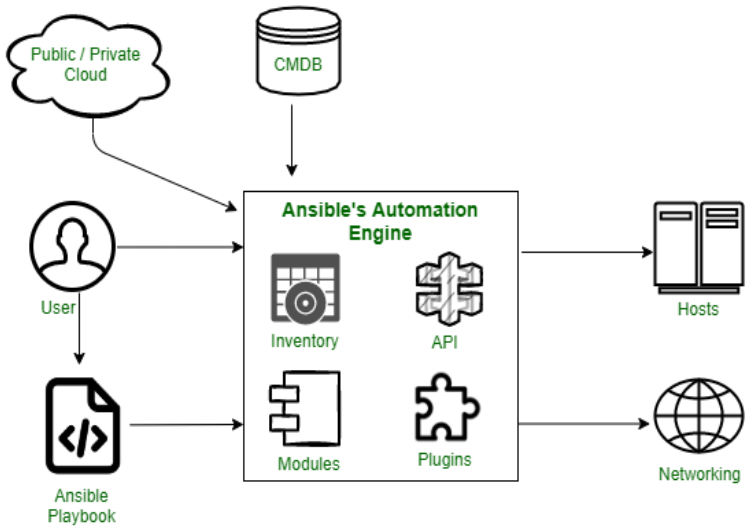
Set-1

1.List the core components of Ansible architecture.

Ansible Architecture components:



- **Inventories** – Ansible inventories are lists of hosts with their IP addresses, servers, and databases which have to be managed via an SSH for UNIX, Linux, or Networking devices, and WinRM for Windows systems.

- **APIs –** Application Programming Interface or APIs are used as a mode of transport for public and private cloud services.

- **Modules** – Modules are executed directly on remote hosts through playbooks and can control resources like services, packages, files, or execute system commands. They act on system files, install packages and make API calls to the service network. There are over 450 Ansible that provide modules that automate various jobs in an environment. For example, Cloud Modules like Cloud Formation create or delete an AWS cloud formation stack.

- **Plugins –** Plugins are pieces of code that augment Ansible's core functionality and allow executing Ansible tasks as a job build step. Ansible ships with several handy plugins and one can also write it on their own. For example, Action plugins act as front-ends to modules and can execute tasks on the controller before calling the modules themselves.

- **Networking –** Ansible uses a simple, powerful, and agent-less automation framework to automate network tasks. It uses a separate data model and spans different network hardware.

- **Hosts** – Hosts refer to the nodes or systems (Linux, Windows, etc) which are automated by Ansible.

- **Playbooks –**
  Playbooks are simple files written in YAML format which describe the tasks to be executed by Ansible. Playbooks can declare configurations, orchestrate the steps of any manual ordered process and can also launch various tasks.

- **CMDB –**
  It stands for Configuration Management Database (CMDB). In this, it holds data to a collection of IT assets, and it is a repository or data warehouse where we will store this kind of data, and It also defines the relationships between such assets.

- **Cloud –**
  It is a network of remote servers hosted on the internet to store, manage and process data instead of storing it on a local server.

2. Explain the difference between a Docker image and a Docker container.

Difference between Docker Images and Containers

Docker Image	Docker Container
It is a blueprint of the Container.	It is an instance of the Image.
Image is a logical entity.	The container is a real-world entity.
Images are created only once.	Containers are created any number of times using an image.
Images are immutable. One cannot attach volumes and networks.	Containers change only if the old image is deleted and a new one is used to build the container. One can attach volumes, networks, etc.
Images do not require computing resources to work.	Containers require computing resources to run as they run with a Docker Virtual Machine.
To make a docker image, you have to write a script in a Dockerfile.	To make a container from an image, you have to run the “docker run <image>” command
Docker Images are used to package up applications and pre-configured server environments.	Containers use server information and a file system provided by an image in order to operate.
Images can be shared on Docker Hub.	It makes no sense in sharing a running entity, always docker images are shared.
There is no such thing as a running state of a Docker Image.	Containers use RAM when created and in a running state.
An image must not refer to any state to remove the image.	A container must be in a running state to remove it.

Docker Image	Docker Container
One cannot connect to the images as these images are like snapshots.	In this, one cannot connect them and execute the commands.
Sharing of Docker Images is possible.	Sharing of containers is not possible directly.
It has multiple read-only layers.	It has a single writable layer.
These image templates can exist in isolation.	These containers cannot exist without images.

Note:-Write Any 10 points in exam.

3. Describe how Kubernetes Services help in exposing pods to external or internal traffic.

Introduction to Kubernetes Services

In Kubernetes, **pods** are temporary and can be replaced or restarted at any time. Each pod has its own IP address, but that IP is not permanent. To provide a **stable way to access pods**, Kubernetes uses **Services**.

A **Service** in Kubernetes is an abstraction that defines a **logical set of pods** and a policy by which to **access them**—either internally (within the cluster) or externally (outside the cluster).

2.Types of Kubernetes Services

Kubernetes supports different types of services based on how and where you want the traffic to be routed:

- **ClusterIP (default)**: Exposes the service inside the cluster only.
- **NodePort**: Exposes the service on a port on each node, allowing external access.
- **LoadBalancer**: Exposes the service through an external cloud-based load balancer.
- **ExternalName**: Maps the service to an external DNS name (for accessing external services).

3. How Services Work

- Services **select pods using labels**.
- Kubernetes automatically creates a **DNS entry** for each service so that it can be accessed by name.
- A **service proxy (kube-proxy)** runs on each node to route traffic to the appropriate pod behind the service.

