**INSTRUCTIONS:**

1. Answer all questions from Section 1, 2 & 3
2. Section 4 is a programming exercise. Prepare complete Terraform IaC with [best practices](https://cloud.google.com/docs/terraform/best-practices-for-terraform) and share your project using GitHub.

**SECTION 1: GIT**

1. **if you using git stash, where will it save data? What is diff b/w index and staging area?**

When you use the git stash command, it saves your current changes in a temporary storage area so that you can switch to a different branch or perform other operations without committing your changes. The data saved by git stash is stored in a stack-like structure, and you can have multiple stashes. By default, these stashes are saved in the .git directory in a special reference named refs/stash. Each stash entry is identified by a unique name (e.g., stash@{0}, stash@{1}), and you can apply or pop them back into your working directory as needed.

The terms "index" and "staging area" are often used interchangeably in the context of Git, but they have slightly different nuances:

Index: The index is sometimes referred to as the "staging area." It is an intermediate area in Git that sits between your working directory and the repository. When you make changes to your files, the changes are initially in your working directory. You use git add to move these changes to the index. The index acts as a snapshot of the changes you want to include in the next commit. In other words, it's a place where you prepare and stage changes before committing them. Once you've added your changes to the index, you can commit them with git commit.

Staging Area: The staging area is essentially the same as the index. It's where you prepare and stage changes before committing. When you run git add, you're placing your changes in the staging area, making them ready for the next commit. The term "staging area" emphasizes the idea that you're preparing changes for a performance (commit), much like actors getting ready to go on stage in a play. Therefore, "staging area" is another name for the index.

In summary, the index and staging area are two terms used to describe the same concept in Git, which is the intermediate area where you prepare and organize changes before committing them to the repository.

1. **when would individuals use git rebase, git fast-forward, or a git fetch then push?**

When you use the git stash command it stores your changes in a temporary storage area. This allows you to switch to a branch or perform tasks without committing your changes right away. The data saved by git stash is stored in a reference named refs/stash within the.git directory. It uses a stack structure. Allows you to have multiple stashes. Each stash entry has a name, like stash@{0} or stash@{1}. You can apply or pop them back, into your working directory whenever needed.

1. **How to revert already pushed changes ?**

Reverting already pushed changes in Git is a common operation when you want to undo or remove specific commits from the repository's history. There are a few different ways to achieve this, depending on the situation and your project's needs. Here are two common methods for reverting pushed changes:

**Using git revert:**

The git revert command is a safe way to undo individual commits by creating a new commit that undoes the changes made in the target commit. This method is preferred when you want to keep a record of the fact that a change was undone.

To revert a specific commit, follow these steps:

git checkout <branch-name>

git log # Note the commit hash of the commit you want to revert

git revert <commit-hash>

git push origin <branch-name>

Here, <branch-name> is the name of the branch where you want to undo the commit, and <commit-hash> is the hash of the commit you wish to revert.

Using git reset and git push

The git reset command can be used to move the branch pointer to a previous commit, effectively "unpublishing" commits from the remote repository. However, this method rewrites history, which can cause issues if others are working on the same branch.

If you are absolutely sure that no one else is working on the branch or you have coordinated with your team, you can use the following approach:

git checkout <branch-name>

git log # Note the commit hash you want to reset to

git reset --hard <commit-hash>

git push -f origin <branch-name> # Force-push the updated branch

Be cautious when using git reset --hard and force-pushing (-f option) because it can disrupt the work of other team members if they have pulled or cloned the branch after the original commits were pushed.

1. **What is the difference between cherry picking commits vs trying a hard reset. What is the final outcome of the head reference?**

**Purpose: Cherry-picking** is used to select specific commits from one branch and apply them to another branch. It's useful when you want to bring in specific changes without merging the entire branch.

Effect on Commit History: Cherry-picking creates new commits that mirror the changes of the selected commits from another branch. These new commits are applied on top of the current branch, preserving your existing commit history.

Outcome of HEAD: After cherry-picking, the HEAD reference will point to the latest commit on your branch, which includes the cherry-picked changes.

**Purpose: A hard reset** is used to move the branch pointer to a specific commit, effectively discarding all the commits that came after that commit. It's often used to "rewind" a branch to a previous state.

Effect on Commit History: A hard reset removes the commits from the branch and effectively "erases" them from the commit history. These discarded commits are not part of the branch's history anymore.

Outcome of HEAD: After a hard reset, the HEAD reference points to the specified commit, effectively setting the branch back to that commit.

1. **Explain the difference between git remote and git clone?**

**git remote:**

Purpose: The git remote command is used to manage remote repositories that are connected to your local Git repository. It provides information about remote repositories and allows you to add, remove, or rename them.

**Usage**: You can use git remote to list the remote repositories associated with your local repository, view details about a specific remote, add a new remote, remove an existing remote, or rename a remote.

Examples:

git remote -v: Lists the remote repositories along with their URLs.

**git clone:**

Purpose: git clone is used to create a copy of a remote repository on your local machine. It clones the entire history and files of the remote repository, setting up a local Git repository for you to work with.

Usage:

git clone <repository-url>: Clones a remote repository to your local machine

**Usage**:

git clone creates a new directory with the contents of the remote repository, initializes a Git repository within it, and checks out the default branch (usually master) so you can start working with the code locally.

**Note**: **git remote** is used for managing connections to remote repositories and remote branches, while **git clone** is used to create a local copy of a remote repository on your machine. The two commands are complementary in a Git workflow: you use git remote to configure and manage your remote connections, and git clone to set up a local working copy of a remote repository.

