**My DevOps Interview Questions:**

1. **How to get Notified or alerts if S3 storage bucket gets updated by an object creation or modification?**

We can set up **event notifications on Amazon S3 buckets** to receive alerts when objects are created, modified, or deleted. This can be achieved using Amazon S3 event notifications and services like Amazon Simple Notification Service (SNS) or AWS Lambda.

Create an S3 Event Notification:

* Go to the Amazon S3 Management Console.
* Select the bucket for which you want to receive notifications.
* Choose "Properties", then "Events".
* Click on "Add Notification".
* Configure the event type (e.g., "All object create events" or "All object delete events").
* Choose the destination for the event notification:
* If you created an SNS topic, select it.
* Alternatively, you can choose to invoke an AWS Lambda function directly.

1. **Do You know IAM if yes, what is IAM and what is the difference between Roles and Policies?**

Yes, I'm familiar with IAM (Identity and Access Management) in the context of AWS. IAM is a web service that helps you securely control access to AWS resources. It enables you to manage users, groups, roles, and their corresponding level of access to the AWS services and resources.

Roles and Policies are two key components of IAM, but they serve different purposes:

**Roles:**

* A role is an AWS IAM entity that defines a set of permissions for making AWS service requests.
* Roles are not associated with a specific user or group; instead, they are assumed by users, applications, or AWS services temporarily.
* For example, you might create a role that allows access to specific S3 buckets or EC2 instances and then assign that role to an EC2 instance. This allows the EC2 instance to access the specified resources without requiring hardcoded credentials.

**Policies:**

* A policy is a JSON document that defines the permissions (what actions are allowed or denied) for AWS resources.
* Policies can be attached to users, groups, or roles to define their access permissions.
* Policies can also be applied at the resource level, defining who can access a specific AWS resource.
* Policies can be inline (directly embedded within an IAM user, group, or role) or managed (standalone policies that can be attached to multiple users, groups, or roles).

1. **What are the Different types of Security levels in IAM & What are different types of policies in IAM?**

In AWS IAM (Identity and Access Management), security levels refer to the various layers of access control and permissions management that can be applied to users, groups, and roles. There are several levels of security in IAM, including:

**1. Root Account**:

* The root account is the AWS account that is created when you first sign up for AWS.
* It has completed administrative access to all resources and services within the account.
* Best practice is to avoid using the root account for everyday tasks and instead create IAM users with appropriate permissions.

**2. IAM Users**:

* IAM users are individual entities within your AWS account.
* They represent individual people, applications, or services that interact with AWS services.
* Permissions for IAM users are managed through policies.

**3. IAM Groups**:

* IAM groups are collections of IAM users.
* Permissions can be assigned to groups, and those permissions are then inherited by all users within the group.
* Groups are useful for managing permissions at scale, allowing you to apply permissions to multiple users simultaneously.

**4. IAM Roles**:

* IAM roles are similar to IAM users but are intended to be assumed by other AWS entities, such as EC2 instances, Lambda functions, or other AWS services.
* Roles are often used to grant temporary access to resources or services without the need for long-term credentials.
* Permissions for roles are also managed through policies.

Now, let's discuss the different types of policies in IAM. Policies in IAM are JSON documents that define permissions. There are three main types of policies:

**1. Managed Policies**:

* Managed policies are standalone policies that you can attach to multiple users, groups, or roles in your AWS account.
* They can be created and managed independently of the IAM entities to which they are attached.
* Managed policies can be AWS managed (created and maintained by AWS) or customer managed (created and maintained by you).

**2. Inline Policies:**

* Inline policies are policies that are directly embedded within a single IAM user, group, or role.
* They are defined within the configuration of the IAM entity to which they apply.
* Inline policies are useful when you want to keep the policy closely associated with a specific IAM entity.

**3. AWS Managed Policies:**

* AWS provides a set of pre-defined managed policies that cover common use cases, such as granting read-only access to specific services or providing full administrative access.
* These policies are maintained and updated by AWS.
* You can attach AWS managed policies directly to IAM users, groups, or roles in your account.

By leveraging these different types of policies, you can effectively manage permissions and access control within your AWS account.

1. **How can you define and give access to 10 users within the team?**

To define and give access to 10 users within a team in AWS IAM, you can follow these general steps:

1. **Create IAM Users**:

* Create IAM users for each member of your team. You can do this through the AWS Management Console, AWS CLI, or AWS SDKs.
* Ensure each user has a unique username and a secure password.

2. **Group Users** (Optional):

* If your users have similar roles or require similar permissions, you can group them together by creating IAM groups.
* For example, you might have groups like "Developers," "Operations," or "Administrators."

3. **Create Policies**:

* Define the permissions that each user or group requires using IAM policies.
* You can create managed policies or inline policies depending on your preference and requirements.

4. **Attach Policies to Users or Groups**:

* Attach the policies you created to the respective IAM users or groups.
* This grants them the necessary permissions to perform actions on AWS services and resources.

5. **Test Access**:

* Test the access for each user to ensure they can perform their intended actions and don't have access to resources they shouldn't.

**Creating IAM Users**: You can create IAM users through the AWS Management Console:

1. Sign in to the AWS Management Console and open the IAM console.
2. In the navigation pane, choose "Users," then "Add user."
3. Enter a username for each user and select the access type (programmatic access, AWS Management Console access, or both).
4. Set permissions for the user. You can either add the user to an existing group or attach policies directly.
5. Review the user details and choose "Create user."

**Grouping Users** (Optional):

1. In the IAM console, navigate to "Groups" and choose "Create group."
2. Enter a name for the group and optionally, a group description.
3. Select the policies that define the permissions for the group.
4. Review the group details and choose "Create group."

**Creating Policies**: You can create policies through the IAM console:

1. In the IAM console, navigate to "Policies" and choose "Create policy."
2. Choose between creating a JSON policy or using the visual editor.
3. Define the permissions for the policy based on your requirements.
4. Review and validate the policy, then choose "Create policy."

**Attaching Policies to Users or Groups**:

1. Select the user or group in the IAM console.
2. Choose the "Permissions" tab.
3. Attach policies by choosing "Add permissions" or "Attach policies."
4. Select the policies you want to attach and choose "Next: Review."
5. Review the permissions and choose "Add permissions" or "Attach policy."
6. **Do you know Terraform; How can you create different types of instances in one Terraform file?**

Yes, I'm familiar with Terraform. Terraform is an infrastructure as code (IaC) tool that allows you to define and manage your infrastructure using declarative configuration files. You can use Terraform to create and manage various resources in different cloud providers, including AWS, Azure, Google Cloud, and more.

To create different types of instances in one Terraform file, you can define multiple resource blocks for each type of instance you want to create.

# Define AWS provider

provider "aws" {

region = "us-west-2"

}

# Define an EC2 instance

resource "aws\_instance" "web\_server" {

ami = "ami-12345678"

instance\_type = "t2.micro"

tags = {

Name = "web-server"

}

}

# Define another EC2 instance with a different instance type

resource "aws\_instance" "database\_server" {

ami = "ami-87654321"

instance\_type = "t2.large"

tags = {

Name = "database-server"

}

}

* We first define the AWS provider with the region where our resources will be created.
* Then, we define two different EC2 instances using the aws\_instance resource block. Each instance has its own configuration, including AMI (Amazon Machine Image) ID, instance type, and tags.
* The aws\_instance resource block allows you to define multiple instances with different configurations within the same Terraform configuration file.

1. **let's say, there are multiple servers and instances are programmed in one single Terraform file. How can you execute a particular service or instance?**

In Terraform, you can target specific resources or instances for execution using the -target flag with the terraform apply command. This allows you to selectively apply changes to only the specified resources while excluding others. Ex: terraform apply -target=aws\_instance.web\_server.

1. **How can you resolve the issue if the instance suddenly disconnected from AWS CLI even the instance is running state?**

If an instance suddenly disconnects from the AWS CLI (Command Line Interface), even though the instance is in a running state, it could be due to various reasons such as network issues, SSH configuration problems, or instance-specific issues. Here are some steps you can take to troubleshoot and resolve the issue:

**1. Check Instance Status in AWS Console:**

Log in to the AWS Management Console and navigate to the EC2 dashboard.

Verify that the instance is indeed in a running state and has not encountered any issues or errors.

**2. Check Network Connectivity:**

Ensure that the instance has proper network connectivity by checking its security group and network ACL settings.

Verify that the instance has a public IP address (if it's in a public subnet) and that there are no restrictions preventing inbound or outbound traffic.

**3. SSH Connection Troubleshooting**:

If you're trying to connect to a Linux instance via SSH, ensure that the SSH service is running on the instance and that the SSH port (default: 22) is open in the security group.

Check the SSH key pair used for authentication and ensure that it's correctly configured in your SSH client and associated with the instance.

If SSH fails, try connecting via the EC2 Instance Connect feature in the AWS Management Console, which allows you to connect to instances without using SSH keys.

**4. Instance Console Logs**:

View the instance console logs in the AWS Management Console to check for any error messages or issues reported by the instance during boot-up or operation.

**5. Restart the Instance**:

If the instance appears to be healthy but you're still unable to connect, try restarting the instance from the AWS Management Console or using the AWS CLI (`aws ec2 reboot-instances`) to see if it resolves the issue.

**6. Review CloudWatch Logs**:

Check CloudWatch Logs for any relevant log messages or metrics that might provide insights into the issue, such as system logs, application logs, or custom metrics.

**7. Review Instance Metadata:**

Use the EC2 instance metadata service (`http://169.254.169.254/latest/meta-data/`) to retrieve information about the instance, such as its instance ID, availability zone, and network interfaces. This can help identify any misconfigurations or issues with the instance itself.

**8. Contact AWS Support:**

If you're unable to resolve the issue using the above steps, consider contacting AWS Support for further assistance. They can help troubleshoot and diagnose the problem, especially if it's related to underlying infrastructure issues or platform-level problems.

**8. What is the use of count in Terraform while creating the infrastructure?**

In Terraform, the count parameter is useful for scenarios where you need to provision multiple similar resources, such as virtual machines, subnets, or security groups, but you don't want to duplicate the configuration for each resource manually. Instead, you can use count to generate multiple instances of the same resource with varying attributes or names.

# Example: Creating multiple EC2 instances using count

resource "aws\_instance" "example" {

count = 3 # Create 3 instances

ami = "ami-12345678"

instance\_type = "t2.micro"

}

**In this example**:

* The aws\_instance resource block defines a single EC2 instance.
* The count = 3 parameter specifies that 3 instances of the aws\_instance resource will be created.
* Each instance will have the same configuration defined within the resource block, including the AMI ID and instance type.
* Terraform will generate three instances with distinct names (e.g., aws\_instance.example[0], aws\_instance.example[1], aws\_instance.example[2]) and manage them as separate resources.

**9. Do you know VPC? If yes, how can you connect if there are multiple VPCs in different regions?**

Yes, I'm familiar with VPC (Virtual Private Cloud) in the context of AWS (Amazon Web Services). If you have multiple VPCs in different regions and you need to connect them, you typically use one of the following methods:

**1. VPC Peering:**

* VPC peering allows you to connect two VPCs within the same AWS region.
* It enables instances in the peered VPCs to communicate with each other using private IP addresses as if they were part of the same network.
* However, VPC peering does not support connections across different AWS regions.

**2. AWS Transit Gateway:**

* AWS Transit Gateway is a service that allows you to connect multiple VPCs, VPNs, and on-premises networks in a hub-and-spoke model.
* You can create a transit gateway in one region and attach VPCs from multiple regions to it.
* This allows for centralized management and routing of traffic between VPCs across different regions.

**3. VPN Connections:**

* If you need to establish secure connections between VPCs in different regions, you can use AWS VPN Connections.
* AWS VPN Connections allow you to create encrypted IPsec VPN tunnels between your VPCs and on-premises networks or other VPCs.
* You can configure VPN connections to connect VPCs in different regions using the AWS VPN Gateway service.

**4. Direct Connect:**

* AWS Direct Connect provides a dedicated network connection between your on-premises data center or network and one or more AWS regions.
* With Direct Connect, you can extend your private network into AWS and establish private connectivity to your VPCs in different regions.

**10. In the Kubernetes cluster, can one server have multiple IP addresses to which it is up and running with high availability?**

In a Kubernetes cluster, each node typically has one primary IP address that is used for communication within the cluster and with external services. However, it's possible for a node to have multiple IP addresses assigned to it.

Having multiple IP addresses on a node can contribute to high availability and fault tolerance by providing redundancy and load distribution. However, it's important to configure these multiple IP addresses properly to ensure that Kubernetes components and workloads can communicate effectively, and that network traffic is appropriately managed and load balanced.

Additionally, achieving high availability in a Kubernetes cluster typically involves configuring redundancy at various levels, including multiple nodes, load balancers, and replication of critical components such as etcd and control plane components. Simply having multiple IP addresses on a single node is just one aspect of achieving high availability in a Kubernetes environment.

**11. Do you know Python, what is difference between List and Tupple?**

Yes, I'm familiar with Python. In Python, lists and tuples are both sequence data types used to store collections of items. However, they have some key differences:

* Lists are mutable, which means that you can modify, add, or remove elements from them after they have been created.
* Tuples are immutable, which means that once a tuple is created, its elements cannot be changed, added, or removed.

**12. What is the state file in Terraform and where it is located?**

In Terraform, the state file (also known as the Terraform state) is a JSON-formatted file that keeps track of the state of your infrastructure managed by Terraform. It stores information about the resources that Terraform manages, their current state, metadata, dependencies, and other relevant details.

The state file is typically stored locally on the machine where Terraform commands are executed, but it can also be stored remotely for collaboration and coordination in a team environment.

**Local Filesystem**: By default, Terraform creates a local state file named terraform.tfstate in the working directory where you run Terraform commands. This file should not be version controlled or shared directly, as it may contain sensitive information.

**Remote Backend:** Terraform supports using remote backends, such as Amazon S3, Azure Blob Storage, Google Cloud Storage, or HashiCorp Consul, to store the state file remotely. Remote backends provide advantages such as improved collaboration, locking mechanisms, and versioning of state files. You can configure Terraform to use a remote backend by specifying backend configurations in your Terraform configuration files (**backend.tf**).

**13. What are the Different types of Storage services in AWS?**

In AWS, there are several types of storage services designed to cater to different use cases & requirements.

**1. Amazon S3 (Simple Storage Service):**

* Amazon S3 is an object storage service that offers scalable storage for storing and retrieving any amount of data.
* It is highly durable, reliable, and designed for 99.999999999% (11 nines) of durability.
* S3 is commonly used for data backup, archiving, data lakes, static website hosting, and serving static content for web applications.

**2. Amazon EBS (Elastic Block Store):**

* Amazon EBS provides block-level storage volumes that can be attached to EC2 instances to provide persistent storage.
* EBS volumes are highly available and reliable, offering low-latency performance.
* EBS volumes are commonly used for databases, file systems, and application storage that require block-level access.

**3. Amazon EFS (Elastic File System):**

* Amazon EFS offers scalable and fully managed file storage that can be accessed by multiple EC2 instances concurrently.
* EFS provides a POSIX-compliant file system, making it suitable for shared file storage across multiple instances.
* It is commonly used for content management, development environments, & shared file systems.

**4. Amazon FSx (File System):**

* Amazon FSx provides fully managed file storage services for Windows and Lustre file systems.
* FSx for Windows File Server offers fully managed Windows file shares accessible over the SMB.
* FSx for Lustre is a high-performance file system optimized for compute-intensive workloads such as machine learning and high-performance computing.

**5. Amazon RDS (Relational Database Service):**

* Amazon RDS is a managed database service that provides scalable, high-performance databases in the cloud.
* RDS supports various database engines, including MySQL, PostgreSQL, MariaDB, Oracle, SQL Server, and Amazon Aurora.
* It offers features such as automated backups, failover, and automated software patching.