4. Example: Service YAML to Expose a Pod Internally

yaml
CopyEdit

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
```

```
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: ClusterIP
```

This service routes traffic from port 80 of the service to port 8080 of matching pods labeled app: my-app.

4.Explain Grafana API and Auto healing in detail.

1. Grafana API

The **Grafana API** is a set of HTTP endpoints that allows users to interact programmatically with Grafana, enabling automation and integration of various Grafana resources like dashboards, data sources, and users. Key features include:

1. **Authentication**: It uses **API keys** or **basic authentication** to secure access.
2. **Dashboard Management**: You can create, update, and retrieve dashboards via API endpoints. This is useful for automating dashboard setup.
3. **Datasources & Alerts**: It allows management of data sources and alert configurations. For example, you can add a new data source or set up alert rules.

Example:

- To create a new dashboard, you can send a **POST** request to /api/dashboards/db with the dashboard's JSON configuration.

The Grafana API helps in automating monitoring setups, integrating Grafana with CI/CD pipelines, and managing resources programmatically.

2. Auto Healing in Kubernetes

Auto healing in Kubernetes refers to the system's ability to automatically detect and recover from failures, ensuring the application remains in the desired state without manual intervention. Key components include:

1. **ReplicaSets**: Ensures that the desired number of Pod replicas are running. If a Pod fails, it automatically creates a new one to maintain the number of replicas.
2. **Health Checks**: Kubernetes uses **liveness** and **readiness probes** to monitor Pod health. If a Pod fails a liveness probe, it is restarted automatically.

3. **Pod Disruption Budgets (PDB)**: Controls how many Pods can be disrupted during voluntary actions (like rolling updates), ensuring application availability.

Example: If a web server Pod crashes, the **ReplicaSet** will automatically launch a new Pod to replace it, ensuring high availability.

5.Write a Selenium test script snippet that checks if a login form returns an error on invalid credentials.

Below is a simple **Selenium WebDriver** test script snippet in **Python** that checks if a login form returns an error when invalid credentials are entered. This script assumes you have a basic login form with fields for username and password, and an error message is displayed when the credentials are invalid.

Prerequisites:

- Install Selenium: pip install selenium
- Ensure you have the appropriate **WebDriver** (e.g., ChromeDriver) for your browser.

Selenium Test Script for Invalid Login:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
import time

# Set up WebDriver (using Chrome in this example)
driver = webdriver.Chrome(executable_path='/path/to/chromedriver')  # Update with your ChromeDriver path

# Open the login page
driver.get('https://yourwebsite.com/login')  # Replace with your login URL

# Find the username and password fields
username_field = driver.find_element(By.ID, 'username')  # Update with the correct element ID
password_field = driver.find_element(By.ID, 'password')  # Update with the correct element ID

# Find the submit button (Assuming it's a button with id 'login_button')
login_button = driver.find_element(By.ID, 'login_button')  # Update with the correct element ID

# Input invalid credentials
username_field.send_keys('invalid_user')  # Invalid username
password_field.send_keys('wrong_password')  # Invalid password

# Submit the form
login_button.click()

# Wait for a moment to allow the error message to appear
time.sleep(2)
```

```
# Check if the error message is displayed (Assuming the error message has a specific class 'error')
error_message = driver.find_element(By.CLASS_NAME, 'error')  # Update with the correct class or selector

# Assert if the error message is present
assert error_message.is_displayed(), "Error message is not displayed for invalid login credentials."

# Print success message
print("Test passed: Error message displayed for invalid credentials.")

# Close the browser
driver.quit()
```

Explanation:

1. **WebDriver Setup**: The script uses Chrome WebDriver (webdriver.Chrome()), but you can change it to Firefox or another browser as needed.
2. **Login Form Interaction**:
    - It locates the username and password fields, then inputs invalid values into these fields.
    - The script submits the form by clicking the login button.
3. **Error Message Verification**:
    - After the form submission, the script waits for 2 seconds to allow time for the error message to appear (adjust time based on actual form response time).
    - It checks whether an error message (identified by its class name error) is displayed.
4. **Assertion**: The assert statement checks if the error message is visible. If it's not, the test will fail, and an assertion error will be raised.
5. **Cleanup**: After the test completes, driver.quit() ensures the browser is closed.

SET-2

1.Explain the purpose of an inventory file in Ansible and describe the types of inventories it supports

Purpose of an Inventory File in Ansible

In Ansible, an **inventory file** defines the list of hosts (machines or devices) that Ansible will manage and automate tasks on. It specifies which servers to target for executing playbooks, commands, or other tasks. The inventory file can also define host groups and associated variables, making it easier to organize hosts into logical units (e.g., web servers, database servers). This file allows Ansible to know where and how to run tasks across a network of machines.

Types of Inventories Supported in Ansible

1. **Static Inventory**:
   - A **static inventory** is a simple, manually maintained text file (usually in INI or YAML format) that lists hosts and groups of hosts. It is easy to set up for small infrastructures where the list of managed hosts doesn't change frequently.
   - **Example**: A typical static inventory could look like this:

     ```
     [web_servers]
     web1.example.com
     web2.example.com

     [db_servers]
     db1.example.com
     ```

2. **Dynamic Inventory**:
   - A **dynamic inventory** is generated by a script or plugin that queries an external system (like AWS, GCP, or Azure) to fetch the list of hosts in real-time. This is ideal for environments where hosts are frequently added or removed, like cloud-based or containerized systems.
   - **Example**: Using a plugin like aws_ec2 to dynamically fetch EC2 instances:

     ```
     ansible-inventory -i aws_ec2.yml --list
     ```

3. **Hybrid Inventory**:
   - A **hybrid inventory** combines both static and dynamic inventory sources. This is useful in scenarios where part of the infrastructure is static (e.g., on-premises servers) and part is dynamic (e.g., cloud instances).
   - **Example**: A hybrid setup could include both a static file and a dynamic script or plugin to manage different types of hosts.

