Section 1: Q&A (20 Questions)

- **Terraform Fundamentals :**

1. **What is Terraform and how does it differ from other IaC tools?**
   Infrastructure as Code (IaC) is a method to provision and manage IT infrastructure (servers, networks, databases, etc.) using machine-readable configuration files, instead of physical hardware configuration or interactive configuration tools.
   Terraform is an open-source IaC tool created by HashiCorp that allows users to define and manage infrastructure across various cloud providers and services using a high-level configuration language (HCL - HashiCorp Configuration Language).
   Terraform takes a declarative approach to IaC, meaning users specify the desired state of the infrastructure, and Terraform automatically determines how to achieve that state.
   It differs from other applications such as Ansible, Puppet, Chef, and AWS CloudFormation since it supports many clouds, manages states, and focuses on infrastructure provisioning. Terraform provides a more complete and scalable cloud automation solution by using a state file for simple planning and change application.
2. **Explain Terraform's declarative nature and state management.**
   Terraform takes a declarative approach to IaC, meaning users specify the desired state of the infrastructure, and Terraform automatically determines how to achieve that state, without you worrying about the step-by-step instructions.
   To keep track of what's already been built, Terraform uses a state file. This helps it understand what exists, what has changed, and what needs to be updated. That way, it won't accidentally recreate resources you already have.
3. **What is the purpose of the Terraform provider?**
   A Terraform provider makes it possible for Terraform to communicate with different platforms, like AWS, Azure, Google Cloud, and others. For every service, it understands how to add, modify, and remove resources. Terraform wouldn't know how to handle infrastructure without providers. For this reason, each project begins by identifying what providers it needs.
4. **How does Terraform handle dependency resolution?**
   According to their dependencies, Terraform automatically determines which resources should be created, updated, or removed first. It accomplishes this by looking at the configuration's resource references. For instance, Terraform makes sure the VPC is generated first if an EC2 instance requires one. It completes tasks in the correct order and creates a dependency tree. When necessary, you may also use "depends on" to create explicit dependencies.
   Errors are avoided and seamless infrastructure upgrades are guaranteed by this management.

5. **What are the key components of a Terraform configuration file?**
   Several essential elements make up a Terraform configuration file.
   -Which cloud or service provider, such as AWS or Azure, you will utilize is determined by the **providers**.
   -**Resources** define the components of the infrastructure you wish to build, like databases or EC2 instances.
   -**Variables** make the setup reusable by enabling you to set versatile input values.
   -Important information, like as instance IPs, are displayed in the **outputs** once Terraform has applied the modifications.
   -**Modules** combine relevant resources so they can be used again in other applications.
   Together, these elements assist you in effectively defining and overseeing your infrastructure.

- **State Management & Backend Configuration :**
  1. **Explain the difference between terraform refresh, terraform plan, and terraform apply.**
     The terraform **refresh** command doesn't change your infrastructure, but it updates your local state file to match any changes made outside of Terraform.

     When you run terraform **plan**, you get a preview of what Terraform will do if you apply any updates. It's like a dry run that lets you see the potential changes before making them.

     Then, when you're ready, terraform **apply** takes that plan and actually makes the changes, updating your infrastructure to match the desired state.

  2. **What is the difference between local and remote backends?**
     The main difference between Terraform's local and remote backends is where the state file is stored.
     A **local backend** saves the state file on your computer, making it a good choice for small projects or personal use. However, since others can't easily access it, collaborating with a team can be tricky and even risky.
     A **remote backend**, on the other hand, stores the state file in a shared location like an S3 bucket or Terraform Cloud. This makes teamwork easier by allowing centralized state management, versioning, and locking—helping to prevent conflicts and keep everything organized.
  3. **How can you prevent state corruption when multiple engineers work on the same infrastructure?**
     Remote backends can include state locking, which helps prevent multiple people from making changes at the same time. Services like Terraform Cloud or S3 with DynamoDB ensure that only one engineer can modify the state at a time, reducing the risk of corruption from overlapping updates.

To keep things even more organized, you can use workspaces and version control. This makes it easier to manage multiple environments and avoid conflicts in your infrastructure setup.

- **Terraform Modules & Reusability :**
  1. **What are the benefits of using Terraform modules?**
     Terraform modules help keep your infrastructure organized and easy to manage. Instead of writing the same code over and over for similar resources, you define it once in a module and reuse it whenever needed. This keeps your setup clean and efficient.
     Modules also help structure resources into logical groups, making everything easier to maintain. Plus, since you can version them, you have better control over changes and can quickly roll back if something goes wrong.
  2. **Explain how to pass variables to a Terraform module.**
     In Terraform, you use the variable block to define variables inside a module. When you call the module, you pass values for those variables in the module block. This makes it easy to customize the module's behavior based on your needs. For even more flexibility, you can use environment variables or variable files to control the values passed to the module.
  3. **What is the difference between count and for_each?**
     In Terraform, **count** and **for_each** both help create multiple resources, but they work in different ways.
     Use **count** when you need a specific number of identical resources. It's great when all the resources are the same, and you know exactly how many you need.
     for_each is more flexible because it creates resources based on a collection, like a list or a map. This is useful when the resources are similar but not exactly the same, allowing each one to have unique values.
  4. **How do you source a module from a Git repository?**
     To use a Terraform module from a Git repository, you set the repository URL in the source parameter of the module block. Terraform will fetch the module directly from Git, and you can specify a particular branch, tag, or commit by adding a reference in the URL. This makes it easy to use modules stored remotely in your configuration.