**6. Amazon DynamoDB**:

* Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability.
* DynamoDB is designed for applications that require single-digit millisecond latency and can handle massive workloads.
* It is commonly used for web and mobile applications, gaming, IoT, and real-time analytics.

**7. Amazon S3 Glacier and Glacier Deep Archive:**

* Amazon S3 Glacier and Glacier Deep Archive are archival storage services for long-term data retention at low costs. They are designed for data archiving and backup, compliance, and data preservation. Glacier offers flexible retrieval options with varying retrieval times and costs to meet different archival needs.

**14. What are the Different types of classifications in S3 and which is high cost and which is low cost?**

Amazon S3 (Simple Storage Service) offers different storage classes, each designed to provide varying levels of durability, availability, and cost. The main storage classes in Amazon S3 are **S3 Standard**, S3 Intelligent-Tiering, S3 Standard-IA (Infrequent Access), S3 One Zone-IA, S3 Glacier, S3 Glacier Deep Archive.

S3 Standard and S3 Intelligent-Tiering offer the highest availability and performance but come at a higher cost, while S3 Glacier and S3 Glacier Deep Archive provide the lowest-cost options for archival storage with longer retrieval times.

**15. What is CI CD pipeline and how can you deploy the application using CI/CD?**

A CI/CD pipeline (Continuous Integration/Continuous Deployment) is a set of automated processes and tools used to deliver software changes quickly, reliably, and repeatedly.

**1. Continuous Integration (CI):**

* CI focuses on integrating code changes from multiple developers into a shared repository frequently, usually several times a day.
* It involves automating the process of building, testing, and validating code changes to ensure that they don't introduce errors or conflicts with existing code.
* CI systems typically use version control systems (e.g., Git) and automated build tools (e.g., Jenkins, Travis CI, CircleCI) to trigger builds and tests automatically whenever code changes are pushed to the repository.

**2. Continuous Deployment (CD):**

* CD focuses on automating the deployment of code changes to production or other environments after they have been successfully built and tested in the CI process.
* It involves automating the process of packaging, deploying, and configuring applications in various environments, such as development, staging, and production.
* CD pipelines use deployment automation tools (e.g., AWS CodeDeploy, Kubernetes, Ansible, Terraform) to deploy applications consistently and reliably across different environments.

Now, let's discuss how you can deploy an application using a CI/CD pipeline:

**1. Setup Version Control:** Start by using a version control system (e.g., Git) to manage your application's source code. Host your repository on platforms like GitHub, GitLab, or Bitbucket.

**2. Configure CI Pipeline**: Create a CI pipeline that automatically builds and tests your application whenever changes are pushed to the repository. Define stages in your CI pipeline, such as build, test, and code quality checks. Use CI tools like Jenkins, Travis CI, or CircleCI to configure your CI pipeline and trigger builds automatically.

**3. Write Tests:** Write automated tests (unit tests, integration tests, end-to-end tests) for your application to ensure its functionality and reliability. Include test execution as part of your CI pipeline to automatically verify code changes.

**4. Define Deployment Process**: Define a deployment process that describes how your application should be deployed to different environments (e.g., development, staging, production).

Specify deployment scripts, configuration files, and infrastructure definitions required for deploying the application.

**5. Configure CD Pipeline**: Create a CD pipeline that automates the deployment of your application to various environments based on predefined triggers (e.g., successful CI build).

Use deployment automation tools (e.g., AWS CodeDeploy, Kubernetes, Ansible, Terraform) to define and execute deployment workflows.

**6. Implement Blue-Green Deployments or Canary Releases (Optional):** Consider implementing advanced deployment strategies such as blue-green deployments or canary releases to minimize downtime and mitigate risks during deployments.

**7. Monitor and Track Deployments**: Implement monitoring and logging solutions to track the health and performance of your application after deployment.

Set up alerts to notify you of any issues or failures during deployment or in production.

**16. Can you write the code or create Terraform file to create VPC in AWS?**

# Declare the AWS provider

provider "aws" {

region = "us-east-1" # Specify the AWS region where you want to create the VPC

}

# Create a VPC

resource "aws\_vpc" "my\_vpc" {

cidr\_block = "10.0.0.0/16" # Specify the CIDR block for the VPC (IPv4 range)

tags = {

Name = "MyVPC" # Tag the VPC with a name for easier identification

}

}

# Create an internet gateway

resource "aws\_internet\_gateway" "my\_igw" {

vpc\_id = aws\_vpc.my\_vpc.id # Attach the internet gateway to the VPC

tags = {

Name = "MyIGW" # Tag the internet gateway with a name

}

}

# Create a route table

resource "aws\_route\_table" "my\_route\_table" {

vpc\_id = aws\_vpc.my\_vpc.id # Associate the route table with the VPC

route {

cidr\_block = "0.0.0.0/0" # Route all traffic to the internet gateway

gateway\_id = aws\_internet\_gateway.my\_igw.id

}

tags = {

Name = "MyRouteTable" # Tag the route table with a name

}

}

# Associate the route table with the VPC's main route table association

resource "aws\_vpc\_association\_route\_table" "main" {

vpc\_id = aws\_vpc.my\_vpc.id

route\_table\_id = aws\_route\_table.my\_route\_table.id

}

**17. What is the command to know only running containers?**

To list only the running containers on your system, you can use the docker ps command with the --filter option to filter containers by their status. **Example**: **docker ps --filter "status=running"**

* To list only the running containers on your system: **docker ps --filter "status=running"**
* To list only stopped containers: docker ps --filter "status=exited"
* To list only paused containers: docker ps --filter "status=paused"
* To list only dead containers: docker ps --filter "status=dead"

**18. What is command to read the file in Linux?**

viewing the entire contents of the file (cat, less, more) or just a portion of it (head, tail).

**19. What is the awk command and where it is used? and What is sed?**

**awk** is a versatile text-processing tool used for pattern scanning and processing. **awk '{print $1}' data.csv.**

**sed** (stream editor) is a powerful text-processing utility used for performing text transformations on input streams. sed is often used for tasks such as search and replace, text substitution, deletion, insertion, and other text manipulations. Ex: **sed 's/Hello/Hi/' example.txt.**

**20. What is the command to know the CPU and memory information of the Linux machine?**

To retrieve CPU and memory information on a Linux machine, you can use various commands. Here are some commonly used commands:

1. `top`: The `top` command displays real-time information about system processes, CPU usage, memory usage, and more. To launch `top`, simply type `top` in the terminal and press Enter.

2. `htop`: `htop` is an interactive process viewer similar to `top` but with a more user-friendly interface.

**3. `free**`: The `free` command displays information about system memory usage, including total, used, and free memory. To use `free`, simply type `free` in the terminal and press Enter.

**4. `cat /proc/cpuinfo`:** The `/proc/cpuinfo` file contains information about the CPU installed on the system.

5. `lscpu`: The `lscpu` command provides detailed information about CPU architecture & configuration.

**6. `dmidecode`:** The `dmidecode` command provides information about hardware components, including memory modules. To use `dmidecode`, you may need to install it first (`sudo apt install dmidecode` on Ubuntu/Debian systems) and then type `sudo dmidecode` in the terminal.

**21. What is Blue Green Deployment?**

Blue-green deployment is a deployment strategy used in software development and release management. It is designed to minimize downtime and risk during the deployment process by ensuring that a new version of an application can be deployed without impacting users.

In a blue-green deployment, two identical production environments, typically referred to as "blue" and "green," are maintained in parallel. At any given time, only one environment (either blue or green) serves live traffic and handles user requests, while the other environment remains inactive.

**22. What is the Difference between Imperative & Declarative Methods in Kubernetes?**

In Kubernetes, imperative and declarative methods represent two different approaches to managing resources within the cluster.

**1. Imperative Method:**

* With the imperative method, you directly instruct Kubernetes on what actions to take. This typically involves using command-line tools like `kubectl` to issue commands such as `kubectl create`, `kubectl delete`, `kubectl scale`, etc.
* Imperative commands are action-oriented and focus on the specific steps needed to achieve a desired state.
* Imperative commands can be useful for quick, one-off tasks or for operations where the desired state is not known in advance.

**2. Declarative Method:**

* In contrast, the declarative method involves describing the desired state of the resources in configuration files, typically written in YAML or JSON format.
* Instead of issuing commands to Kubernetes, you provide configuration files that define the desired state of the resources, including their properties and desired behaviors.
* Kubernetes then uses these configuration files to reconcile the current state of the cluster with the desired state specified in the files.
* Common tools for managing resources declaratively include `kubectl apply` which applies the configuration defined in a YAML file to the cluster.

**Key Differences:**

* Imperative methods focus on direct commands to perform actions, while declarative methods focus on defining desired states in configuration files.
* Imperative methods may be more intuitive for ad-hoc or interactive operations, while declarative methods are generally preferred for managing resources consistently and reproducibly over time.
* Declarative methods lend themselves well to automation and version control, as configuration files can be stored and managed alongside application code.

**23. How Can you deploy or create serverless instances?**

In AWS, you can deploy serverless applications using services like AWS Lambda, AWS API Gateway, AWS DynamoDB, AWS S3, and others. Here's a basic overview of how you can create serverless instances using AWS Lambda:

**1. Create a Lambda Function:**

* Log in to the AWS Management Console and navigate to the Lambda service.
* Click on "Create function" and choose an authoring method. You can either author your function from scratch, or you can use blueprints, which are preconfigured templates.
* Configure the function settings, such as the function name, runtime (e.g., Node.js, Python, Java), and permissions.
* Write your function code or paste it if you're using a blueprint.
* Define the handler function, which is the entry point to your Lambda function.

**2. Configure Triggers:**

* Lambda functions can be triggered by various AWS services such as API Gateway, S3, DynamoDB, SNS, CloudWatch Events, etc.
* Choose the appropriate trigger for your Lambda function. For example, if you want your function to be triggered by an HTTP request, you can configure API Gateway as a trigger.

**3. Set Up IAM Roles**: Ensure that your Lambda function has the necessary permissions to interact with other AWS services. You can create an IAM role and attach policies granting specific permissions to your Lambda function.

**4. Test Your Lambda Function:** Once your Lambda function is created, you can test it using the AWS Management Console, command-line interface (CLI), or SDKs. AWS provides a testing console where you can input test events and observe the output of your function.

**5. Deploy and Monitor:** After testing, deploy your Lambda function to make it available for production use. Monitor your Lambda function's performance and usage using CloudWatch metrics and logs. You can set up alarms and notifications to respond to any issues.

**6. Scaling and Optimization**: AWS Lambda automatically scales your function in response to incoming requests. You don't need to provision or manage servers. Optimize your Lambda functions for performance and cost by adjusting memory allocation, optimizing code, and configuring concurrency settings.

By following these steps, you can create and deploy serverless instances using AWS Lambda and other related services in the AWS ecosystem. Additionally, you can leverage additional AWS services like API Gateway, DynamoDB, S3, etc., to build fully serverless applications.

**24. How Can you expose the 10 Kubernetes services over internet?**

To expose Kubernetes services over the internet, you typically use Kubernetes Service resources combined with an Ingress controller or a Load Balancer.

**1. Create Kubernetes Services:**

* Define Kubernetes Service resources for each of the 10 services you want to expose.
* Kubernetes Services are used to expose applications running on the cluster, either internally or externally.

**2. Choose Expose Method:**

* Determine whether you want to expose your services using an Ingress resource or a Load Balancer.
* Ingress: If you have multiple services to expose and want to manage routing and SSL termination at the HTTP level.
* Load Balancer: If you want each service to have its own public IP address and manage traffic at the TCP/UDP level.

**3. Deploy Ingress Controller or Load Balancer**:

* If you choose Ingress, deploy an Ingress controller such as Nginx Ingress Controller, Traefik, or HAProxy Ingress.
* If you choose Load Balancer, Kubernetes will provision a cloud provider Load Balancer for each Service with type `Load Balancer`.

**4. Configure Ingress or Load Balancer Rules:**

* For Ingress, define rules in the Ingress resource to route incoming HTTP(S) traffic to the appropriate Services based on the hostname and path.
* For Load Balancer, Kubernetes automatically provisions a public IP address or DNS name that forwards traffic to the selected Service.

**5. Set Up DNS:**

* If you're using Ingress, set up DNS records (A or CNAME) to point to the public IP address of your Ingress controller.
* If you're using Load Balancer, DNS setup may vary depending on your cloud provider. Some providers automatically configure DNS for Load Balancers.

**6. Secure Communication (Optional):**

* If your services require encryption, configure SSL termination either at the Ingress controller or the Load Balancer.
* You can use TLS certificates from a Certificate Authority like Let's Encrypt.

**7. Test and Monitor:**

* Test the connectivity to your services from external networks.
* Monitor the performance and health of your services and the Ingress controller or Load Balancer.

By following these steps, you can expose your Kubernetes services over the internet using either an Ingress controller or a Load Balancer, depending on your requirements and preferences. Make sure to follow best practices for security and scalability when exposing services to the internet.

**25. Can you please tell me few commands you use daily in Kubernetes?**

Certainly! Here are a few common Kubernetes commands that you might use daily:

**1. kubectl get:** **kubectl get pods** -- Lists all pods in the current namespace.

**kubectl get deployments** -- Lists all deployments in the current namespace.

**kubectl get services** -- Lists all services in the current namespace.

**2. kubectl describe: kubectl describe pod <pod-name>** Provides detailed information about a specific pod. **kubectl describe deployment <deployment-name>** Provides detailed information about a specific deployment. **kubectl describe service <service-name>** Provides information about a specific service.

**3. kubectl logs: kubectl logs <pod-name>** Displays logs from a specific pod.

kubectl logs -f <pod-name> Streams logs from a specific pod.

**4. kubectl exec: kubectl exec -it <pod-name> -/bin/bash** Executes a shell session inside a specific pod.

**kubectl exec -it <pod-name> -<command>** Executes a specific command inside a pod.

**5. kubectl create/apply: kubectl create -f <yaml-file>** Creates resources defined in a YAML file.

**kubectl apply -f <yaml-file>** Applies changes to resources defined in a YAML file.

**6. kubectl delete**: **kubectl delete pod <pod-name>** Deletes a specific pod.

**kubectl delete deployment <deployment-name>** Deletes a specific deployment.

**kubectl delete service <service-name>** Deletes a specific service.

**7. kubectl scale:** kubectl scale deployment <deployment-name> --replicas=<number> Scales the number of replicas in a deployment.

**8. kubectl rollout:**

**kubectl rollout status deployment/<deployment-name>** Checks the status of a deployment rollout.

**kubectl rollout history deployment/<deployment-name>** Lists revision history for a deployment.

**kubectl rollout undo deployment/<deployment-name>** Rolls back a deployment to a previous revision.

**26. How Can you manage your Kubernetes cluster?**

Managing a Kubernetes cluster involves various tasks to ensure that the cluster is running smoothly, efficiently, and securely. Here are some key aspects of managing a Kubernetes cluster:

**1. Cluster Provisioning and Setup:**

* Provisioning the underlying infrastructure where Kubernetes will run, whether it's on-premises, in the cloud, or using a managed Kubernetes service like Amazon EKS, Google Kubernetes Engine (GKE), or Azure Kubernetes Service (AKS).
* Installing Kubernetes using tools like kubeadm, kops, or managed Kubernetes services.
* Configuring network plugins (CNI), storage solutions, and other add-ons.

**2. Cluster Configuration:**

* Managing cluster-level configuration settings such as authentication, authorization, networking, and storage.
* Configuring security policies, resource quotas, and admission controllers.
* Setting up monitoring and logging solutions to track cluster performance and troubleshoot issues.