2. Write and define the command for creating a container, start a container and stop a Container?

Here are the commands for creating, starting, and stopping a Docker container:

1. Creating a Container

To create a container from an image, use the docker run command. This command not only creates the container but also starts it. The general syntax is:

docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

- IMAGE: The Docker image you want to use to create the container.
- [OPTIONS]: Various options you can provide (e.g., port mapping, volume mounting, etc.).

Example:

To create a container from the nginx image:

docker run --name my_nginx -d -p 8080:80 nginx

- --name my_nginx: Assigns the container the name my_nginx.
- -d: Runs the container in detached mode (in the background).
- -p 8080:80: Maps port 8080 on the host to port 80 inside the container (useful for accessing the container's web service).
- nginx: The image name (Docker will pull this from Docker Hub if not present locally).

2. Starting a Container

To start a container that has already been created (but is stopped), use the docker start command.

bash
CopyEdit
docker start [OPTIONS] CONTAINER

- CONTAINER: The name or ID of the container you want to start.

Example:

To start the container named my_nginx:

docker start my_nginx

This will start the container that was previously created with the docker run command.

3. Stopping a Container

To stop a running container, use the docker stop command.

bash
CopyEdit

docker stop [OPTIONS] CONTAINER

- CONTAINER: The name or ID of the container you want to stop.

Example:

To stop the container named my_nginx:

docker stop my_nginx

This command will stop the my_nginx container if it's running.

Note:-you can also write docker container creating command in lab manual line.
3.Write a basic YAML manifest to deploy an Nginx pod in a Kubernetes cluster and expose it using a ClusterIP service.

3.Basic YAML Manifest for Nginx Pod Deployment and Service (5 Marks)

Below is a YAML manifest to deploy an **Nginx Pod** in a Kubernetes cluster and expose it using a **ClusterIP service**.

Yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80

---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
```

```
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

Explanation :

1. **Deployment Resource**:
    - **apiVersion: apps/v1**: Specifies the API version for the Deployment.
    - **kind: Deployment**: Declares the resource type as a Deployment.
    - **metadata**: Provides a name for the deployment (nginx-deployment).
    - **spec**:
        - **replicas: 1**: Ensures that only one Nginx Pod is running.
        - **selector**: Matches the labels for selecting the Nginx Pods.
        - **template**: Defines the Pod template used to create Nginx Pods.
            - **containers**: Specifies the container details.
                - **image: nginx:latest**: Uses the latest Nginx image from Docker Hub.
                - **ports**: Exposes port 80 on the container.
2. **Service Resource**:
    - **apiVersion: v1**: Specifies the API version for the Service.
    - **kind: Service**: Declares the resource type as a Service.
    - **metadata**: Names the service nginx-service.
    - **spec**:
        - **selector**: Selects Pods with the label app: nginx to associate with the service.
        - **ports**:
            - **port: 80**: Exposes the service on port 80.
            - **targetPort: 80**: Forwards the traffic to port 80 inside the Nginx container.
        - **type: ClusterIP**: Exposes the service within the cluster, not externally.

4. What is a Data Source in Grafana? Which types of data sources can Grafana connect to?

What is a Data Source in Grafana?

In **Grafana**, a **data source** refers to the connection configuration that allows Grafana to fetch and visualize data from external systems or services. It acts as a bridge between Grafana and the underlying data storage (such as databases, monitoring systems, or cloud platforms), enabling users to query, process, and display the data on Grafana dashboards. Each data source has its own configuration and query language, allowing Grafana to interact with different types of data.

Types of Data Sources Grafana Can Connect To

Grafana supports a wide variety of data sources to accommodate different use cases. Below are the main types of data sources Grafana can connect to:

1. **Time-Series Databases**:
    - **Prometheus**: Used for storing time-series metrics, widely used in monitoring and alerting.

    - **InfluxDB**: A time-series database for high-volume data, commonly used in IoT and monitoring.
    - **Graphite**: A tool for storing and visualizing time-series data.
    - **OpenTSDB**: A distributed time-series database for monitoring large-scale systems.
2. **SQL Databases**:
    - **MySQL**: A popular relational database management system (RDBMS).
    - **PostgreSQL**: A powerful open-source RDBMS.
    - **SQL Server**: Microsoft's RDBMS used for enterprise applications.
3. **Cloud Services and APIs**:
    - **AWS CloudWatch**: Monitors AWS resources and applications.
    - **Google Cloud Monitoring (Stackdriver)**: For monitoring applications in Google Cloud.
    - **Azure Monitor**: Microsoft Azure's cloud monitoring service.
4. **NoSQL Databases**:
    - **Elasticsearch**: A search and analytics engine, often used for log and event data.
    - **MongoDB**: A NoSQL database designed for scalability and flexibility.
5. **Other Data Sources**:
    - **Loki**: A log aggregation system often used with Prometheus.
    - **Zabbix**: A network monitoring tool.
    - **CSV Files**: Grafana can read and visualize data from CSV files.

5.What is Test-Driven Development (TDD), What are the main steps involved in the TDD process?

What is Test-Driven Development (TDD)?