- **Terraform with AWS :**
  1. **How do you create an EC2 instance with Terraform?**
     To create an EC2 instance with Terraform, you start by defining the **AWS provider** and a **resource block** for the instance. This configuration includes essential details like the **instance type** and **AMI ID**, as well as additional settings such as **security groups**, **key pairs**, and **networking options** to control access and connectivity.
     When you run terraform apply, Terraform provisions the EC2 instance based on your configuration. It ensures the instance is created, updated, or deleted as needed to match the desired state. By managing the infrastructure declaratively, Terraform simplifies provisioning while maintaining consistency across deployments.

2. **What are the required fields for defining a VPC in Terraform?**
   When defining a VPC in Terraform, some key settings help shape its structure. The cidr_block parameter sets the VPC's IP address range, while enable_dns_support and enable_dns_hostnames control whether DNS features are available within the VPC.
   Although these are essential for most setups, you can also include optional parameters like **tags** to label and organize your VPC. These settings ensure the VPC is properly configured for your network needs.
3. **Explain how Terraform manages IAM policies in AWS.**
   With Terraform, you can manage IAM policies in AWS by defining them directly in your configuration or linking to an external JSON file. When you apply your setup, Terraform automatically creates or updates the policies and assigns them to the right users, roles, or groups.
   This approach keeps your access permissions organized, consistent, and version-controlled, making it easier to manage security across your environment.

4. **How do you use Terraform to provision and attach an Elastic Load Balancer?**

   To set up an Elastic Load Balancer (ELB) in Terraform, you need to define key resources in your configuration. Typically, this includes:
   - **aws_lb –** Creates the load balancer itself.
   - **aws_lb_target_group–** Defines the group of instances or services that will receive traffic.
   - **aws_lb_listener –** Configures how incoming traffic is routed to the target group.
   Once everything is set up, running `terraform apply` provisions the ELB and automatically establishes the necessary connections between components. This ensures traffic is distributed efficiently across your infrastructure.

- **Debugging & Error Handling (4 questions)**

1. **What does the terraform validate command do?**
   The terraform validate command checks your Terraform configuration files for errors. It doesn't create or modify any resources—it simply ensures that your code is well-structured and follows Terraform's rules. Running this command before planning or applying changes is a quick way to catch mistakes and keep your configuration error-free.
2. **How can you debug Terraform errors effectively?**
   To troubleshoot Terraform issues, you can enable detailed logging by setting **TF_LOG=DEBUG**. This provides in-depth information about what Terraform is doing.
   Before making changes, use **terraform plan** to review updates and catch potential problems early. If you run into errors, check the messages carefully, consult the Terraform documentation, and review the state file for inconsistencies.
   Breaking your configuration into smaller parts and applying changes gradually can also help identify and resolve issues more effectively.

3. **What is Terraform's ignore_changes lifecycle policy used for?**
   The **ignore_changes** lifecycle policy in Terraform prevents specific resource properties from being updated or changed. When applied, it tells Terraform to ignore any differences in those properties, even if they don't match the configuration.
   This is useful when you want to allow manual or external updates to a resource without Terraform overwriting them the next time you apply changes.
4. **How do you import existing AWS infrastructure into Terraform?**
   You must first declare the resource in your Terraform configuration without applying it before you may import current AWS infrastructure into Terraform. The existing resource can then be linked to the configuration by using the Terraform import command along with the resource type and AWS identification. Run Terraform Plan after importing to examine any discrepancies and make any necessary configuration updates to reflect the resource's current state.

Section 2: Hands-on Practical Assignments

Task 5: Debugging and Fixing Terraform Issues

- Given a misconfigured Terraform file, identify and fix the following errors:
  - Invalid provider configuration
  - Incorrect security group rules
  - Missing IAM permissions for Terraform to create resources

answer:

Solving Terraform Problems

To diagnose Terraform issues more effectively, we enable debugging and capture logs:

Enable detailed logging by running:

```
export TF_LOG=DEBUG
```

This helps identify errors by providing more in-depth logs.

Save logs to a file for further analysis:

```
export TF_LOG_PATH=./terraform.log
```

This creates a terraform.log file where you can review error details.

Reinitialize Terraform to ensure a clean state:

```
terraform init -reconfigure
```

This refreshes the setup and can resolve configuration-related issues.

Verifying Security Groups

To ensure that the security groups controlling access to EC2 and ALB match your Terraform configuration,we follow these steps:

Check the current state of the security groups:

```
terraform state show aws_security_group.alb_sg
terraform state show aws_security_group.ec2_sg
```

Run targeted plans to preview changes:

```
terraform plan -target=aws_security_group.alb_sg
terraform plan -target=aws_security_group.ec2_sg
```

These commands help confirm that security group rules are correctly configured before applying any updates.

Resolving IAM Permission Issues

If we encounter AccessDenied errors, Terraform might not have the required permissions. To troubleshoot we:

Enable debugging to get detailed logs:

```
TF_LOG=DEBUG terraform apply
```

If the issue persists, attach the necessary IAM policies:

```
aws iam attach-user-policy --user-name TERRAFORM_USER --policy-arn
arn:aws:iam::aws:policy/AmazonEC2FullAccess


aws iam attach-user-policy --user-name TERRAFORM_USER --policy-arn
arn:aws:iam::aws:policy/ElasticLoadBalancingFullAccess
```

These policies grant Terraform access to manage EC2 instances and load balancers, resolving permission-related errors.