**3. Resource Management:**

* Deploying and managing applications and services using Kubernetes resources like Pods, Deployments, Services, Config Maps, and Secrets.
* Scaling applications horizontally or vertically based on resource usage and demand.
* Monitoring resource utilization and optimizing resource allocation to improve efficiency.

**4. Cluster Security:**

* Implementing security best practices such as RBAC (Role-Based Access Control), network policies, and pod security policies.
* Regularly applying security patches and updates to Kubernetes components and underlying infrastructure.
* Securing sensitive information using secrets management and encryption.

**5. Monitoring and Logging:**

* Setting up monitoring tools like Prometheus, Grafana, or Kubernetes-native solutions like Kubernetes Metrics Server.
* Collecting and analyzing logs from Kubernetes components and application pods using tools like Fluentd, Elasticsearch, and Kibana (EFK) stack or Loki and Grafana (Grafana Labs).

**6. Backup and Disaster Recovery:**

* Implementing backup and restore procedures for critical data and configuration settings.
* Creating disaster recovery plans to minimize downtime in case of cluster failures or data loss events.

**7. Upgrades and Maintenance:**

* Planning and executing Kubernetes version upgrades for both control plane components and worker nodes.
* Performing routine maintenance tasks such as node repairs, pod evictions, and garbage collection.

1. **Troubleshooting and Debugging:** Diagnosing and resolving issues related to networking, storage, application deployment, and performance. Using Kubernetes-native debugging tools like kubectl exec, kubectl logs, and kubectl describe to troubleshoot pods and deployments.

**27. How many clusters we can be deployed and managed in an organization?**

The number of Kubernetes clusters that can be deployed and managed in an organization depends on various factors including the organization's size, infrastructure complexity, resource availability, management overhead, and specific use cases. There is no fixed limit to the number of clusters an organization can manage, but here are some considerations:

**1. Organizational Size and Structure:**

1. Large organizations with multiple teams or departments may deploy separate clusters to isolate workloads, manage access control, and meet compliance requirements.
2. Smaller organizations or startups might manage a single cluster to simplify operations and reduce overhead.

**2. Isolation and Security:**

* Organizations may deploy separate clusters for different environments such as development, staging, and production to isolate workloads and enforce security boundaries.
* Clusters may also be deployed for specific projects, teams, or customers to ensure data and resource isolation.

**3. Geographic Distribution:**

* Organizations with global presence may deploy clusters in multiple regions or data centers to reduce latency and ensure high availability.
* Multi-cluster deployments can also improve compliance with data residency regulations by keeping data within specific geographic boundaries.

**4. Specialized Use Cases:**

* Certain use cases may require dedicated clusters optimized for specific workloads such as machine learning, big data analytics, or high-performance computing.
* Separate clusters may be deployed for stateful applications requiring specialized storage solutions or hardware accelerators.

**5. Resource Constraints:**

* Resource availability including compute, storage, and networking infrastructure can limit the number of clusters an organization can deploy and manage effectively.
* Managing multiple clusters increases operational overhead and resource consumption for tasks such as monitoring, logging, and upgrades.

**6. Management Complexity:**

* Each cluster adds complexity to management tasks such as configuration, monitoring, and troubleshooting.
* Organizations need to consider whether they have the necessary expertise, tooling, and processes to manage multiple clusters efficiently.

In summary, while there is no hard limit to the number of clusters an organization can deploy and manage, the decision to deploy multiple clusters should be based on factors such as organizational structure, security requirements, geographic distribution, specialized use cases, resource constraints, and management complexity. It's important for organizations to evaluate their specific requirements and priorities to determine the optimal number of Kubernetes clusters for their environment.

**28. What are the various storage types in Kubernetes?**

In Kubernetes, there are several types of storage options available to manage persistent data for applications running in pods. These storage options cater to different use cases, performance requirements, and storage backends. Here are some of the various storage types in Kubernetes:

**1. Persistent Volume (PV):** Persistent Volume (PV) is a storage abstraction in Kubernetes that represents a piece of storage in the cluster. PVs are provisioned by an administrator and can be statically provisioned or dynamically provisioned using storage classes. PVs have a lifecycle independent of any pod and can be dynamically or statically bound to Persistent Volume Claims (PVCs) from pods.

**2. PersistentVolumeClaim (PVC):** PersistentVolumeClaim (PVC) is a request for storage by a pod. It allows pods to request specific storage resources without needing to know the details of the underlying storage.

Pods use PVCs to claim a PV and use the requested storage within the pod.

**3. StorageClass**: StorageClass is used to define different classes of storage in Kubernetes. It provides a way to dynamically provision storage based on the storage requirements specified in PVs or PVCs. StorageClass allows administrators to define storage configurations such as volume type, access mode, and provisioner.

**4. HostPath**: HostPath volumes are used to mount files or directories from the host node's filesystem into pods. They are primarily used for development and testing purposes and should be avoided in production environments due to lack of data persistence and potential security risks.

**5. EmptyDir:** EmptyDir volumes are ephemeral volumes that are created when a pod is assigned to a node and deleted when the pod is terminated. They are useful for sharing data between containers within the same pod or for temporary storage.

**6. NFS (Network File System):** NFS volumes allow pods to mount file systems from remote NFS servers.

They provide a shared storage solution for distributed applications running in Kubernetes clusters.

**7. CSI (Container Storage Interface):** CSI is a standard interface for exposing arbitrary block and file storage systems to containerized workloads in Kubernetes. CSI drivers enable integration with a wide range of external storage platforms and provide advanced features such as snapshots, cloning, and encryption.

**8. Local Persistent Volumes:** Local Persistent Volumes allow pods to use local storage devices directly attached to the node where the pod is scheduled. They are suitable for applications that require high-performance storage or have specific requirements for local disk access.

**29. How you create a cluster? is it manually or using AWS Cloud provider?**

Creating a Kubernetes cluster can be done manually or using a cloud provider like AWS.

**1. Manual Cluster Creation:**

* Manually creating a Kubernetes cluster involves setting up the required infrastructure, installing Kubernetes components, and configuring the cluster.
* Steps for manual cluster creation typically include:
* Provisioning virtual or physical machines to act as Kubernetes nodes (master & worker nodes).
* Installing container runtime like Docker or contained on each node.
* Installing Kubernetes control plane components (API server, Controller Manager, Scheduler, etc.) on master node(s).
* Joining worker nodes to the cluster by configuring kubelet and kube-proxy.
* Configuring networking (e.g., using tools like Flannel, Calico, or Weave).
* Optionally, configuring additional features such as RBAC, ingress controllers, monitoring tools.
* Manual cluster creation provides flexibility and control over cluster configuration but can be time-consuming and requires expertise in Kubernetes administration.

**2. Cloud Provider Cluster Creation:**

* Cloud providers like AWS offer managed Kubernetes services that simplify the process of creating and managing Kubernetes clusters.
* With AWS, you can use services like Amazon EKS (Elastic Kubernetes Service) to create Kubernetes clusters.
* To create a cluster using AWS EKS, you typically follow these steps:
* Create an EKS cluster using the AWS Management Console, CLI, or SDK.
* Define cluster settings such as region, node instance types, networking, & Kubernetes version.
* AWS EKS automatically provisions the required infrastructure and manages Kubernetes control plane components.
* You can then connect to the EKS cluster using kubectl and deploy applications and workloads.
* Cloud provider managed Kubernetes services abstract away the complexities of cluster setup and management, allowing you to focus on deploying and managing applications.

**30. How Can you access the instance or server over internet but can’t be edited in AWS?**

In AWS, you can restrict access to an EC2 instance or server over the internet while ensuring that it cannot be edited or modified by unauthorized users. One common approach to achieve this is by utilizing AWS Identity and Access Management (IAM) along with security groups and network ACLs.

1**. IAM Policies:**

* Create an IAM policy that grants read-only access to the EC2 instance or server. This policy should allow users to view the instance details but restrict any modifications.
* Specify permissions for actions like `ec2:DescribeInstances`, `ec2:DescribeInstanceAttribute`, and other read-only actions relevant to your use case.
* Attach this policy to the IAM user, group, or role that needs access to the instance.

**2. Security Groups:**

* Configure the security group associated with the EC2 instance to allow inbound traffic only from authorized IP addresses or CIDR blocks.
* Limit access to specific ports required for viewing the instance (e.g., SSH for Linux instances or RDP for Windows instances).
* Ensure that outbound traffic is allowed as necessary for the instance to function properly.

**3. Network ACLs:**

* Network ACLs provide an additional layer of security at the subnet level. You can use them to control inbound and outbound traffic to and from the EC2 instance.
* Configure network ACL rules to allow inbound traffic only from authorized sources and restrict outbound traffic as necessary.
* Network ACLs are stateless, so you need to configure both inbound and outbound rules separately.

**4. Instance Hardening**: Implement additional security measures such as disabling unnecessary services, regularly applying security patches, and configuring appropriate logging and monitoring. For Linux instances, consider using tools like AWS Systems Manager (SSM) for session logging and auditing.

**31. Explain the difference between a Deployment and statefulset in kubernetes?**

Deployments are used to manage stateless applications and provide features like rolling updates & scaling.

StatefulSets are used to manage stateful applications that require stable, unique network identifiers and persistent storage. StatefulSets provide ordering and quiqueness guarantees for pod management.

**32. What are the security policies do you have in AWS?**

In AWS, there are various security features and policies that you can implement to secure your resources and infrastructure. Some of the key security policies and best practices in AWS include:

**1. Identity and Access Management (IAM):** IAM enables you to manage users, groups, and roles to control access to AWS services and resources. Implement least privilege principles by granting only the permissions necessary for users or roles to perform their tasks. Use IAM roles for EC2 instances, Lambda functions, and other services to securely delegate permissions without using long-term credentials.

**2. Network Security:** Use Virtual Private Cloud (VPC) to create isolated network environments and define network access control using security groups and network ACLs. Implement least privilege by restricting inbound and outbound traffic based on the principle of least privilege. Utilize AWS PrivateLink for private connectivity between VPCs and AWS services, avoiding exposure to the public internet.

**3. Data Encryption:** Enable encryption at rest for data stored in AWS services such as Amazon S3, Amazon RDS, and Amazon EBS volumes using AWS Key Management Service (KMS) encryption. Implement encryption in transit by using SSL/TLS for communication between clients and AWS services or between AWS services.

**4. Monitoring and Logging:** Use AWS CloudTrail to log API activity and monitor changes to resources, providing visibility into user activity and potential security threats. Enable AWS Config to assess, audit, and evaluate the configurations of AWS resources for compliance with security policies and best practices. Utilize AWS CloudWatch for monitoring and alerting on resource utilization, performance, security events.

**5. Security Compliance**: Implement security compliance standards such as PCI DSS, HIPAA, GDPR, and others by configuring AWS services and resources according to the specific requirements of each standard. Use AWS Artifact to access compliance reports and documentation to support audit and compliance requirements.

**6. Distributed Denial of Service (DDoS) Protection**: Use AWS Shield Standard or AWS Shield Advanced to protect against DDoS attacks by leveraging AWS's global network infrastructure and automated mitigation capabilities.

**7. Incident Response and Disaster Recovery:** Implement incident response procedures and disaster recovery plans to respond to security incidents and ensure business continuity. Utilize AWS services such as AWS Backup, Amazon S3 versioning, and cross-region replication for data backup and disaster recovery.

**8. Security Automation and Orchestration:** Use AWS Security Hub to centrally manage and automate security findings from multiple AWS services and third-party tools. Leverage AWS Config Rules, AWS Lambda, and AWS Systems Manager Automation for automated security remediation and enforcement of security policies.

**33. How can you create an organization using IAM in AWS?**

In AWS, an organization is typically created and managed using AWS Organizations, a service that enables you to centrally manage and govern multiple AWS accounts.

**1. Sign in to the AWS Management Console:**

* Log in to the AWS Management Console using an AWS account with administrative privileges.

**2. Navigate to AWS Organizations:**

* In the AWS Management Console, navigate to the AWS Organizations service by either searching for "Organizations" in the services search bar or locating it under the "Management & Governance" section.

**3. Create a New Organization:**

* If you don't have an organization already, you'll be prompted to create one. Click on the "Create organization" button to get started.

**4. Choose Organization Type:**

* Select the type of organization you want to create: either Consolidated Billing or All Features.
* Consolidated Billing allows you to consolidate billing and payment for multiple AWS accounts linked to the organization, while All Features provides additional capabilities for managing accounts, policies, and organizational units (OUs).

**5. Enter Organization Details:**

* Enter a name for your organization and specify the email address for the AWS account that will act as the management account.
* The management account will have full control over the organization and its member accounts.

**6. Review and Confirm:**

* Review the organization details and configuration settings to ensure they are correct.
* Click on the "Create organization" button to create the organization.

**7. Invite Member Accounts (Optional):**

* If you selected the All Features option, you can invite existing AWS accounts to join the organization as member accounts.
* Send invitations to the email addresses associated with the member accounts, and the account owners can accept the invitations to join the organization.

**8. Set up Organizational Units (OUs) (Optional):**

* Create OUs to organize and group member accounts based on your organizational structure or business units.
* Define policies at the OU level to manage permissions, service control policies (SCPs), and other governance controls.

**9. Manage Accounts and Policies:**

* Once the organization is created, you can manage member accounts, policies, and organizational units from the AWS Organizations console.
* Assign service control policies (SCPs) to control access to AWS services and resources across member accounts.
* Enable features like AWS Single Sign-On (SSO) for centralized authentication and access management.

**34. How Can you transform the data in AWS and is there any service for data transform in AWS?**

Yes, we can transform data using various services depending on your specific requirements & use cases.

**1. AWS Glue:**

* AWS Glue is a fully managed extract, transform, and load (ETL) service that makes it easy to prepare and load data for analytics.
* Glue provides a serverless environment for running ETL jobs, allowing you to transform data at scale without managing infrastructure.
* You can define data transformation workflows using Glue's visual interface or write custom scripts using Apache Spark APIs.

**2. Amazon EMR (Elastic MapReduce):**

* Amazon EMR is a fully managed big data platform that allows you to process and analyze large amounts of data using open-source frameworks like Apache Hadoop, Spark, and Presto.
* EMR clusters can be customized to perform various data transformation tasks, including ETL, data cleaning, and data preparation.
* You can use EMR's built-in applications and frameworks or install custom software packages to perform specific data transformations.

**3. Amazon Athena:**

* Amazon Athena is an interactive query service that allows you to analyze data stored in Amazon S3 using standard SQL queries.
* While Athena does not directly perform data transformation, you can use SQL queries to transform and filter data as part of your analysis process.

**4. AWS Data Pipeline:**

* AWS Data Pipeline is a web service for orchestrating and automating data-driven workflows.
* Data Pipeline allows you to define data processing tasks, dependencies, and schedules using a visual interface or JSON-based definitions.
* You can use Data Pipeline to transform data between different AWS services, such as moving data from S3 to Redshift, running EMR jobs, or executing SQL queries.

**5. Amazon Kinesis Data Analytics:**

* Amazon Kinesis Data Analytics is a service for processing and analyzing streaming data in real-time.
* Kinesis Data Analytics allows you to run SQL queries on streaming data streams to perform data transformations, aggregations, and analytics.

**6. AWS Lambda:**

* AWS Lambda is a serverless compute service that allows you to run code in response to events without provisioning or managing servers.
* You can use Lambda functions to perform lightweight data transformations, data enrichment, or data validation as part of your data processing pipelines.

**35. How can you connect Jenkins with AWS EKS?**

To connect Jenkins with AWS EKS (Elastic Kubernetes Service), you can use Jenkins plugins and Kubernetes configuration to enable Jenkins to deploy and manage applications on your EKS cluster.