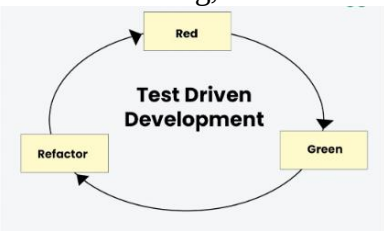
Test-Driven Development (TDD) is a software development approach where tests are written before writing the actual code. The primary goal of TDD is to ensure that code meets the specified requirements by continuously writing tests, implementing functionality, and refactoring code in small iterations. TDD encourages clean, maintainable, and error-free code by focusing on testing throughout the development process.

Main Steps Involved in the TDD Process

TDD follows a simple, structured cycle often referred to as the **Red-Green-Refactor** cycle, which includes the following main steps:

1. **Write a Test (Red)**:
    - First, you write a **test** for the functionality you plan to implement. This test typically focuses on a small, specific unit of the application.
    - Initially, the test will fail because the functionality has not been implemented yet.
    - The goal at this stage is to define the expected behavior of the code.
2. **Write the Code (Green)**:
    - Next, you write the **minimal code** necessary to make the test pass. This code should be as simple as possible to achieve the test's goal.
    - After writing the code, run the tests to ensure that the written code passes the test.
    - This step ensures that your implementation is focused on fulfilling the test's requirements.
3. **Refactor (Refactor)**:
    - Once the test passes, **refactor** the code to improve its structure and readability without changing its behavior.
    - Refactoring makes the code cleaner, more efficient, and easier to maintain.

- After refactoring, the test should still pass to ensure that no functionality was broken.



Set-3

1.Describe the difference between ad-hoc commands and playbooks in Ansible.

1.Definition and Purpose:

- **Ad-Hoc Commands**:
  - **Ad-Hoc commands** are used for quick, one-time tasks without needing to write a playbook. They are useful for executing simple tasks directly from the command line.
  - **Example (Installing Tomcat using Ad-Hoc Command)**: You can run an ad-hoc command to install Tomcat on a remote server:

    ansible all -m yum -a "name=tomcat state=present"

    This command installs Tomcat on all the servers defined in your Ansible inventory.

- **Playbooks**:
  - **Playbooks** are more structured and allow you to define a series of tasks to be executed on remote hosts. Playbooks are written in **YAML format** and are used for more complex tasks or workflows.
  - **Example (Installing and Configuring Tomcat using a Playbook)**:

```
---
- name: Install and configure Tomcat
  hosts: all
  become: yes
  tasks:
    - name: Install Tomcat package
      yum:
        name: tomcat
        state: present
    - name: Start Tomcat service
      service:
        name: tomcat
        state: started
    - name: Ensure Tomcat is enabled on boot
      service:
        name: tomcat
        enabled: yes
```

2. Execution and Flexibility:

- **Ad-Hoc Commands**:
  - Ad-Hoc commands are executed **immediately** from the command line, without the need for a playbook. They are suitable for simple, single-step tasks.
  - They are **not flexible** for more complex tasks or repetitive actions.
- **Playbooks**:
  - Playbooks allow for **multiple tasks** to be defined and executed in sequence. They support **loops**, **conditionals**, and **variables** for more complex and reusable workflows.
  - **Example**: You can write a playbook that installs Tomcat, configures it, and ensures it starts on boot — all in one place.

3. Use Cases:

- **Ad-Hoc Commands**:
  - Ideal for **quick, one-off tasks** such as installing a package, restarting a service, or checking the status of a service.
  - Example use case: Quickly install **Tomcat** on all remote hosts:

    ansible all -m yum -a "name=tomcat state=present"

- **Playbooks**:
  - Ideal for **automating complex, multi-step tasks** that need to be executed across multiple machines or environments. Playbooks are **reusable** and can be used for tasks like **Tomcat installation, configuration, and service management**.
  - Example use case: Install Tomcat, configure the system settings, and ensure that the Tomcat service is started and enabled on boot.

4. Reusability and Maintainability:

- **Ad-Hoc Commands**:
  - Ad-Hoc commands are **not reusable** in the traditional sense. Each time you need to perform a task, you must re-enter the command.
  - Example: If you want to install Tomcat on a new server, you would have to manually type out the ad-hoc command again.
- **Playbooks**:
  - Playbooks are **highly reusable** and can be saved for future use. You can use version control (e.g., Git) to manage and update your playbooks.
  - Example: The Tomcat installation and configuration playbook can be reused for multiple environments or servers by simply modifying the host inventory.

5. Complexity:

- **Ad-Hoc Commands**:

    - Simple and best for **single, straightforward tasks**. They don’t support complex logic such as loops, conditionals, or handling multiple steps in one execution.
- **Playbooks**:
    - Playbooks can handle **complex tasks** and sequences of steps. They are written in YAML, which makes them easy to read and manage. They can include logic, variables, and loops for more sophisticated workflows.
    - Example: A Tomcat playbook could not only install Tomcat but also configure Java, download application files, and start the Tomcat service with the required configuration.

2.Write a simple docker-compose.yml file to set up a web application with an Nginx container and a backend container.

version: '3.8'

services:

backend:

image: node:14

container_name: backend

working_dir: /app

volumes:

- ./backend:/app

command: npm start

ports:

- "5000:5000"

networks:

- webapp_network

nginx:

image: nginx:latest

container_name: nginx

volumes:

- ./nginx.conf:/etc/nginx/nginx.conf

- ./frontend:/usr/share/nginx/html

ports:

- "80:80"

depends_on:

- backend

networks:

- webapp_network

networks:

webapp_network:

(Below Explanation understanding purpose only no need to write)

