. **Working Directory**:
    - The folder where your project resides. Changes made here are considered untracked by Git until they are added.
2. **Staging Area**:
    - A temporary area where changes are stored before committing. Files are "staged" using the git add command.
    - It allows you to prepare and review changes before finalizing them.
3. **Git Repository (Local)**:
    - Once changes are staged, they are committed to the local Git repository using the git commit command.
    - This creates a snapshot of the project’s state, which is stored in the repository's history.
4. **Remote Repository**:
    - If you're collaborating with others, you can push your local commits to a remote repository (e.g., GitHub) using the git push command.
    - The remote repository allows others to pull the latest changes and contribute.

Git Workflow Stages

1. **Untracked / Modified**: Any file in the working directory that hasn't been added to Git yet, or has been modified after the last commit.
2. **Staged**: Files added to the staging area, ready to be committed.
3. **Committed**: Changes stored in the local repository with a unique commit identifier.
4. **Pushed**: Changes uploaded to a remote repository for collaboration with other developers.

Commands in Git Workflow

- **git init**: Initialize a new Git repository.
- **git add [file]**: Add files to the staging area.
- **git commit -m "message"**: Commit staged changes with a message.
- **git push**: Push local commits to a remote repository.
- **git pull**: Fetch and merge changes from the remote repository.
- **git status**: Check the status of the working directory and staging area.
- **git clone [repo URL]**: Copy a remote repository to your local system.