**1. Install Jenkins Kubernetes Plugin:**

* In your Jenkins instance, navigate to "Manage Jenkins" -> "Manage Plugins" -> "Available" tab.
* Search for "Kubernetes" plugin and install it.
* This plugin enables Jenkins to dynamically provision Kubernetes pods to run build agents and execute jobs.

**2. Configure Kubernetes Cloud:**

* Go to "Manage Jenkins" -> "Configure System".
* Scroll down to the "Cloud" section and click on "Add a new cloud" -> "Kubernetes".
* Configure the Kubernetes Cloud details:
* Kubernetes URL: Enter the URL of your EKS cluster's API server.
* Kubernetes server certificate key: (Optional) Enter the CA certificate key for your EKS cluster.
* Kubernetes Namespace: (Optional) Specify the namespace where Jenkins agents will be provisioned.
* Credentials: Add Kubernetes Service Account credentials or kubeconfig file with appropriate permissions to access the EKS cluster.
* Click on "Test Connection" to verify the connection to the Kubernetes cluster.

**3. Configure Jenkins Agents:**

* In the Kubernetes Cloud configuration, specify the Docker image to use for Jenkins agents. You can use pre-built Jenkins agent images from Docker Hub or your custom Docker images.
* Define the resource requests and limits for Jenkins agents, including CPU and memory requirements.
* Optionally, configure pod retention policies, labels, and other advanced settings as needed.

**4. Create a Jenkins Pipeline:**

* Create a Jenkins pipeline or job that defines the steps for building, testing, and deploying your application to the EKS cluster.
* Use Kubernetes-specific steps in your pipeline to interact with the EKS cluster, such as `kubectl` commands or Kubernetes DSL.

**5. Deploy to EKS:**

* Use Kubernetes deployment manifests or Helm charts to define your application deployment configuration.
* Configure your Jenkins pipeline to deploy the application to the EKS cluster using `kubectl apply` or Helm commands.

**6. Test the Pipeline:**

* Run the Jenkins pipeline to test the integration with the EKS cluster.
* Verify that Jenkins agents are dynamically provisioned in the EKS cluster to execute the pipeline stages.
* Ensure that the application is deployed successfully to the EKS cluster.

**36. How can you connect Jenkins with Docker and deploy the application?**

To connect Jenkins with Docker and deploy applications using Docker, you'll need to set up Jenkins with the Docker plugin and configure your Jenkins jobs or pipelines to build Docker images and deploy them to your desired environment.

**1. Install Docker Plugin for Jenkins:**

* In your Jenkins instance, navigate to "Manage Jenkins" -> "Manage Plugins" -> "Available" tab.
* Search for "Docker" plugin and install it.
* This plugin enables Jenkins to interact with Docker, allowing you to build Docker images and run Docker containers as part of your CI/CD pipelines.

**2. Configure Docker in Jenkins:**

* Go to "Manage Jenkins" -> "Configure System".
* Scroll down to the "Cloud" section and click on "Add a new cloud" -> "Docker".
* Configure the Docker Cloud details:
* Name: Enter a name for the Docker Cloud.
* Docker Host URI: Specify the URI of your Docker daemon (e.g., `tcp://localhost:2375` for a local Docker daemon).
* Click on "Test Connection" to verify the connection to the Docker daemon.
* You can also configure other options such as Docker agent templates, labels, and advanced settings based on your requirements.

**3. Create Docker Build Jobs or Pipelines:**

* Create a new Jenkins job or pipeline to build Docker images.
* Configure the job or pipeline to clone your application source code repository from a version control system like Git.
* Use Docker build commands in your Jenkinsfile or build steps to build Docker images for your application.
* Optionally, you can tag the Docker images with version numbers or other labels.

**4. Push Docker Images to Registry:**

* After building Docker images, push them to a Docker registry like Docker Hub, Amazon ECR, or a private registry.
* Use Docker push commands in your Jenkinsfile or build steps to push the images to the registry.
* Ensure that you have appropriate credentials configured in Jenkins to authenticate with the Docker registry.

**5. Deploy Docker Containers:**

* Create another Jenkins job or pipeline to deploy Docker containers to your desired environment (e.g., development, staging, production).
* Use Docker run commands or Docker-compose files to deploy containers based on the Docker images you built and pushed earlier.
* You can also use Docker swarm or Kubernetes for container orchestration and deploy containers to a cluster.

**6. Test the Pipeline:** Run the Jenkins job or pipeline to build Docker images, push them to the registry, and deploy containers to your environment. Monitor the build and deployment process in the Jenkins console output and verify that the application is deployed successfully.

**37. How to connect and configure the Docker with AWS EKS?**

To connect and configure Docker with AWS EKS (Elastic Kubernetes Service), you don't directly interact with Docker on EKS worker nodes as you would with a standalone Docker environment. Instead, you deploy containerized applications to EKS using Kubernetes manifests or Helm charts.

**1. Build Docker Images Locally:**

* Develop and build your Docker images locally on your development machine or a build server using Docker CLI or Dockerfile.

**2. Push Docker Images to a Registry:**

* Push your Docker images to a container registry such as Amazon ECR (Elastic Container Registry), Docker Hub, or another private registry.
* Amazon ECR is a fully managed Docker container registry that integrates seamlessly with AWS services like EKS.

**3. Define Kubernetes Deployment Manifests or Helm Charts:**

* Create Kubernetes Deployment manifests or Helm charts to describe your application's containerized components, including Docker image references, resource requirements, and configuration settings.
* Specify the Docker image URI from the container registry in your Kubernetes manifests or Helm charts.

**4. Deploy Applications to EKS:**

* Use kubectl or a Kubernetes management tool to apply your Deployment manifests or Helm charts to your EKS cluster.
* Kubernetes will schedule and deploy the specified Docker containers onto the worker nodes in your EKS cluster.

**5. Monitor and Manage Deployments:**

* Monitor the status and health of your application deployments using Kubernetes tools or AWS services like Amazon CloudWatch.
* Scale your deployments up or down, perform rolling updates, and manage application lifecycle as needed using Kubernetes commands or automation scripts.

**6. Optional: Configure Docker Build in CI/CD Pipeline:**

* If you're using Jenkins or another CI/CD tool, you can configure your pipeline to build Docker images, push them to a container registry, and trigger Kubernetes deployments to EKS.
* Use Docker CLI or Docker SDK in your pipeline to build Docker images and Docker registry APIs to push images to the registry.

**38. How to connect and configure the Kubernetes with AWS EKS?**

To connect and configure Kubernetes with AWS EKS (Elastic Kubernetes Service), you'll need to set up your Kubernetes environment, configure access credentials, and establish communication with your EKS cluster.

1. Install `kubectl`:

* Install the `kubectl` command-line tool, which is used to interact with Kubernetes clusters.
* You can download `kubectl` from the Kubernetes release page: https://kubernetes.io/docs/tasks/tools/install-kubectl/

2. Install `aws-iam-authenticator`:

* AWS EKS requires the use of AWS IAM (Identity and Access Management) for authentication. Install the `aws-iam-authenticator` CLI tool, which is used to authenticate `kubectl` commands with AWS IAM.
* You can download `aws-iam-authenticator` from the official GitHub repository: https://github.com/kubernetes-sigs/aws-iam-authenticator

3. Configure AWS CLI:

* Install and configure the AWS CLI (Command Line Interface) if you haven't already done so.
* Run `aws configure` and provide your AWS access key, secret key, default region, and output format.

4. Update `kubectl` Configuration:

* Update your `kubectl` configuration to authenticate with the AWS EKS cluster using `aws-iam-authenticator`.
* Run the command to update the `~/.kube/config` file: aws eks --region <region> update-kubeconfig --name <cluster-name>
* Replace `<region>` with your AWS region (e.g., `us-west-2`) and `<cluster-name>` with the name of your EKS cluster.

5. Verify Connection:

* Run `kubectl get nodes` to verify that you can connect to the EKS cluster and retrieve information about the worker nodes.
* If successful, you should see a list of worker nodes in your EKS cluster.

6. (Optional) Install Additional Tools:

* Install additional tools like `helm`, `kubectl-aliases`, or Kubernetes dashboard as per your requirements.
* Helm is a package manager for Kubernetes that simplifies the deployment and management of applications.
* `kubectl-aliases` provides shortcuts and aliases for common `kubectl` commands to improve productivity.
* Kubernetes dashboard is a web-based user interface for Kubernetes clusters, providing insights and management capabilities.

7. Manage Kubernetes Resources:

* Once connected, you can use `kubectl` commands to manage Kubernetes resources such as pods, deployments, services, and namespaces.
* Deploy applications to your EKS cluster using Kubernetes manifests (YAML files) or Helm charts.

**39. How can you connect AWS instance from Linux machine?**

To connect to an AWS instance (EC2 instance) from a Linux machine, you can use SSH (Secure Shell) which is a widely used protocol for securely accessing remote servers.

**1. Obtain the Public IP or DNS Name of the AWS Instance:**

Log in to the AWS Management Console.Navigate to the EC2 dashboard.

Find the EC2 instance you want to connect to and note down its Public IP address or Public DNS name.

**2. Open a Terminal on your Linux Machine:** Open a terminal window on your local Linux machine. You can use Terminal, PuTTY, or any other SSH client.

**3. Use SSH to Connect to the AWS Instance:**

* In the terminal, use the `ssh` command to connect to the AWS instance. The syntax is: ssh -i <path\_to\_private\_key> <username>@<public\_ip\_or\_dns>
* `<path\_to\_private\_key>`: Path to the private key file (.pem) associated with the key pair used to launch the EC2 instance.
* `<username>`: The username for accessing the instance. For Amazon Linux, the default username is usually `ec2-user`. For Ubuntu instances, it's `ubuntu`, and for CentOS instances, it's `centos`.
* `<public\_ip\_or\_dns>`: The Public IP address or DNS name of the EC2 instance.
* For example: ssh -i ~/.ssh/my-key.pem ec2-user@12.34.56.78

**4. Authenticate the SSH Connection:**

When connecting for the first time, SSH will prompt you to authenticate the EC2 instance's host key.

Type `yes` to confirm and proceed with the connection.

**5. Access the EC2 Instance:**

Once authenticated, you'll be logged into the EC2 instance via SSH.

You can now execute commands on the EC2 instance directly from your Linux machine's terminal.

**6. Disconnect from the Instance**: To disconnect from the instance, simply type `exit` and press Enter in the terminal.

**40. What is the port number of SSH connection?**

The default port number for SSH (Secure Shell) connections is 22 to a remote server.

**41. What are the Monitoring Mechanism Services are there in AWS?**

In AWS, there are several monitoring mechanisms and services available to monitor the health, performance, and availability of your resources and applications.

**1. Amazon CloudWatch:**

* Amazon CloudWatch is a centralized monitoring and observability service that provides monitoring for AWS resources and applications.
* CloudWatch collects and stores metrics, logs, and events from various AWS services, including EC2, RDS, Lambda, and more.
* You can use CloudWatch to set alarms, create dashboards, and perform real-time analysis of metrics to monitor the health and performance of your AWS resources.
* CloudWatch Logs allows you to monitor and troubleshoot logs from EC2 instances, Lambda functions, and other sources.

**2. AWS CloudTrail:**

* AWS CloudTrail is a service that provides logging and auditing of API activity across your AWS infrastructure.
* CloudTrail records API calls made by users, services, and AWS CLI commands, providing visibility into who did what, when, and from where in your AWS account.
* You can use CloudTrail to track changes to resources, investigate security incidents, and ensure compliance with regulatory requirements.

**3. Amazon Inspector:**

* Amazon Inspector is an automated security assessment service that helps you identify security vulnerabilities and compliance issues in your EC2 instances and applications.
* Inspector analyzes the network, file system, and process activity of your instances to identify security vulnerabilities, deviations from best practices, and potential security misconfigurations.

**4. AWS X-Ray:**

* AWS X-Ray is a distributed tracing service that helps you understand and debug the performance of your applications.
* X-Ray provides end-to-end visibility into requests as they travel through your application, helping you identify bottlenecks, latency issues, and errors.
* You can use X-Ray to trace requests across microservices, identify performance optimizations, and improve the overall reliability of your applications.

**5. Amazon CloudWatch Synthetics:**

* Amazon CloudWatch Synthetics is a monitoring service that enables you to create canaries to monitor your endpoints and APIs.
* Synthetics canaries simulate user interactions with your applications and endpoints, allowing you to proactively detect issues and ensure the availability and reliability of your services.

**6. AWS Config:**

* AWS Config is a service that provides automated inventory, configuration management, and compliance monitoring of your AWS resources.
* Config continuously tracks changes to resource configurations and provides a detailed inventory of your AWS environment.
* You can use Config rules to enforce compliance policies, detect configuration drift, and remediate non-compliant resources.

**42. What is Lambda and How it works?**

AWS Lambda is a serverless compute service provided by Amazon Web Services (AWS) that allows you to run code without provisioning or managing servers. Lambda automatically scales and manages the underlying infrastructure for you, allowing you to focus on writing code and building applications.

**43. What are the tools available in AWS to Migrate to Cloud from on-premises?**

AWS provides a variety of tools and services to help organizations migrate their workloads, applications, and data from on-premises environments to the cloud.

1. AWS Server Migration Service (SMS):

* AWS Server Migration Service (SMS) is an agentless service that makes it easier and faster for you to migrate on-premises servers to AWS.
* SMS allows you to automate the process of replicating your server volumes to AWS, converting them into Amazon Machine Images (AMIs), and launching EC2 instances from those AMIs.
* You can use SMS to migrate both virtualized and non-virtualized servers, including VMware vSphere and Microsoft Hyper-V environments.

2. AWS Database Migration Service (DMS):

* AWS Database Migration Service (DMS) helps you migrate your databases to AWS quickly and securely.
* DMS supports heterogeneous database migrations, including Oracle, SQL Server, MySQL, PostgreSQL, MongoDB, and others.
* You can use DMS to migrate databases from on-premises data centers, AWS RDS instances, or other cloud providers to Amazon RDS, Amazon Aurora, Amazon Redshift, or EC2 instances.

3. AWS DataSync:

* AWS DataSync is a data transfer service that makes it easy to move large amounts of data between on-premises storage systems and AWS.
* DataSync optimizes data transfer over the network, handles data integrity verification, and provides automation and scheduling capabilities for data migration tasks.
* You can use DataSync to migrate data from file systems, NAS appliances, or object storage systems to Amazon S3, Amazon EFS, or Amazon FSx for Windows File Server.

4. AWS Snow Family:

* AWS Snow Family consists of physical devices designed to help you transfer large amounts of data to and from AWS securely and efficiently.
* Snowball Edge and Snowmobile are ruggedized, tamper-resistant devices that you can use to migrate petabytes of data from on-premises environments to AWS.
* Snowball Edge also supports local compute and storage capabilities, allowing you to perform data processing and analytics tasks on the edge.

5. AWS Discovery Tools:

* AWS provides various discovery tools and partner solutions to help you discover and assess your on-premises environment before migration.
* Tools like AWS Application Discovery Service (ADS), AWS Migration Hub, and third-party solutions can help you identify applications, dependencies, and resource utilization to plan your migration strategy effectively.

6. AWS Partner Solutions:

* AWS has a vast ecosystem of consulting partners and independent software vendors (ISVs) that offer migration tools, services, and expertise to assist with your cloud migration journey.
* Partners provide specialized tools and services for application discovery, assessment, migration, and optimization, tailored to your specific requirements and use cases.

**44. What are the steps to Migrate to Cloud from on-premises?**

Migrating to the cloud from an on-premises environment involves several steps to ensure a smooth and successful transition.