Explanation

1. **Version**:
   - The version: '3.8' specifies the version of Docker Compose syntax to use.
2. **Backend Service**:
   - image: node:14: Uses the official Node.js image to run the backend service.
   - container_name: backend: Names the backend container backend.
   - working_dir: /app: Sets the working directory inside the container.
   - volumes: ./backend:/app: Mounts the local ./backend directory to /app in the container.
   - command: npm start: Runs npm start to start the Node.js application.
   - ports: "5000:5000": Maps port 5000 of the container to port 5000 on the host machine.
   - networks: webapp_network: Connects the backend container to a custom network.
3. **Nginx Service**:
   - image: nginx:latest: Uses the official Nginx image for the web server.
   - container_name: nginx: Names the Nginx container nginx.
   - volumes: ./nginx.conf:/etc/nginx/nginx.conf: Mounts a custom nginx.conf configuration file for Nginx.
   - volumes: ./frontend:/usr/share/nginx/html: Mounts the local ./frontend directory containing static files to the container.
   - ports: "80:80": Maps port 80 of the container to port 80 on the host machine.
   - depends_on: backend: Ensures Nginx starts after the backend service is up.
   - networks: webapp_network: Connects Nginx to the same network as the backend container.
4. **Networks**:
   - Defines a custom bridge network (webapp_network) to allow communication between the containers.

3.Compare the orchestration approaches of Docker Swarm and Kubernetes, and determine in which scenarios each would be more appropriate.

Comparison of Docker Swarm and Kubernetes (Table Format)

Feature	Docker Swarm	Kubernetes
Ease of Use	Simple to set up and use, integrated with Docker	Steeper learning curve, requires understanding of complex concepts
Scalability	Limited scalability, suitable for small-medium projects	Highly scalable, can manage large clusters and applications
High Availability	Basic fault tolerance, manual recovery	Advanced self-healing, auto-scaling, high availability
Networking	Simple overlay network, basic load balancing	Advanced networking, customizable, supports complex network policies and external load balancers
Ecosystem	Limited integrations and third-party tools	Rich ecosystem with integrations for monitoring, logging, CI/CD, etc.
Best Use Cases	Small to medium deployments, simpler applications	Large-scale applications, complex microservices, multi-cloud/hybrid-cloud environments

4.Explain how Prometheus collects metrics from infrastructure components.

1. Prometheus
Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability. It is widely used for collecting, storing, and querying metrics from various infrastructure components like servers, databases, applications, and networking devices.

2. Data Collection Method - Pull Model
Prometheus primarily uses a **pull-based model** to collect metrics. In this model:

- **Prometheus server** periodically scrapes metrics data from targets (infrastructure components) via HTTP endpoints.
- The components (or exporters) expose their metrics on specific **HTTP endpoints**, usually in **Prometheus exposition format**.
- Prometheus scrapes these endpoints at regular intervals to collect time-series data.

Example: Prometheus might scrape metrics from a web server by making HTTP requests to http://<target>/metrics, where the target is exposed by the infrastructure component.

3. Exporters
Infrastructure components generally do not expose metrics in a format Prometheus understands directly. To bridge this gap, **exporters** are used:

- **Exporters** are small programs or services that convert metrics from various systems (like operating systems, databases, or web servers) into a format Prometheus can scrape.
- For example:
    - **Node Exporter**: Exposes hardware and OS metrics (CPU, memory, disk, etc.) for Linux/Unix systems.
    - **MySQL Exporter**: Collects and exposes metrics from MySQL databases.

These exporters expose metrics on predefined HTTP endpoints (e.g., http://localhost:9100/metrics for Node Exporter).

4. Metric Scraping
Prometheus uses the **scraping mechanism** to collect data:

- **Scrape Interval**: Prometheus is configured to scrape metrics at specific intervals, for example, every 15 seconds.
- It pulls data from a set of **targets**, which can be:
  - Individual infrastructure components (e.g., servers, databases).
  - **Service discovery** mechanisms (to automatically find and track services in dynamic environments like Kubernetes).

Prometheus stores this data as **time-series metrics**, which are timestamped and organized by **labels** (such as instance, job, and region) to track different dimensions of the data.

5. Service Discovery
Prometheus can dynamically discover services in an environment, particularly in dynamic environments like cloud infrastructure or containerized environments. **Service discovery** allows Prometheus to find new targets automatically:

- **Kubernetes**: Prometheus can use Kubernetes API to discover containerized applications.
- **Consul/Etcd**: Service discovery for infrastructure components.
- This dynamic discovery ensures that Prometheus always scrapes relevant components, even in environments where services frequently change.

5.What is Selenium and what are its main components? Explainthe features of Selenium.

1. Introduction to Selenium
Selenium is a popular **open-source** automation tool for **web application testing**. It allows testers and developers to automate the process of interacting with web browsers, mimicking real user actions to validate the functionality of a website or web application. Selenium supports multiple programming languages like **Java**, **Python**, **C#**, **Ruby**, and more, allowing users to write scripts in their preferred language.

2. Main Components of Selenium
Selenium consists of four main components, each serving different roles in the automation process:

- **Selenium WebDriver**:
  WebDriver is the core component of Selenium that allows for browser automation. It interacts directly with the web browser, sends commands to control browser actions like clicking buttons, filling forms, and verifying page content. WebDriver supports multiple browsers such as Chrome, Firefox, Safari, and Edge.
- **Selenium IDE (Integrated Development Environment)**:
  Selenium IDE is a **record and playback** tool, primarily used for beginners. It is a browser extension (available for Firefox and Chrome) that allows users to record their interactions with a web application and then generate corresponding test scripts. While it's easier to use, it is less flexible compared to WebDriver.
- **Selenium Grid**:
  Selenium Grid enables parallel test execution on multiple machines and browsers simultaneously. It helps in **distributing tests** across various environments, reducing the overall execution time for large test suites. Grid allows running tests on remote machines, making it ideal for cross-browser testing.
- **Selenium Remote Control (RC)** (Deprecated):
  Selenium RC was one of the earlier components used for browser automation but has been deprecated in favor of WebDriver due to its limitations. RC required a special server to be running, which created extra complexity and performance issues.