**1. Assessment and Planning**:

* Assess your current on-premises environment, including applications, data, infrastructure, and dependencies.
* Identify workloads that are suitable for migration to the cloud based on factors such as complexity, performance requirements, and business priorities.
* Develop a migration strategy and roadmap outlining the sequence of migration, target cloud architecture, migration approach (lift-and-shift, re-platforming, re-architecting), and estimated timeline.

**2. Preparation:**

* Prepare your on-premises environment for migration by addressing any prerequisites or dependencies, such as updating software versions, optimizing configurations, and resolving compatibility issues.
* Determine the required resources, skills, and expertise needed for the migration project.
* Set up governance, security, and compliance policies for the cloud environment, including identity and access management (IAM), network security, encryption, and compliance requirements.

**3. Data Migration:**

* Migrate your data from on-premises storage systems to the cloud using tools and services such as AWS DataSync, AWS Database Migration Service (DMS), or AWS Snow Family devices.
* Assess data transfer requirements, data integrity, and encryption needs to ensure secure and reliable data migration.
* Validate data integrity and consistency after migration to confirm that data has been successfully transferred to the cloud.

**4. Application Migration:**

* Migrate your applications and workloads to the cloud using migration tools and services such as AWS Server Migration Service (SMS), AWS Database Migration Service (DMS), or third-party solutions.
* Choose the appropriate migration approach based on your application requirements and objectives (lift-and-shift, re-platforming, re-architecting).
* Test application migration in a non-production environment to identify and address any issues or compatibility issues before migrating production workloads.

**5. Testing and Validation:**

* Perform testing and validation of migrated applications and workloads to ensure they function correctly and meet performance and reliability requirements.
* Conduct performance testing, load testing, and user acceptance testing to validate application functionality and performance in the cloud environment.
* Verify that applications and services are integrated with other cloud services and components as intended.

**6. Optimization and Performance Tuning:**

* Optimize and fine-tune your cloud environment for performance, scalability, and cost-effectiveness.
* Implement best practices for resource optimization, auto-scaling, caching, and cost management to optimize the usage and cost of cloud resources.
* Monitor and analyze the performance of your applications and infrastructure in the cloud to identify opportunities for further optimization and improvement.

**7. Training and Knowledge Transfer:**

* Provide training and knowledge transfer to your IT teams and stakeholders on how to manage, operate, and optimize the cloud environment effectively.
* Ensure that your teams have the necessary skills and expertise to support and maintain cloud-based applications and infrastructure.

**8. Go-Live and Post-Migration Support:**

* Plan and execute the cutover to the cloud environment, including the migration of production workloads and services.
* Monitor the migration process closely and address any issues or challenges that arise during the cutover phase.
* Provide post-migration support and troubleshooting to address any issues or concerns that arise after the migration.
* Conduct post-migration reviews and evaluations to assess the success of the migration project and identify lessons learned for future migrations.

**45. What are the best and most branching strategies used by Devops Engineer?**

DevOps engineers often use various branching strategies based on the specific requirements of their project and the team collaboration preferences. Git is the distributed version control system, and GitHub is the popular Git repository hosting service.

**1. Gitflow Workflow:**

* Gitflow is a branching model that defines specific branches for different types of development and releases.
* It typically consists of two main branches: `master` (or `main`) for production-ready code and `develop` for ongoing development.
* Feature branches are created from the `develop` branch and merged back into `develop` once complete.
* Release branches are created from `develop` to prepare for a new release and undergo testing before being merged into both `master` (or `main`) and `develop`.
* Hotfix branches are created from `master` (or `main`) to address critical issues in production and are merged into both `master` (or `main`) and `develop`.

**2. GitHub Flow:**

* GitHub Flow is a lightweight branching model optimized for simplicity and continuous delivery.
* It consists of a single `master` (or `main`) branch where all development and releases occur.
* Developers create feature branches from `master` (or `main`), work on their features, and submit pull requests to merge changes back into `master` (or `main`).
* Each pull request undergoes code review and testing before being merged into `master` (or `main`), which triggers automated deployment to production.

**46. Can you write multistage docker file?**

A multi-stage Dockerfile is a Dockerfile that uses multiple stages to build and optimize Docker images. This technique helps reduce the size of the final Docker image by separating the build environment from the runtime environment and only including necessary dependencies and artifacts in the final image.

# Stage 1: Build Stage

FROM node:14 AS build

WORKDIR /app

COPY package.json package-lock.json ./

RUN npm install

COPY . .

RUN npm run build

# Stage 2: Runtime Stage

FROM nginx:alpine

COPY --from=build /app/build /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

In this example:

Stage 1 (Build Stage):

* Uses the node:14 image as the base image for the build environment.
* Sets the working directory to /app.
* Copies package.json and package-lock.json to the working directory and installs dependencies using npm install.
* Copies the application code to the working directory.
* Runs the build command (npm run build) to compile the application code into a production-ready bundle.

Stage 2 (Runtime Stage):

* Uses the nginx:alpine image as the base image for the runtime environment.
* Copies the built artifacts from the build stage (/app/build) to the default Nginx HTML directory (/usr/share/nginx/html).
* Exposes port 80 for incoming HTTP traffic.
* Defines the default command to start the Nginx server (nginx -g 'daemon off;').

When you build the Docker image using this multi-stage Dockerfile, Docker will execute both stages sequentially. The final Docker image will only contain the artifacts from the runtime stage.

To build the Docker image using Dockerfile: **docker build -t my-app-image .**

**47. What is the difference between playbook and Ad-hoc commands in Ansible?**

In Ansible, playbooks and ad-hoc commands are both used to automate tasks and manage infrastructure. playbooks are used for structured, repeatable automation tasks involving multiple steps and configurations, while ad-hoc commands are used for quick, one-time operations and ad-hoc tasks. Playbooks offer more flexibility, scalability, and maintainability for complex automation workflows, while ad-hoc commands provide convenience and immediacy for simple, on-demand tasks.

**Playbooks:**

* Playbooks are YAML files that define a series of tasks to be executed on remote hosts.
* Playbooks are used for configuration management, application deployment, and infrastructure orchestration.
* Playbooks can contain multiple plays, each consisting of one or more tasks.
* Tasks in playbooks are idempotent, meaning they can be run multiple times without causing unintended side effects.
* Playbooks support variables, loops, conditionals, handlers, and other advanced features to manage complex configurations and workflows.
* Playbooks are typically used for repeatable, structured automation tasks that require flexibility and maintainability.

**Ad-hoc Commands:**

* Ad-hoc commands are one-time, on-demand commands executed directly from the command line.
* Ad-hoc commands are used for quick tasks, troubleshooting, and one-off operations.
* Ad-hoc commands are executed using the ansible command-line tool with options to specify hosts, modules, and parameters.
* Ad-hoc commands are not idempotent by default, meaning they may have unintended consequences if run multiple times without proper precautions.
* Ad-hoc commands are suitable for tasks like gathering information, running simple commands, checking system status, and performing ad-hoc operations.
* Ad-hoc commands are less structured and less maintainable compared to playbooks, but they are convenient for quick tasks and immediate results.

**48. What are Roles in Ansible and those use cases?**

In Ansible, roles are a way to organize and structure playbooks and related files into reusable units. Roles provide a method for breaking down complex automation tasks into smaller, more manageable components, making it easier to organize, maintain, and reuse Ansible code across multiple projects.

Roles in Ansible offer a structured and modular approach to organizing and managing automation code, promoting reusability, abstraction, modularity, and consistency across projects. By leveraging roles, you can streamline your automation workflows, improve code maintainability, and scale your infrastructure and applications more effectively.

**49. What are the Handlers in Ansible?**

Handlers in Ansible are tasks that are triggered by other tasks, typically at the end of a playbook run, in response to specific events or conditions. Handlers are useful for managing services, restarting daemons, reloading configurations, or performing other actions that need to be executed only when certain changes occur.

Handlers are a powerful feature of Ansible that allow you to orchestrate actions in response to changes or events during playbook execution, ensuring consistency and reliability in your automation workflows.

**50. How many types of triggers are there in Jenkins and What are they?**

In Jenkins, triggers are mechanisms that initiate the execution of Jenkins jobs or pipelines based on certain events or conditions. Those are 1. SCM Trigger (Poll SCM) , 2. Scheduled (Build periodically) , 3. Webhook, 4. Upstream Trigger (Build after other projects are built), 5. Parameterized Trigger (Build with parameters), 6. Pipeline Trigger (Pipeline script from SCM, Pipeline webhook, etc.), 7. Remote Trigger (Remote API, CLI, etc.), 8. Manual Trigger (Build Now), 9. Trigger on Node Availability.

**51. Can you write a script for logs backup and monitoring?**

#!/bin/bash

# Configuration

LOG\_DIR="/var/log"

BACKUP\_DIR="/backup/logs"

THRESHOLD\_SIZE="100M" # Threshold size in megabytes

EMAIL="admin@example.com"

# Create backup directory if it doesn't exist

mkdir -p "$BACKUP\_DIR"

# Backup log files

echo "Backing up log files to $BACKUP\_DIR..."

rsync -av --delete "$LOG\_DIR/" "$BACKUP\_DIR/"

# Monitor log file sizes

echo "Monitoring log file sizes..."

for logfile in "$LOG\_DIR"/\*.log; do

size=$(du -m "$logfile" | awk '{print $1}')

if [[ $size -gt $(echo "$THRESHOLD\_SIZE" | sed 's/M//') ]]; then

echo "Log file $logfile exceeds threshold size: ${size}M"

# Send email notification

echo "Log file $logfile exceeds threshold size (${size}M). Please check." | mail -s "Log file size alert" "$EMAIL"

fi

done

echo "Log backup and monitoring complete."

**To use this script:**

* Modify the LOG\_DIR, BACKUP\_DIR, THRESHOLD\_SIZE, and EMAIL variables according to your environment.
* Save the script to a file (e.g., log\_backup\_monitor.sh) on your system.
* Make the script executable: chmod +x log\_backup\_monitor.sh.
* Optionally, schedule the script to run periodically using cron or any other task scheduler.

**This script performs the following tasks:**

* It backs up log files from the specified log directory (LOG\_DIR) to the backup directory (BACKUP\_DIR) using rsync.
* It then checks the size of each log file in the log directory.
* If any log file exceeds the specified size threshold (THRESHOLD\_SIZE), it sends an email notification to the specified email address (EMAIL).
* Finally, it prints a message indicating that the log backup and monitoring process is complete.

**52. What are the Linux commands you use regularly?**

As a DevOps engineer or system administrator working with Linux systems, there are several commands that I use regularly for managing and troubleshooting servers, applications, and infrastructure. Here are some commonly used Linux commands:

**1. File and Directory Management:**

* ls: List files and directories., **cd**: Change directory.
* pwd: Print working directory.
* mkdir: Create directories.,
* touch: Create empty files or update file timestamps.
* rm: Remove files or directories.
* cp: Copy files or directories.
* mv: Move or rename files or directories.

**2. File Content Viewing and Editing:**

* cat: Concatenate and display file content.
* less or more: View file content interactively.
* head: Display the beginning of a file.
* tail: Display the end of a file.
* vi or vim: Text editor for viewing and editing files.
* nano: Simple text editor for viewing and editing files.

**3. Process Management:**

* ps: Display information about running processes.
* top: Display dynamic real-time information about running processes and system resource usage.
* kill: Terminate or send signals to processes.
* killall: Terminate processes by name.
* pgrep: List processes based on name or other attributes.
* pkill: Send signals to processes based on name or other attributes.

**4. User and Permission Management:**

* useradd / adduser: Add a user.
* userdel: Delete a user.
* passwd: Change user password.
* su: Switch user.
* sudo: Execute commands with superuser privileges.
* chmod: Change file permissions.
* chown: Change file ownership.
* chgrp: Change group ownership.

**5. Networking:**

* ifconfig / ip: Display and configure network interfaces.
* ping: Send ICMP echo requests to network hosts.
* traceroute / tracepath: Trace the route packets take to a network host.
* netstat / ss: Display network connections, routing tables, and network statistics.
* iptables / firewalld: Configure firewall rules.
* curl / wget: Download files from the internet.

**6. System Information and Monitoring:**

* uname: Display system information.
* hostname: Display or set system hostname.
* df: Display disk space usage.
* du: Display disk usage of files and directories.
* free: Display memory usage.
* uptime: Display system uptime.
* dmesg: Display kernel ring buffer messages.

**7. Package Management:**

* apt / apt-get: Package management commands for Debian-based systems.
* yum / dnf: Package management commands for Red Hat-based systems.
* rpm: Package management command for installing, querying, and managing RPM packages.

**53. How many stages are there in Jenkins pipeline and What are they?**

1. Checkout: Retrieves the source code from version control.

2. Build: Compiles, builds, or packages the application.

3. Test: Executes automated tests, such as unit tests, integration tests, and functional tests.

4. Static Code Analysis: Analyzes the codebase for code quality issues, style violations, and security vulnerabilities.

5. Artifact Archiving:Archives built artifacts for later deployment.

6. Deployment to Dev: Deploys the application to the development environment for testing.

7. Integration Testing: Performs integration testing to ensure components work together correctly.

8. Deployment to QA: Deploys the application to the QA environment for further testing.

9. Security Scanning: Scans the application for security vulnerabilities and compliance issues.

10. Performance Testing: Conducts performance testing to evaluate system performance under load.

11. Deployment to Staging: Deploys the application to the staging environment for final validation.

12. User Acceptance Testing (UAT): Executes user acceptance tests to validate the application against user requirements.

13. Deployment to Production: Deploys the application to the production environment for release.

14. Smoke Testing: Performs basic tests to verify that critical functionalities are working after deployment.

15. Rollback: Rolls back changes in case of deployment failures or issues.

16. Cleanup: Performs cleanup tasks, such as removing temporary files or stopping services.

17. Notification: Notifies stakeholders about the build status or deployment status.

**54. What are the libraries or variables are there in Jenkins?**

In Jenkins, there are several built-in libraries, variables, and objects that you can use to customize and enhance your pipelines. These libraries and variables provide access to various functionalities, metadata, and environment information within your Jenkins Pipeline.

**55. what are the Jenkins libraries?**

A shared library in Jenkins is a reusable collection of Groovy scripts that can be used by multiple Jenkins jobs. This allows you to share code and functionality between different jobs, which can make your builds more efficient and easier to maintain.

Shared Libraries are typically organized within a version-controlled repository, such as Git.

In Jenkins, **go to Manage Jenkins** → Configure System. Under Global Pipeline Libraries, add a library with the following settings: Name: pipeline-library-demo. Default version: Specify a Git reference (branch or commit SHA), e.g. master.

**56. What are the libraries or variables are there in Jenkins?**

In Jenkins Pipeline, you have access to various libraries, variables, and objects that empower you to customize and optimize your pipelines. Here's a concise overview of the essential ones:

**1. Global Variables:**

* **env**: Provides access to environment variables defined in the Jenkins environment.
* **params**: Contains parameters passed to the pipeline from external sources like build or pipeline parameters.
* **currentBuild**: Represents the ongoing build with properties such as build number, status, and parameters.
* **pipeline**: Offers access to pipeline-specific methods and functions.

**2. Pipeline Steps:** Jenkins Pipeline offers a comprehensive set of built-in steps for performing various tasks within your pipeline. These include actions like source code management, building, testing, deployment, and notification.

**3. Pipeline Syntax**: Jenkins Pipeline supports both declarative and scripted syntax for defining pipelines. You can use keywords, directives, and functions specific to each syntax for defining stages, steps, and behavior.

**4. Shared Libraries**: Shared libraries enable you to define reusable code and functions shared across multiple pipelines. They can contain custom steps, utilities, helper functions, and shared logic, enhancing maintainability and reusability.