3. Features of Selenium
Selenium provides several powerful features for web automation:

- **Cross-Browser Compatibility**:
  Selenium supports all major browsers, including Google Chrome, Mozilla Firefox, Safari, and Microsoft Edge. It allows users to write tests once and run them across different browsers to ensure cross-browser compatibility.
- **Language Support**:
  Selenium supports multiple programming languages like **Java**, **Python**, **C#**, **Ruby**, **JavaScript**, and **Kotlin**, making it flexible for developers and testers with varying technical backgrounds.
- **Platform Independence**:
  Selenium is platform-independent, meaning it can run on multiple operating systems, including Windows, macOS, and Linux. It integrates with various tools and frameworks, such as **TestNG**, **JUnit**, and **Maven**, to facilitate a wide range of testing needs.
- **Automated Interaction with Web Elements**:
  Selenium allows users to interact with web elements such as buttons, links, text boxes, and dropdowns, just like a human user would. This includes actions like clicking, typing, selecting, and verifying elements.
- **Support for Dynamic Web Applications**:
  Selenium is capable of handling dynamic web pages that rely on JavaScript, such as Ajax-based applications. It can wait for elements to load asynchronously, ensuring that tests are executed correctly even in complex web applications.
- **Integration with Other Tools**:
  Selenium can be easily integrated with various testing frameworks like **JUnit**, **TestNG**, and **Cucumber**. It also works with continuous integration tools like **Jenkins** and can be used in combination with tools like **Appium** for mobile testing and **Allure** for reporting.
- **Parallel Test Execution**:
  Selenium Grid allows parallel test execution on multiple machines, browsers, and platforms, which speeds up the testing process, especially for large test suites.

Set-4

1.Describe the Roles and modules in Ansible? Give some examples.

Introduction to Ansible
Ansible is an **open-source automation tool** used for configuration management, application deployment, task automation, and orchestration. It uses **simple YAML files** called playbooks to define tasks that automate the configuration of systems, networks, and applications.

1. Roles in Ansible
A **role** in Ansible is a way of organizing playbooks and related files in a structured manner. It is a modular and reusable component that allows for the logical grouping of tasks, variables, files, templates, and handlers.

- **Purpose of Roles**: Roles are designed to facilitate code reusability, modularity, and maintainability. By defining roles, you can divide a large playbook into smaller, more manageable sections, each performing a specific task (like setting up a web server or a database).
- **Structure of a Role**: Each role has a specific directory structure that includes the following key directories:
  - **tasks/**: Contains the main set of tasks (YAML files).
  - **vars/**: Contains variables used by the role.
  - **files/**: Contains static files that need to be copied to managed nodes.
  - **templates/**: Contains Jinja2 templates.
  - **handlers/**: Contains tasks triggered by events, such as service restarts.
  - **defaults/**: Contains default variables for the role.
- **Example of Using a Role**:
  To use a role in a playbook, you can simply reference it as follows:

  ```
  - hosts: webservers
    roles:
      - apache
  ```

  Here, apache is the role defined to set up an Apache web server.

2. Modules in Ansible
A **module** in Ansible is a discrete unit of work that Ansible executes. Modules perform tasks like installing packages, copying files, managing services, and more. Ansible provides a wide range of built-in modules, and users can also create custom modules if needed.

- **Purpose of Modules**: Modules are the building blocks of Ansible tasks. They allow Ansible to interact with systems and perform actions like modifying files, executing commands, installing software, and managing system services.
- **Types of Modules**:
  - **Core Modules**: These are modules that come with Ansible by default, such as yum, apt, copy, and service.
  - **Extra Modules**: These modules are maintained separately and can be installed from external collections or repositories. Examples include modules for cloud providers like ec2 for AWS or gce for Google Cloud.
- **Examples of Common Modules**:
  - **apt**: Installs, removes, or upgrades packages on **Debian-based systems**.

```
- name: Install nginx
  apt:
    name: nginx
    state: present
```

- **service:** Manages system services (start, stop, restart, etc.).

```
- name: Ensure nginx is running
  service:
    name: nginx
    state: started
```

  - **copy**: Copies files to remote machines.

```
- name: Copy the configuration file
  copy:
    src: /local/path/config.yml
    dest: /remote/path/config.yml
```

  - **user**: Manages user accounts on remote systems.

```
- name: Create a new user
  user:
    name: johndoe
    state: present
```

2.What is Docker, and what are its primary use cases in software development?

Introduction to Docker
Docker is an **open-source platform** that allows developers to automate the deployment, scaling, and management of applications in lightweight, portable containers. It uses containerization technology to package software and its dependencies into a standardized unit, ensuring that the application works seamlessly across different environments, from a developer's machine to production servers.

- **Containers** are isolated environments that run applications and services independently, with all required libraries, configurations, and dependencies bundled together. This solves the common problem of "works on my machine" by ensuring consistency across various systems.

1. Primary Components of Docker

- **Docker Engine**: The core component responsible for creating, running, and managing containers.
- **Docker Images**: Read-only templates that contain the application and its dependencies. Docker containers are created from images.
- **Docker Containers**: Lightweight, standalone, and executable packages created from Docker images. They encapsulate an application and all its dependencies.
- **Docker Hub**: A cloud-based registry service for sharing and storing Docker images.

2. Primary Use Cases of Docker in Software Development

- **Environment Consistency**:
  Docker ensures that applications run consistently across different environments (e.g., development, testing, staging, production). This eliminates issues related to environment discrepancies and dependency mismatches. Developers can package their applications and run them on any system with Docker, ensuring **environment consistency**.

  Example: A developer builds and tests an application on their local machine, and the exact same environment can be used in staging and production without changes.

- **Simplified Dependency Management**:
  Docker containers bundle the application and all of its dependencies (libraries, frameworks, configurations) together, ensuring that developers do not have to worry about installing and managing dependencies manually on each machine. This **dependency isolation** reduces conflicts between different software versions.

  Example: A Python application with specific library versions can be packaged in a Docker container, ensuring it runs with the same libraries across all environments.

- **Microservices Architecture**:
  Docker is widely used in **microservices architecture**, where each service (e.g., database, front-end, back-end) is encapsulated within its own container. This approach allows for scalable, modular development and easier management of independent services.

  Example: A complex application can be broken down into multiple Docker containers: one for the user interface, one for the back-end logic, and one for the database, all running and scaling independently.

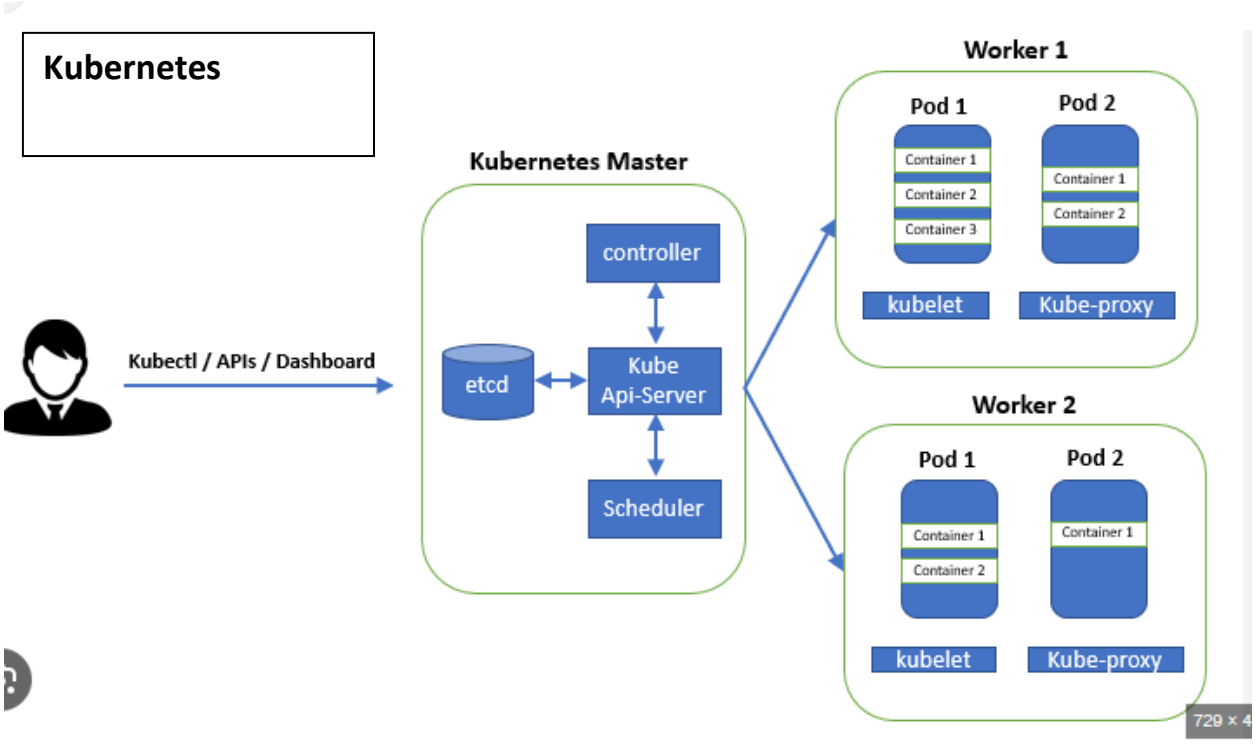
- **Continuous Integration and Continuous Deployment (CI/CD)**:
  Docker enables **automated testing, building, and deployment** in CI/CD pipelines. It helps in **faster delivery** of applications by ensuring consistency in every stage, from development to production. Docker containers can be spun up quickly for testing and deployment, making them ideal for **automation**.

  Example: In a CI/CD pipeline, Docker is used to run tests in isolated containers to ensure the application works correctly before it is deployed to production.

- **Resource Efficiency and Scalability**:
  Docker containers are **lightweight** compared to traditional virtual machines, making them resource-efficient. This allows developers to run more containers on the same hardware. Additionally, containers can be easily scaled up or down depending on demand, making Docker ideal for applications that need to scale efficiently.

  Example: In cloud environments, a Docker containerized application can automatically scale based on traffic, using orchestration tools like Kubernetes to manage the scaling process.

3.Explain the role of the Kubernetes Master Node and Worker Node.



Introduction to Kubernetes Architecture

Kubernetes is an open-source container orchestration platform used to automate the deployment, scaling, and management of containerized applications. The Kubernetes architecture consists of **two primary types of nodes**: the **Master Node** and the **Worker Nodes**. Each node has specific roles and responsibilities in managing the cluster and ensuring that the containerized applications are running effectively.

1. Role of the Kubernetes Master Node

The **Master Node** is the central control plane of a Kubernetes cluster. It is responsible for managing the overall cluster state, making global decisions about scheduling, monitoring, and controlling the system. It ensures the cluster's health and operation.