**5. Pipeline Global Library Variables**: Global variables, configurable in Jenkins global configuration or loaded from external scripts, offer flexibility and consistency across pipelines. They are ideal for defining common configurations, credentials, utilities, or customizations.

**6. Pipeline Environment Variables**: Access to environment variables provides valuable information about the Jenkins environment, build context, and execution. These variables include build parameters, status, workspace directory, and Git commit details.

Utilizing these libraries, variables, and objects effectively enables you to create robust, flexible, and maintainable pipelines tailored to your specific requirements and use cases in Jenkins Pipeline.

**57. Can you write Kubernetes manifest file?**

Below is an example of a simple Kubernetes manifest file for deploying a basic web application using a Deployment and a Service.

# deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: webapp-deployment

spec:

replicas: 3 # Number of desired replicas

selector:

matchLabels:

app: webapp

template:

metadata:

labels:

app: webapp

spec:

containers:

- name: webapp-container

image: your-webapp-image:latest # Replace with your actual Docker image

ports:

- containerPort: 80 # Port your web application is listening on

---

# service.yaml

apiVersion: v1

kind: Service

metadata:

name: webapp-service

spec:

selector:

app: webapp

ports:

- protocol: TCP

port: 80 # Port to expose on the service

targetPort: 80 # Port your application is listening on in the pods

type: ClusterIP # Type of service - ClusterIP, NodePort, LoadBalancer, or ExternalName

You can save this YAML file as deployment.yaml and service.yaml. Then you can apply these manifests using the kubectl apply commands: **kubectl apply -f deployment.yaml** & **kubectl apply -f service.yaml**

This will create a Deployment with three replicas of your web application pods and a Service to expose your application within the Kubernetes cluster.

Make sure to replace "your-webapp-image:latest" with the actual name and tag of your Docker image. Additionally, you can customize other fields in the manifests according to your requirements, such as resource limits, environment variables, and labels.

**58. Can you write a dockerfile?**

Below is a basic example of a Dockerfile for a Node.js web application.

**FROM node:14** # Use an official Node.js runtime as the base image with version 14.

**WORKDIR /usr/src/app** # Set the working directory inside the container.

**COPY package\*.json ./** # Copy package.json and package-lock.json to the working directory

**RUN npm install** # Install dependencies.

**COPY . .** # Copy the rest of the application code to the working directory

**EXPOSE 3000** # Expose the port your app runs on. Docker container will listen on port 3000 at runtime.

**CMD ["npm", "start"]** # Specifies the Command to run the application when the container starts.

:wq!

This assumes that your package.json file has a script called "start" to start the application.

To build a Docker image using Dockerfile, from Dockerfile directory: **docker build -t your-image-name .**

Replace "your-image-name" with the desired name for your Docker image.

Once the image is built, you can run a container: **docker run -p 3000:3000 your-image-name**

This command will start a container based on the image you built, mapping port 3000 on your host machine to port 3000 inside the container. Adjust the port mapping as needed based on your application configuration.

**59. Can you write a docker file to deploy a nginx web application?**

Below is an example of a Dockerfile for deploying a simple NGINX web application.

**FROM nginx:latest** # Use the official NGINX base image

# Copy the content of the local 'html' directory to the NGINX default public directory

**COPY html/ /usr/share/nginx/html**

**EXPOSE 80** # Expose port 80 to allow external access

**CMD ["nginx", "-g", "daemon off;"]** # Command to start NGINX when the container starts.

:wq!

**To use this Dockerfile:**

* Create a directory named html in the same directory as the Dockerfile.
* Place your HTML, CSS, JavaScript, or any other static web content files inside the html directory.
* Build the Docker image using the following command in the terminal (make sure you're in the directory containing the Dockerfile): **docker build -t my-nginx-app** .

Once the image is built, you can run a container using: **docker run -d -p 8080:80 my-nginx-app**

This will start a container running NGINX and serving your static web content on port 8080 of your host machine. You can access the web application by visiting http://localhost:8080 in your web browser.

**60. Can you please write a playbook in Ansible?**

Below is an example of an Ansible playbook for deploying a simple web application to a group of servers:

# deploy\_web\_app.yaml

---

- name: Deploy web application

hosts: web\_servers # Group of servers to deploy the application to

become: yes # Enable privilege escalation (sudo)

tasks:

- name: Install required packages

apt:

name:

- nginx

- git

state: present # Ensure the packages are installed

- name: Clone web application repository

git:

repo: https://github.com/example/web-app.git # URL of your application's Git repository

dest: /var/www/web-app # Destination directory to clone the repository

version: master # Branch or tag to checkout

- name: Configure Nginx

template:

src: nginx.conf.j2 # Path to the Jinja2 template file for Nginx configuration

dest: /etc/nginx/sites-available/web-app # Destination path for the Nginx configuration file

notify:

- Restart Nginx # Notify the handler to restart Nginx if the configuration changes

handlers:

- name: Restart Nginx

service:

name: nginx

state: restarted # Restart the Nginx service

In this playbook:

* The playbook is named deploy\_web\_app.yaml. and It targets a group of servers named web\_servers.
* Privilege escalation (sudo) is enabled (become: yes) to execute tasks that require root permissions.
* The playbook consists of three tasks: Install required packages (nginx and git) using the apt module. Clone the web application repository from a Git repository specified by the repo parameter using the git module. Configure Nginx by copying a Jinja2 template file (nginx.conf.j2) to the server and notifying the handler to restart Nginx if the configuration changes.
* The playbook includes a handler named Restart Nginx that restarts the Nginx service if the configuration changes.

You need to create a Jinja2 template file for Nginx configuration (nginx.conf.j2) and place it in the same directory as the playbook. Replace "https://github.com/example/web-app.git" with the actual URL of your web application's Git repository. To execute the playbook: **ansible-playbook deploy\_web\_app.yaml**

**61. Can you write a terraform code to provisioning AWS Services?**

Below is an example of Terraform code to provision various AWS services, including an S3 bucket, an IAM user, an IAM policy & EC2 Instances:

# main.tf

provider "aws" {

region = "us-west-2" # Specify the AWS region

}

# Create an S3 bucket

resource "aws\_s3\_bucket" "example\_bucket" {

bucket = "example-bucket-name" # Specify the name of the bucket

acl = "private" # Set the bucket ACL (Access Control List)

}

# Create an IAM user

resource "aws\_iam\_user" "example\_user" {

name = "example-user" # Specify the name of the IAM user

}

# Attach a policy to the IAM user

resource "aws\_iam\_user\_policy\_attachment" "example\_user\_policy\_attachment" {

user = aws\_iam\_user.example\_user.name

policy\_arn = "arn:aws:iam::aws:policy/AmazonS3FullAccess" # Attach an existing policy (AmazonS3FullAccess) to the IAM user

}

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0" # Specify the AMI ID for the EC2 instance

instance\_type = "t2.micro" # Specify the instance type

tags = {

Name = "ExampleInstance" # Name tag for the instance

}

}

**In this Terraform configuration:**

* The provider block specifies the AWS provider and the region where the resources will be provisioned.
* The aws\_s3\_bucket resource creates an S3 bucket named "example-bucket-name" with a private ACL.
* The aws\_iam\_user resource creates an IAM user named "example-user".
* The aws\_iam\_user\_policy\_attachment resource attaches an existing policy (AmazonS3FullAccess) to the IAM user.

Before running this code, make sure you have the AWS credentials configured on your system. You can set them using environment variables (AWS\_ACCESS\_KEY\_ID and AWS\_SECRET\_ACCESS\_KEY) or AWS shared credentials file (~/.aws/credentials).

**To provision these AWS services, follow these steps:**

Initialize the Terraform working directory: **terraform init**

Review the execution plan to ensure that Terraform will create the desired resources: **terraform plan**

Apply the Terraform configuration to create the resources: **terraform apply**

Confirm by typing **yes** when prompted.

Terraform will provision the specified AWS services as defined in the configuration. Once the process is complete, you can verify the creation of the resources in the AWS Management Console or by using the AWS CLI.

**62. Can you write a Jenkins pipeline?**

This pipeline script defines stages for building, testing, and deploying a simple web application:

pipeline {

agent any // Executes the pipeline on any available agent

environment { // Define environment variables

APP\_NAME = 'my-web-app'

DOCKER\_REGISTRY = 'docker.example.com'

DOCKER\_IMAGE = "${DOCKER\_REGISTRY}/${APP\_NAME}"

GIT\_REPO = 'https://github.com/example/my-web-app.git'

}

stages {

stage('Checkout') {

steps {

git branch: 'master', url: GIT\_REPO // Checkout the source code from Git repository

}

}

stage('Build') {

steps { // Build the Docker image

script {

docker.build(DOCKER\_IMAGE)

}

}

}

stage('Test') {

steps { // Run tests (replace with actual test commands)

sh 'echo "Running tests..."'

}

}

stage('Deploy') {

steps { // Push the Docker image to the registry

script {

docker.withRegistry('https://${DOCKER\_REGISTRY}', 'credentials-id') {

docker.image(DOCKER\_IMAGE).push('latest')

}

} // Deploy the Docker image (replace with actual deployment steps)

sh 'echo "Deploying ${DOCKER\_IMAGE}..."'

}

}

}

post {

success { // Notification for successful build

echo 'Deployment successful. Notifying the team...'

}

failure { // Notification for failed build

echo 'Deployment failed. Notifying the team...'

}

}

}

**63. Can you write simple CICD pipeline script?**

Below is a simple CI/CD pipeline script using Jenkins declarative pipeline syntax. This pipeline script automates the process of building, testing, and deploying a web application:

pipeline {

agent any // Executes the pipeline on any available agent

environment { // Define environment variables

APP\_NAME = 'my-web-app'

DOCKER\_REGISTRY = 'docker.example.com'

DOCKER\_IMAGE = "${DOCKER\_REGISTRY}/${APP\_NAME}"

GIT\_REPO = 'https://github.com/example/my-web-app.git'

}

stages {

stage('Checkout') {

steps { // Checkout the source code from Git repository

git branch: 'master', url: GIT\_REPO

}

}

stage('Build') {

steps { // Build the Docker image

script {

docker.build(DOCKER\_IMAGE)

}

}

}

stage('Test') {

steps { // Run tests (replace with actual test commands)

sh 'echo "Running tests..."'

}

}

stage('Deploy') {

steps { // Push the Docker image to the registry

script {

docker.withRegistry('https://${DOCKER\_REGISTRY}', 'credentials-id') {

docker.image(DOCKER\_IMAGE).push('latest')

}

} // Deploy the Docker image (replace with actual deployment steps)

sh 'echo "Deploying ${DOCKER\_IMAGE}..."'

}

}

}

post {

success { // Notification for successful build

echo 'Deployment successful. Notifying the team...'

}

failure { // Notification for failed build

echo 'Deployment failed. Notifying the team...'

}

}

}

**Git**: Git is a distributed version control system (DVCS) used for tracking changes in source code during software development. It allows multiple developers to collaborate on a project efficiently by managing different versions of files, tracking changes, and facilitating team collaboration.

git config --list: Lists all Git configuration settings.

git config --global user.name "Your Name": Sets the user name globally.

git config --global user.email "youremail@example.com": Sets the email address globally.

git config --local user.name "Your Name": Sets the user name for the current repository.

git config --local user.email "youremail@example.com": Sets the email address for the current repository.

git init: Initializes a new Git repository in the current directory.

git clone <repository\_url>: Clones a repository from a remote URL to the local machine.

git branch: Lists all local branches.

git branch <branch\_name>: Creates a new branch.

git checkout <branch\_name>: Switches to the specified branch.

git checkout -b <branch\_name>: Creates a new branch and switches to it.

git merge <branch\_name>: Merges changes from the specified branch into the current branch.

git branch -d <branch\_name>: Deletes the specified branch.

git branch -m <old\_branch\_name> <new\_branch\_name>: Renames a branch.

git add <file\_name> or git add -A: Add the files to the staging area.

git commit -m "Commit message": Commits changes in the staging area to the repository.

git commit --amend: Amends the last commit by adding changes to it.

git reset HEAD <file\_name>: Unstages a file from the staging area.

git log: Displays the commit history.

git log --graph --oneline --decorate: Displays a compact representation of the commit graph.

git log --author="<author\_name>": Displays commits by a specific author.

git diff: Shows changes between commits, commit and working tree, etc.

git cherry-pick <commit\_id>: Applies the changes introduced by the specified commit to the current branch.

git remote add <remote\_name> <remote\_url>: Adds a new remote repository.

git remote -v: Lists all remote repositories.

git push <remote\_name> <branch\_name>: Pushes local changes to a remote repository.

git push -u origin <branch\_name>: Sets up the upstream branch and pushes changes to the remote repository.

git pull <remote\_name> <branch\_name>: Fetches changes from a remote repository and merges them into the current branch.

git fetch <remote\_name>: Fetches changes from a remote repository but does not merge them.

git remote rm <remote\_name>: Removes a remote repository.

git tag <tag\_name>: Creates a lightweight tag for the current commit.

git tag -a <tag\_name> -m "Tag message": Creates an annotated tag with a message.

git tag -d <tag\_name>: Deletes a tag.

git status: Shows the status of the working directory and staging area.

git diff <file\_name>: Shows the differences between the working directory and the staging area.

Resolve conflicts manually in files marked with conflict markers (<<<<<<<, =======, >>>>>>>).

git stash: Temporarily shelves changes so you can work on another task.

git stash list: Lists all stashed changes.

git stash apply: Applies the most recent stash to the current working directory.

git stash pop: Removes the most recent stash and applies it to the current working directory.

git stash drop: Discards the most recent stash without applying it.

git revert <commit\_id>: Creates a new commit that undoes the changes introduced by the specified commit.

git reset --hard <commit\_id>: Resets the current branch to a specific commit, discarding all changes after that commit.

git reset --soft <commit\_id>: Resets the current branch to a specific commit, keeping changes staged.

.gitignore: A file containing patterns for files and directories to be ignored by Git.

git rebase -i <base\_commit>: Initiates an interactive rebase session to squash, edit, reorder, or drop commits.

git rebase <branch\_name>: Applies changes from the specified branch onto the current branch, rewriting commit history.

git lfs install: Installs Git LFS globally on your system.

git lfs track "<file\_pattern>": Tracks large files using Git LFS.

git lfs ls-files: Lists the files tracked by Git LFS.

git lfs pull: Fetches and downloads files tracked by Git LFS.

git bisect start: Initiates the binary search to find the commit that introduced a bug.

git bisect bad: Marks the current commit as bad (contains the bug).

git bisect good <commit\_id>: Marks a known good commit.

git bisect reset: Ends the bisect session and returns to the original branch.

git worktree add <path> <branch>: Creates a new worktree at the specified path and checks out the specified branch.

git worktree list: Lists all the worktrees associated with the repository.

git worktree prune: Cleans up stale worktrees.

Git Workflows:

Feature Branch Workflow: Creating branches for new features, making changes, and merging them back into the main branch.

Gitflow Workflow: A branching model for larger projects, consisting of main branches (master, develop) and supporting branches (feature, release, hotfix).

Forking Workflow: Used in open-source projects, where contributors fork the main repository, make changes in their fork, and submit pull requests to the main repository.

Git Hooks:

Pre-commit hook: Executes before a commit is created. Useful for code linting, formatting checks, etc.

Post-commit hook: Executes after a commit is created. Can be used for notifications, triggering builds, etc.

Pre-push hook: Executes before pushing commits to a remote repository. Useful for running tests, ensuring code quality, etc.

Git Reflog: git reflog: Lists the reference logs, showing the history of changes to the HEAD reference.

Git Aliases:

git config --global alias.<alias\_name> "<command>": Defines a custom alias for a Git command.

Example: git config --global alias.co checkout creates an alias co for the git checkout command.

**Jenkins:** Jenkins is an open-source automation server widely used in DevOps practices for continuous integration (CI) and continuous delivery (CD) pipelines. It helps automate various stages of software development and deployment processes, allowing teams to build, test, and deploy applications more efficiently.

Jenkins is like a digital assistant that helps software development teams work together more smoothly. It's a tool that automates many tasks involved in building, testing, and deploying software.

**Installation and Setup**: Download and install Jenkins on your server or local machine. Configure Jenkins with necessary plugins and settings.

**Creating Jobs**: Jobs in Jenkins are like to-do lists for your assistant. You create a new job for each task you want Jenkins to do, like building or testing your software. Create a new Jenkins job for a specific task, such as building, testing, or deploying an application.

**Example:** Creating a Freestyle project job: node {

stage('Build') {

// Your build steps here

}

}

**Pipeline as Code**: Define pipelines using Jenkinsfile written in Groovy syntax for automated workflows.

Example:

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'echo "Building..."'

}

}

stage('Test') {

steps {

sh 'echo "Testing..."'

}

}

stage('Deploy') {

steps {

sh 'echo "Deploying..."'

}

}

}

}

**Integrating Source Control**: Jenkins can automatically get the latest version of your software from a place where it's stored, like GitHub. It's like your assistant fetching the latest materials for your project. Configure Jenkins to pull source code from Git repositories.

Example: Configuring a Jenkins job to pull source code from a Git repository: Go to job configuration -> Source Code Management -> Git -> Repository URL.

**Build Triggers**: You can tell Jenkins when to start working on a job. For example, you can ask it to start whenever there's a new change in your software. It's like setting an alarm for your assistant to start working. Configure triggers to start Jenkins jobs automatically based on events like code commits, time intervals, or webhook notifications. **Example**: Configuring a Jenkins job to trigger on SCM changes: Go to job configuration -> Build Triggers -> Poll SCM.

**Build Environment**: Jenkins can create a special workspace with everything needed to do its tasks, like tools and settings. It's like preparing a workstation for your assistant with all the necessary tools. Set up build environments, including build tools, dependencies, and environment variables.

**Example:** Configuring environment variables in a Jenkins job: Go to job configuration -> Build Environment -> Set environment variables.

**Artifacts and Archives**: Archive build artifacts for later use or deployment. After Jenkins finishes a job, it can save important files or information for later. It's like your assistant putting finished work in a special folder for safekeeping.

**Example**: Archiving artifacts in a Jenkins job: Add a post-build action -> Archive the artifacts.

**Testing:** Jenkins can run tests on your software automatically to make sure everything works as expected. It's like your assistant checking your work to catch any mistakes. Integrate automated testing frameworks into Jenkins pipelines to ensure code quality.

**Example:** Running unit tests in a Jenkins pipeline:

stage('Test') {

steps {

sh 'npm test'

}

}

**Deployment**: Jenkins can deploy your software to different places, like a website or server, once it's ready. It's like your assistant delivering your finished work to the right destination. Automate deployment processes to various environments (e.g., staging, production).

**Example**: Deploying to a server in a Jenkins pipeline:

stage('Deploy') {

steps {

sh 'ssh user@server "deploy\_script.sh"'

}

}

**Monitoring and Notifications:** Jenkins can keep an eye on its tasks and let you know if something goes wrong. It's like your assistant sending you a message if it runs into any problems. Set up monitoring for Jenkins jobs and receive notifications on build status.

Example: Configuring email notifications for Jenkins job: Go to job configuration -> Add post-build action -> Editable Email Notification.

**Security and Access Control:** You can control who can use Jenkins and what they're allowed to do. It's like giving keys to your assistant and deciding which rooms it can enter. Configure security settings to control user access and permissions.

Example: Setting up authentication and authorization in Jenkins: Go to Jenkins dashboard -> Manage Jenkins -> Configure Global Security.

**Scaling and High Availability:** Set up Jenkins for scalability and high availability to handle large workloads. Jenkins can work with other assistants to handle it all. It's like hiring more assistants to get the job done faster. **Example:** Setting up Jenkins Master-Slave architecture: Configure Jenkins Master -> Add Jenkins Slaves for distributed builds.

**Ansible:** Ansible is an open-source automation tool used for configuration management, application deployment, orchestration, and task automation. It simplifies complex tasks like deploying applications, managing server configurations, and coordinating software updates across multiple machines. Ansible uses simple YAML-based configuration files and SSH to communicate with remote machines.

**Inventory:** An inventory file lists all the servers or nodes you want to manage with Ansible.

Example Inventory File (inventory.ini): **ansible-playbook -i inventory.yml deploy.yml**

[web\_servers]

server1.example.com

server2.example.com

[db\_servers]

db1.example.com

**Playbooks**: Playbooks are YAML files that define a set of tasks to be executed by Ansible.

Example Playbook (deploy.yml): **ansible-playbook deploy.yml**

---

- name: Deploy web application

hosts: web\_servers

tasks:

- name: Copy application files

copy:

src: /path/to/application

dest: /var/www/html

- name: Restart Apache

service:

name: apache2

state: restarted

**Tasks:** Tasks are individual actions performed by Ansible on managed hosts.

Example Task in Playbook:

- name: Copy application files

copy:

src: /path/to/application

dest: /var/www/html

**Modules:** Ansible modules are reusable units of code that perform specific tasks on managed hosts.

Example Module (copy module):

- name: Copy files to a remote server

copy:

src: /path/to/local/file

dest: /path/on/remote/server

**Roles**: Roles are collections of playbooks, tasks, variables, and files organized into a directory structure for better organization and reusability.

Example Role Directory Structure:

roles/

└── webserver

├── tasks

│ └── main.yml

├── handlers

│ └── main.yml

├── templates

├── files

├── vars

│ └── main.yml

└── defaults

└── main.yml

**Variables:** Ansible allows you to define variables that can be used in playbooks or roles to customize behavior.

Example Variable Definition: my\_var: "some\_value"

**Handlers:** Handlers are tasks triggered by other tasks when a change is made, typically to restart services or perform similar actions.

Example Handler:

- name: Restart Apache

service:

name: apache2

state: restarted

**Tags**: Tags allow you to categorize tasks and run only those tasks with specific tags.

Example Tag in Playbook:

- name: Install Apache

apt:

name: apache2

state: present

tags:

- install

**Conditionals:** Ansible supports conditional statements to execute tasks based on predefined conditions.

Example Conditional Task:

- name: Install Apache on Debian

apt:

name: apache2

state: present

when: ansible\_distribution == "Debian"

**Vault:** Ansible Vault is a tool for encrypting sensitive data (like passwords or API keys) within Ansible playbooks or roles.

Example Vault Usage: ansible-vault encrypt vars/secrets.yml

Ad-hoc Execution Commands:

Executes ad-hoc commands on remote hosts without the need for a playbook. Example: ansible all -m ping

Specifies a module to execute on remote hosts. Example: ansible all -m shell -a "echo hello"

Specifies the inventory file to use for targeting hosts. Example: ansible -i inventory.yml all -m ping

Specifies the remote user to use for executing commands. Example: ansible all -u username -m ping

Executes commands with escalated privileges (sudo). Example: ansible all -b -m shell -a "reboot"

**Terraform:**

Terraform is an open-source infrastructure as code (IaC) tool created by HashiCorp. It allows users to define and provision infrastructure resources such as virtual machines, networks, storage, and more using a declarative configuration language. Terraform provides a simple and consistent way to manage infrastructure across different cloud providers and on-premises environments.

Infrastructure as Code (IaC): Terraform uses a declarative configuration language to define infrastructure in code.

Example Terraform configuration file (main.tf):

provider "aws" {

region = "us-east-1"

}

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

}

Providers: Providers are plugins that Terraform uses to interact with infrastructure APIs from cloud providers, such as AWS, Azure, Google Cloud, etc.

Example provider block in Terraform configuration:

provider "aws" {

region = "us-east-1"

}

Resources: Resources are the building blocks of infrastructure defined in Terraform configurations. They represent cloud components like virtual machines, databases, networks, etc.

Example resource block for an AWS EC2 instance:

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

}

**Variables:** Terraform allows you to define variables to parameterize your configurations and make them reusable.

Example variable definition in Terraform configuration:

variable "instance\_type" {

description = "The type of EC2 instance to launch"

default = "t2.micro"

}

Usage of variables in resource configuration:

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = var.instance\_type

}

**Outputs:** Outputs are values that Terraform displays after applying the configuration. They can be used to provide information about resources provisioned by Terraform.

Example output block in Terraform configuration:

output "public\_ip" {

value = aws\_instance.example.public\_ip

}

**Modules:** Modules are self-contained packages of Terraform configurations that can be reused across different projects to encapsulate infrastructure components.

Example module structure:

modules/

└── vpc

├── main.tf

├── variables.tf

├── outputs.tf

**State Management:**

Terraform maintains a state file (terraform.tfstate) that keeps track of the current state of managed infrastructure.

Commands for state management:

terraform init: Initializes Terraform and downloads providers and modules.

terraform plan: Generates an execution plan showing what Terraform will do.

terraform apply: Applies the changes required to reach the desired state of the configuration.

terraform destroy: Destroys the infrastructure defined in the configuration.

**Terraform Cloud and Enterprise:**

Terraform Cloud and Terraform Enterprise are managed services that provide collaboration, governance, and self-service infrastructure provisioning features for Terraform users.

Example usage:

Collaborative workspace for teams to manage Terraform configurations.

Policy enforcement and compliance checks.

**Provisioner**s: Provisioners in Terraform allow you to run scripts or commands on the provisioned resources after creation or destruction. They are useful for tasks like software installation or configuration management.

Example provisioner in resource configuration:

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

provisioner "local-exec" {

command = "echo 'Instance provisioned!'"

}

}

**Data Sources:**

Data sources allow Terraform to fetch information from external sources such as cloud providers, APIs, or other Terraform configurations. They provide dynamic information that can be used in resource configurations.

Example data source for fetching AWS AMI IDs:

data "aws\_ami" "ubuntu" {

most\_recent = true

filter {

name = "name"

values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-\*"]

}

owners = ["099720109477"] # Canonical

}

Terraform Modules Registry: The Terraform Module Registry is a public repository of reusable Terraform modules contributed by the community. It allows users to discover, share, and collaborate on infrastructure configurations.

Example usage to use a module from the registry:

module "vpc" {

source = "terraform-aws-modules/vpc/aws"

version = "2.0.0"

# Configuration parameters

...

}

Remote State Storage:

Terraform state files can be stored remotely to enable collaboration and prevent state corruption. Various backends like Amazon S3, Azure Blob Storage, Google Cloud Storage, and Terraform Cloud can be used for remote state storage.

Example configuration to use Amazon S3 for remote state storage:

terraform {

backend "s3" {

bucket = "terraform-state-bucket"

key = "terraform.tfstate"

region = "us-west-2"

}

}

Terraform Providers Development: Terraform providers are responsible for managing resources and communicating with APIs of different infrastructure providers. You can develop custom providers to extend Terraform's capabilities for managing specialized infrastructure.

Example development workflow for creating a custom provider.

**Terraform Enterprise Features**: Terraform Enterprise offers additional features such as Sentinel policy as code, role-based access control (RBAC), and integration with version control systems for advanced governance and collaboration.

Example usage of Sentinel policies to enforce compliance rules.

**Terraform CLI Plugins:** Terraform CLI supports third-party plugins to extend its functionality. Plugins can add new commands, data sources, provisioners, and more.

Example usage of a third-party plugin to enhance Terraform's capabilities.

Remote Execution: Terraform supports remote execution modes where the Terraform CLI runs commands against a remote backend, such as Terraform Cloud or a custom remote backend.

Example usage of remote execution:

terraform plan -out=tfplan

terraform apply tfplan

**Workspaces**: Workspaces in Terraform allow you to manage multiple distinct sets of infrastructure configurations within the same root module. Each workspace has its own state file.

Example usage of workspaces:

terraform workspace new staging

terraform workspace select staging

**Interpolation and Functions**: Terraform supports interpolation syntax and built-in functions to manipulate values and dynamically generate configuration.

Example usage of interpolation:

resource "aws\_instance" "example" {

tags = {

Name = "instance-${count.index}"

}

}

State Locking: Terraform state locking prevents concurrent modifications to state files, ensuring consistency and preventing data corruption in collaborative environments.

Example usage of state locking with a lock file stored in a shared location.

Workspace Variables: Terraform allows you to define workspace-specific variables, which can have different values for each workspace.

Example usage of workspace-specific variables:

variable "instance\_type" {

type = string

description = "The type of EC2 instance to launch"

default = "t2.micro"

workspace\_default = {

staging = "t2.small"

production = "t2.large"

}

}

**External Data Sources:** Terraform allows you to fetch data from external sources, such as APIs or databases, and use it in your configurations.

Example usage of an external data source to fetch data from a REST API.

**Custom Providers Development:** Advanced users can develop custom Terraform providers to manage resources not supported by built-in providers or to integrate with proprietary systems.

Example development workflow for creating a custom provider.

**Terraform HCL (HashiCorp Configuration Language) Syntax**: Understanding the syntax and best practices of HCL for writing concise and maintainable Terraform configurations.

Example usage of HCL syntax features like blocks, arguments, and expressions.

**what is the difference between terraform providers and terraform provisioners?**

Terraform Providers and Terraform Provisioners are both essential components of Terraform, but they serve different purposes and operate at different stages of the infrastructure lifecycle:

**Terraform Providers:**

**Purpose**: Terraform Providers are responsible for managing and interacting with the resources offered by various infrastructure providers, such as cloud providers (AWS, Azure, GCP), on-premises platforms (VMware, OpenStack), or software services (Kubernetes, Docker).

**Functionality:** Providers translate Terraform configuration files into API calls specific to each infrastructure provider, enabling the creation, update, and deletion of resources. They handle resource lifecycle management, including CRUD operations (Create, Read, Update, Delete), and ensure the desired state of the infrastructure aligns with the configuration.

**Example:** The AWS provider allows Terraform to manage resources like EC2 instances, S3 buckets, VPCs, and more within the AWS cloud environment.

**Terraform Provisioners:**

**Purpose**: Terraform Provisioners are used to execute scripts or commands on local or remote resources during the creation or destruction phase of infrastructure provisioning. They are primarily used for tasks such as software installation, configuration management, or post-deployment initialization.

**Functionality:** Provisioners run after resource creation or destruction, but before state is written. They can be configured to run either locally on the machine running Terraform (local-exec provisioner) or remotely on the provisioned resources via SSH or WinRM connections (remote-exec provisioner). Provisioners are typically used when a resource provider does not offer built-in configuration capabilities.

**Example:** Using a provisioner to execute a shell script on an EC2 instance after it has been created to install additional software packages or perform custom configurations.

**Key Differences:**

**Responsibility:** Providers manage the lifecycle of infrastructure resources and interact directly with APIs of infrastructure providers. Provisioners execute scripts or commands on provisioned resources to perform additional configuration or setup tasks.

**Scope:** Providers are focused on managing resources and ensuring their desired state matches the configuration. Provisioners are focused on executing scripts or commands for configuration tasks not handled by the provider.

**Execution Time:** Providers execute during the Terraform apply phase to create, update, or delete resources. Provisioners execute after resource creation or destruction but before Terraform updates its state.

**Docker:**

Docker is a platform for developing, shipping, and running applications inside containers. Containers allow developers to package an application with all of its dependencies into a standardized unit for software development. Docker provides tools and a platform to manage these containers efficiently.