Key responsibilities of the Master Node include:

- **API Server**: The API Server (kube-apiserver) is the front-end of the Kubernetes control plane and exposes the REST API to interact with the cluster. It acts as the main entry point for users, administrators, and components to communicate with the Kubernetes cluster.
- **Controller Manager**: The Controller Manager (kube-controller-manager) runs controllers that regulate the state of the system. For example, the replication controller ensures that the correct number of pod replicas are running, and the deployment controller manages the deployment lifecycle of applications.
- **Scheduler**: The Scheduler (kube-scheduler) assigns newly created pods to the worker nodes based on resource availability, constraints, and policies. It determines which worker node should run the pod.
- **etcd**: The etcd database is a distributed key-value store that holds the cluster's state and configuration data. It is used for storing all the necessary data to maintain the cluster's configuration and state, such as pod details, deployment information, and more.
- **Cloud Controller Manager**: This component interacts with cloud provider APIs (if using a cloud environment) to manage services like load balancers, persistent volumes, and other cloud-specific resources.

2. Role of the Kubernetes Worker Node
The **Worker Node** is responsible for running the containerized applications (pods) and providing the necessary resources to support them. It communicates with the Master Node to receive instructions on what to do and reports back on the state of the pods running on it.

Key responsibilities of the Worker Node include:

- **Kubelet**: The kubelet is an agent that runs on each worker node. It ensures that the containers (pods) are running in the desired state, as specified by the Master Node. The kubelet communicates with the API server to get the desired state and manages the lifecycle of the pods and containers on its node.
- **Container Runtime**: The container runtime (e.g., Docker, containerd) is responsible for running and managing containers within the pods. It handles the starting, stopping, and execution of containers inside the pods on the worker node.
- **Kube Proxy**: The kube-proxy is a network proxy that maintains network rules on the worker node. It enables communication between services within the cluster by load balancing network traffic and managing routing to the correct pod.

4.Explain about PromQL with some example, and define time-series database.

What is PromQL?

PromQL (Prometheus Query Language) is a powerful query language used in **Prometheus** to retrieve and analyze time-series data stored in its database. It allows users to filter, aggregate, and calculate metrics for monitoring, alerting, and visualization purposes.

PromQL is designed specifically for working with **metrics data** collected over time (e.g., CPU usage, memory usage, HTTP requests, etc.). It supports functions like rate calculation, averages, and comparisons to help in real-time monitoring.

1. Examples of PromQL Queries

- **Basic Metric Query**

  http_requests_total

  This returns the total number of HTTP requests recorded.

- **Filtering by Labels**

  http_requests_total{method="GET", status="200"}

  This returns the number of successful GET requests.

- **Rate of Change Over Time**

  rate(cpu_usage_seconds_total[5m])

  This calculates the **per-second rate** of CPU usage over the last 5 minutes.

- **Average CPU Usage Across All Instances**

  avg(cpu_usage_seconds_total)

  This returns the **average CPU usage** across all instances.

- **Alerting Example**

  node_memory_Active_bytes / node_memory_MemTotal_bytes > 0.9

  This expression can be used to trigger an alert if memory usage goes above **90%**.

3. What is a Time-Series Database?

A **Time-Series Database (TSDB)** is a type of database optimized for storing and querying **time-stamped data**. Each data point in a time-series has:

- A **timestamp** (when it was recorded),
- A **metric name** (what is being measured),
- A **value** (the measurement),
- And optional **labels** (key-value pairs for identifying dimensions like host, region, etc.).

In Prometheus, the built-in time-series database stores all the collected metrics as time-series data, making it easy to query trends over time using PromQL.

5.What is REPL-driven development, How can REPL-driven development help in writing Selenium tests?

1. What is REPL-Driven Development?

REPL stands for **Read-Eval-Print Loop**. It is an interactive programming environment that:

- **Reads** the user's input (a line of code),
- **Evaluates** the input (executes the code),
- **Prints** the result,
- And **Loops** back for the next input.

REPL-driven development is a development approach where programmers use a REPL to write and test small chunks of code **interactively** and **iteratively**, rather than writing large blocks of code at once. This allows for **real-time feedback**, **faster debugging**, and **immediate testing of code logic**.

Languages like **Python**, **JavaScript**, and **Clojure** support REPL environments.

2. How REPL-Driven Development Helps in Writing Selenium Tests

Selenium is used for automating web browser interactions. REPL-driven development can make writing Selenium tests **easier**, **faster**, and **more reliable** in the following ways:

a) Instant Feedback for Browser Commands

Using REPL, testers can execute Selenium commands (like click(), find_element(), get()) one at a time and immediately see how the browser responds. This helps in:

- Verifying that elements are correctly located,
- Checking if interactions like clicking or typing work,
- Quickly experimenting with test steps before adding them to a full script.

Example:

```
>>> from selenium import webdriver
>>> driver = webdriver.Chrome()
>>> driver.get("https://example.com")
>>> driver.find_element("id", "login").click()
```

Each line is executed and tested immediately, making it easy to debug and refine.

b) Faster Test Development

REPL-driven testing avoids the need to run full test scripts repeatedly. Developers and testers can build tests step-by-step and only include steps that are confirmed to work, resulting in faster test creation.

c) Better Debugging

When a test step fails, you can try alternative commands right away in the REPL instead of modifying and re-running the whole script. This reduces the time spent debugging test scripts.

d) Learning and Exploration

New users of Selenium can use REPL to learn how Selenium works by trying commands interactively. It helps them understand how to locate elements and perform actions without writing full test cases.