1. **Docker Image**: An image is a lightweight, standalone, and executable software package that includes everything needed to run a piece of software, including the code, runtime, libraries, environment variables, and configuration files. Example command to pull a Docker image: **docker pull nginx**

2. **Docker Container**: A container is a runtime instance of a Docker image. It encapsulates an application and its dependencies, isolating it from the host system and other containers. Example command to run a Docker container: **docker run -d --name mynginx nginx**

3. **Dockerfile:** A Dockerfile is a text file that contains instructions for building a Docker image. It defines the environment inside the container and what should be included in the image. Example Dockerfile for a simple Node.js application: Dockerfile

FROM node:14

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

CMD ["node", "index.js"]

4. **Docker Compose**: Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file to configure the application's services and dependencies. Example `docker-compose.yml` file:

yaml

version: '3'

services:

web:

build: .

ports:

- "8080:8080"

volumes:

- .:/app

Example command to run Docker Compose: docker-compose up

5. **Docker Volume**: Docker volumes are a way to persist data generated by and used by Docker containers. They allow data to be stored separately from the container's filesystem, making it easier to manage and share data between containers. Ex: command to create Docker volume: **docker volume create myvolume**

6. **Docker Network**: Docker networks allow containers to communicate with each other securely, either on the same host or across different hosts.

Example command to create a Docker network: **docker network create mynetwork**

7. **Docker Registry**: A Docker registry is a storage and distribution system for Docker images. It allows you to store and share Docker images either publicly or privately. Example command to push a Docker image to Docker Hub: **docker push username/image\_name**. Example command to pull a Docker image from Docker Hub: **docker pull username/image\_name**

8. **Docker Swarm**: Docker Swarm is Docker's native clustering and orchestration tool, which allows you to manage a cluster of Docker hosts and deploy services across them. Example: docker swarm init ,

To deploy a service on Docker Swarm: **docker service create --name myservice -p 8080:80 myimage**

9. **Docker Stack**: A Docker Stack is a collection of services deployed together as a single application. It is typically used in Docker Swarm mode for managing multi-container applications.

Example command to deploy a stack: docker stack deploy -c docker-compose.yml myapp

10. **Docker Exec**: Docker Exec allows you to run commands inside a running container.

Example command to execute a command inside a container: docker exec -it container\_name command

11. **Docker Logs**: Docker Logs allows you to view the logs generated by a running container.

Example command to view logs of a container: docker logs container\_name

12**. Docker Commit**: Docker Commit allows you to create a new image from changes made to a container.

Example command to commit changes to a container: **docker commit container\_id new\_image\_name**

13. **Docker Swarm Services**: Docker Swarm services define the tasks to be executed on the cluster. They allow you to scale applications across multiple Docker hosts. Example: **docker service ls**

Example command to scale a service in Docker Swarm: **docker service scale myservice=5**

14. **Docker Build:** Docker Build is used to build Docker images from a Dockerfile.

Example command to build a Docker image: docker build -t myimage .

15. **Docker Push/Pull for Private Registry**: Docker Push/Pull commands can be used to push & pull Docker images to/from a private registry. Ex: **docker push myregistry/myimage** , **docker pull myregistry/myimage**

16. **Docker Image Tagging**: Docker image tagging allows you to assign tags to images for versioning and identification purposes. Ex command to tag a Docker image: **docker tag myimage:latest myimage:v1.0**

17. Docker Image Cleanup: Docker Image Cleanup is used to remove unused images from the local Docker host to free up disk space. Example command to cleanup Docker images: **docker image prune**

18. Docker Network Inspection: Docker Network Inspection allows you to inspect Docker networks and view detailed information about them. Example command to inspect a Docker network: docker network inspect mynetwork.

20. **Docker Healthchecks**: Docker Healthchecks are used to monitor the status of a container and perform actions based on its health status. Example Dockerfile with a healthcheck: Dockerfile

FROM nginx

HEALTHCHECK --interval=5s --timeout=3s \

CMD curl -f http://localhost/ || exit 1

21. **Docker Overlay Networking**: Docker Overlay Networking is a method used to connect multiple Docker hosts together and enable communication between containers running on different hosts.

Example command to create an overlay network: **docker network create -d overlay my-overlay-network**

22. **Docker Secret Management:** Docker Secret Management allows you to securely store sensitive information such as passwords, API keys, and certificates, and use them in your Docker containers.

Example command to create a Docker secret**: echo "my\_secret" | docker secret create my\_secret** -

23. **Docker Volumes with Bind Mounts**: Docker Volumes with Bind Mounts allow you to mount a file or directory from the host machine into a container.

Example: **docker run -v /host/directory:/container/directory myimage**

24. **Docker Multi-stage Builds**: Docker Multi-stage Builds allow you to optimize Dockerfiles by using multiple build stages to reduce the size of the final image.

Example Dockerfile with multi-stage build: Dockerfile

FROM node:14 AS builder

WORKDIR /app

COPY . .

RUN npm install

RUN npm run build

FROM nginx

COPY --from=builder /app/build /usr/share/nginx/html

25. **Docker Compose Environment Variables**: Docker Compose allows you to define environment variables in a `.env` file and use them in your `docker-compose.yml` file.

Example `.env` file:

DB\_USER=myuser

DB\_PASSWORD=mypassword

Example `docker-compose.yml` file: yaml

version: '3'

services:

myapp:

image: myimage

environment:

- DB\_USER=${DB\_USER}

- DB\_PASSWORD=${DB\_PASSWORD}

26. **Docker Container Lifecycle Management**: Docker provides commands to manage the lifecycle of containers, including starting, stopping, pausing, and restarting containers.

To start a stopped container: docker start container\_name

To stop a running container: docker stop container\_name

To pause a running container: docker pause container\_name

To unpause a paused container: docker unpause container\_name

27. **Docker System Prune**: Docker System Prune is used to remove unused data, including stopped containers, unused networks, dangling images, and more, to reclaim disk space.

To perform system prune: **docker system prune**

To remove volumes as well during system prune: **docker system prune -a --volumes**

28. **Docker Events**: Docker Events command allows you to monitor Docker daemon events such as container creation, start, stop, and more. Ex: To stream Docker events: **docker events**

29. **Docker Security Scanning**: Docker Security Scanning allows you to scan Docker images for security vulnerabilities using Docker Hub or a third-party scanning service.

To scan a Docker image with Docker Hub: docker scan myimage

30. **Docker CPU and Memory Constraints**: Docker allows you to limit the CPU and memory resources available to a container.

To run a container with CPU and memory constraints: docker run --cpus=2 --memory=1g myimage

31. **Docker Remote Containers**: Docker Remote Containers allow you to work with Docker containers on a remote Docker host.

Example command to connect to a remote Docker host: docker -H tcp://remote-host:2375 ps

**Docker commands that are commonly used on a daily basis.**

1. docker run: Used to create and start a new container from an image. Ex: docker run -d --name mycontainer nginx

2. docker ps: Lists all running containers. Example: docker ps

3. docker stop: Stops a running container. Example: docker stop mycontainer

4. docker rm: Removes one or more containers. Example: docker rm mycontainer

5. docker images: Lists all locally available Docker images. Example: docker images

6. docker pull: Pulls an image from a registry. Example: docker pull ubuntu

7. docker push: Pushes an image to a registry. Example: docker push myusername/myimage

8. docker build: Builds a Docker image from a Dockerfile. Example: docker build -t myimage .

9. docker exec: Executes a command inside a running container. Example: docker exec -it mycontainer bash

10. docker logs: Displays logs from a container. Example: docker logs mycontainer

11. docker-compose up: Builds, (re)creates, starts, and attaches to containers for a service defined in a docker-compose.yml file.

Example: docker-compose up -d

12. docker-compose down: Stops and removes containers created by `docker-compose up`. Example: docker-compose down

13. docker network ls: Lists all Docker networks. Example: docker network ls

14. docker volume ls: Lists all Docker volumes. Example: docker volume ls

15. docker inspect mycontainer: Returns low-level information about Docker objects such as containers, images, volumes, and networks.

16. docker-compose build: Builds or rebuilds services defined in a docker-compose.yml file.

17. docker-compose restart: Restarts services defined in a docker-compose.yml file.

18. docker-compose logs myservice: Displays logs from services defined in a docker-compose.yml file.

19. docker-compose exec myservice bash: Executes a command inside a running container defined in a docker-compose.yml file.

20. docker-compose stop: Stops services defined in a docker-compose.yml file.

21. docker-compose down -v: Stops and removes containers, networks, volumes, and images created by `docker-compose up`.

22. docker image prune: Removes unused images.

23. docker system prune -a --volumes: Removes stopped containers, unused networks, dangling images, and unused volumes.

24. docker cp mycontainer:/path/to/container/file /local/path: Copies files or directories between a container and the local filesystem.

25. docker network create mynetwork: Creates a new Docker network.

**Kubernetes:**

Kubernetes is an open-source container orchestration platform used for automating deployment, scaling, and management of containerized applications. It allows you to manage a cluster of nodes, where each node may host multiple containers. Kubernetes provides features for deploying applications, scaling them up or down based on demand, rolling out updates seamlessly, and managing resources efficiently.

1. Pods: A pod is the smallest deployable unit in Kubernetes, representing one or more containers that share networking and storage resources. Pods are scheduled onto nodes in the cluster.

Example command to create a pod: kubectl create pod mypod --image=myimage

2. Deployments: Deployments manage the lifecycle of pods, allowing you to easily scale, update, and rollback application instances.

Example command to create a deployment: kubectl create deployment myapp --image=myimage

Scale up the deployment: kubectl scale deployment myapp --replicas=3

3. Services: Services provide networking for pods, allowing them to communicate with each other or external clients. There are different types of services such as ClusterIP, NodePort, and LoadBalancer.

Example command to create a service: kubectl expose deployment myapp --port=80 --target-port=8080 --type=NodePort

4. ConfigMaps and Secrets: ConfigMaps store configuration data as key-value pairs, while Secrets store sensitive information like passwords or API keys.

Example command to create a ConfigMap: kubectl create configmap myconfig --from-literal=key1=value1 --from-literal=key2=value2

Example command to create a Secret: kubectl create secret generic mysecret --from-literal=password=mypassword

5. Namespace: Namespaces provide a way to logically partition a Kubernetes cluster, allowing multiple users or teams to share the same cluster without interfering with each other.

Example command to create a namespace: kubectl create namespace mynamespace

6. Labels and Selectors: Labels are key-value pairs attached to objects in Kubernetes, which can be used for identifying and selecting objects.

Example command to label a pod: kubectl label pod mypod app=myapp

Example command to select pods with a specific label: kubectl get pods -l app=myapp

7. Volumes: Volumes provide persistent storage for containers in Kubernetes. They allow data to persist beyond the lifetime of a pod.

Example command to create a PersistentVolumeClaim (PVC): kubectl create pvc mypvc --storage=1Gi

Example command to mount a PVC to a pod: kubectl run mypod --image=myimage --restart=Never --attach --overrides='{ "spec": { "containers": [ { "name": "mypod", "image": "myimage", "volumeMounts": [ { "name": "myvolume", "mountPath": "/data" } ] } ], "volumes": [ { "name": "myvolume", "persistentVolumeClaim": { "claimName": "mypvc" } } ] } }'

8. StatefulSets: StatefulSets are used for managing stateful applications in Kubernetes, such as databases, where each pod has a unique identity and persistent storage.

Example command to create a StatefulSet: kubectl create statefulset mystatefulset --image=myimage --replicas=3

9. DaemonSets: DaemonSets ensure that a copy of a pod runs on each node in the cluster, useful for deploying system daemons or logging agents.

Example command to create a DaemonSet: kubectl create daemonset mydaemonset --image=myimage

10. Horizontal Pod Autoscaler (HPA): HPA automatically adjusts the number of pods in a deployment or replica set based on CPU or custom metrics.

Example command to create an HPA: kubectl autoscale deployment myapp --cpu-percent=50 --min=1 --max=10

11. Jobs and CronJobs: Jobs are used for running batch processes or one-off tasks, while CronJobs enable scheduling recurring tasks.

Example command to create a Job: kubectl create job myjob --image=myimage

Example command to create a CronJob: kubectl create cronjob mycronjob --schedule="\*/1 \* \* \* \*" --image=myimage

12. RBAC (Role-Based Access Control): RBAC allows you to define fine-grained access control policies for Kubernetes resources.

Example command to create a Role: kubectl create role myrole --verb=get,list,create --resource=pods

Example command to bind a Role to a User: kubectl create rolebinding myrolebinding --role=myrole --user=myuser

13. Ingress: Ingress manages external access to services within a Kubernetes cluster, typically handling HTTP/HTTPS traffic.

Example command to create an Ingress: kubectl create ingress myingress --rule=myrules

Example of myingress:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: myingress

spec:

rules:

- host: myapp.example.com

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: myapp-service

port:

number: 80

**This is my Self Introduction**.

I'm K Venkata Narasimha Rao. I have **12 years of IT experience** including **5 years of experience** **on cloud and DevOps** engineering. When I say cloud. I have an experience with AWS and DevOps.

I **started my career as a GIS Engineer**, worked for 4 years, where I specialized in geospatial technologies. Subsequently, I transitioned to a **support engineer** role for 1 year and provided technical assistance for various troubleshooting issues.

Driven by a passion about the technology, I transitioned into a **Linux Administrator**, where I spent nearly 2 years by ensuring the **stability and security of Linux servers**. During that time, I was discovered my interest in **cloud technologies & DevOps practices**, and I was moved as cloud and DevOps engineer.

From last 5 years, I'm working DevOps and AWS Cloud operations, implemented CI-CD pipelines.

My roles and responsibility are to build CI/CD pipelines, cloud infrastructure build, deployments.

I also prepare the Terraform templets as Infrastructure as Code (IAC) to Provisioning the Infrastructure and Create Ansible Playbooks for Configuration Management.

And work with the git Version control tool, build tools like maven and Integration tool Jenkins. And Containerization using **Docker**, **Kubernetes** for Orchestration.

In AWS Cloud, i have worked on various services, like EC2 VM’s, Storage, Monitoring, VPC Networking and ,Traffic manager, load balancer and also there is a requirement to build and deployment i try to work closely with developers and set up CI pipelines and also responsible for creating CD deployments groups and based on the sprint releases we will create release pipelines and release branches and we create on given release date we create in different environments(VM, Deployment, webapps, Containerizations etc.).

**DAY 2 DAY ACTIVITIES**

* Creating CICD Pipelines, Infrastructure provisioning for new projects using IaC. Configuration Management using Ansible, Branching Strategies, Application Deployment using Docker & K8s.
* Any tickets which come to my service now bin and request for new Infrastructure server building and new software deployment.
* If we face any server issues like connecting issues or ping issues, I will work on that to resolve and close tickets within the time frame based on SLA
* we will try to resolve, basically in our projects. my day start with taking the shift hand over from previous shift members,
* understanding pending tickets and working on them, if any build request comes responding to them, in general we get service request and we try to complete.
* them and submit to client and we attend everyday daily team calls with our team and also with client meetings based on requirement.

what is the last project or latest activity you did using Terraform?

what is the use of VPC?

How can you create a VPC using Terraform and what are steps to be followed?

you have created some resources using terraform, how can you delete them

I want to execute some scripts or commands in Terraform resources which we created? How can you do that?

what is Terraform Provisioner?

Some AWS resources & Servers are configured in AWS DynamoDB and some users and Clients wants to connect or access them. How can you resolve and provide access?

AWS EC2 instance is attached to EBS volume, and it not encrypted, how can you resolve it what is the process?

How can you create EC2 instance using Terraform?

If someone come to you for access the AWS resources. How can you resolve the issue.

How can you create CloudFormation using Terraform?

instance is running state and how can you change the configuration t2 to t3 like?

Let us say, one server should run 24hours, and other servers should run depend on client request and load. How can you achieve this & resolve?

How many types of Auto scaling groups in AWS?

Do you know python end to end programming to deploy the application?