**SECTION 2: TERRAFORM**

1. **what is the difference between terraform count and for\_each meta data function? and give a scenario-based example to use them?**

**count:**

Use Case: The count argument is used when you want to create a fixed number of resource instances based on a numerical value or expression. It's typically used for scenarios where you know in advance how many instances of a resource you need.

**Example Scenario**: Let's say you want to create a certain number of virtual machines in a cloud environment, and you know exactly how many you need. You can use count to specify the number of virtual machines to create.

**resource "aws\_instance" "example" {**

**count = 5 # Create 5 EC2 instances**

**ami = "ami-0c55b159cbfafe1f0"**

**instance\_type = "t2.micro"**

**}**

**for\_each:**

**Use Case**: The for\_each meta-argument is used when you want to create resource instances based on a map or set of key-value pairs, where the keys represent unique identifiers for the instances. This is useful when the number of instances and their attributes can vary and you want more control over resource management.

Example Scenario: Suppose you need to create security groups for different teams in your organization, and each security group has different rules. You can use for\_each to create security groups and define their rules in a map.

variable "security\_groups" {

type = map(object({

name = string

description = string

ingress = list(object({

from\_port = number

to\_port = number

protocol = string

cidr\_blocks = list(string)

}))

}))

}

resource "aws\_security\_group" "example" {

for\_each = var.security\_groups

name\_prefix = each.value.name

description = each.value.description

ingress = each.value.ingress

}

1. **What is Terraform taint ? When to use it? When would you use terraform state rm vs terraform taint?**

**Use Cases for terraform taint**: You might use terraform taint in scenarios where you want to force the recreation of a specific resource, such as when you suspect that a resource has become corrupted or when you want to apply changes to that resource that are not achievable through a regular configuration update.

**erraform State Remove (terraform state rm):**

Purpose: The terraform state rm command is used to remove a resource from the Terraform state entirely. This effectively means Terraform forgets about the resource, and it will no longer be managed or tracked.

Use Cases for terraform state rm: You would use terraform state rm when you want to completely get rid of a resource from your state, typically because it no longer exists or is managed by Terraform, and you no longer wish to track it.

1. **How would you show a diagram of all terraform resources in the state file? When is this useful?**

Generate a diagram of all Terraform resources in your state file using a tool called "Terraform Graph." This tool creates a graphical representation of your infrastructure, showing the relationships and dependencies between resources.

**command to generate the Terraform graph:**

**terraform graph | dot -Tpng > graph.png**

**useful in several scenarios:**

**Documentation:** It provides a visual representation of your infrastructure, making it easier to understand the relationships between resources and their dependencies. This can be valuable for documentation purposes, especially when sharing information with team members or stakeholders.

**Troubleshooting:** When you encounter issues or unexpected behavior in your infrastructure, the graph can help you identify resource dependencies and visualize the potential sources of the problem.

**Infrastructure Review:** Before applying changes to your infrastructure, you can use the graph to review the existing configuration and understand how modifications might impact the entire infrastructure.

1. **Solve this expression:**

count                  = var.run\_remote\_environment ? var.TFC\_RUN\_ID !=["Yes"]) : null

count = var.run\_remote\_environment && var.TFC\_RUN\_ID != "Yes" ? 1 : 0

1. How would you apply terraform to multiple accounts simultaneously? We want to ensure this

follows security best practices.

**SECTION 3: AWS**

1. **You have an EC2 instance that has an unencrypted volume. You want to create another Encrypted volume from this unencrypted volume. Which of the following steps can achieve this?** How would you share this encrypted volume to another account? What must you ensure to make sure this cross-account encryption is shared?

First, create a snapshot of the unencrypted volume that you want to encrypt. This snapshot serves as the source for the new encrypted volume.

After creating the snapshot, you can create a new encrypted EBS volume from it. During this step, you can specify that the new volume should be encrypted.

Once the new encrypted volume is created, you can attach it to your EC2 instance, much like you would with any other volume.

After attaching the encrypted volume to your EC2 instance, you'll need to format it with a file system (e.g., ext4, NTFS) and then mount it to a directory on your instance.

Here's a more detailed breakdown of the steps:

Create a Snapshot of the Unencrypted Volume:

Go to the AWS Management Console.

Navigate to the "Snapshots" section under "Elastic Block Store" in the EC2 Dashboard.

Select "Create Snapshot" and choose the unencrypted volume as the source.

Review and confirm the snapshot creation.

Create an Encrypted Volume from the Snapshot:

Still in the AWS Management Console, go to the "Snapshots" section.

Select the snapshot you just created.

Choose "Actions" and then "Create Volume."

Specify the size, availability zone, and select the option to encrypt the volume. You can choose a KMS (Key Management Service) key for encryption or use the default key.

Click "Create."

Attach the Encrypted Volume to Your EC2 Instance:

Go to the "Volumes" section under "Elastic Block Store" in the EC2 Dashboard.

Select the new encrypted volume.

Choose "Actions" and then "Attach Volume."

Select your EC2 instance and specify the device name (e.g., /dev/xvdf).

Format and Mount the Encrypted Volume:

SSH into your EC2 instance.

Format the newly attached encrypted volume with a file system (e.g., mkfs.ext4, mkfs.ntfs).

Create a directory where you want to mount the volume (e.g., /mnt/mydata).

Mount the encrypted volume to the directory.

Update your /etc/fstab to ensure the volume is mounted automatically on instance startup.

1. **How will you implement service control policy and in which area are you using it?**

Service Control Policies (SCPs) are a feature provided by AWS Identity and Access Management (IAM) that allow you to set fine-grained permissions and access controls over your AWS Organizations. SCPs are used to control access and permissions at the root level of your organization or within organizational units (OUs). You can use SCPs to manage permissions for member accounts and enforce security and compliance policies across your AWS organization.

**Implementation Steps:**

Access the AWS Organizations Console:

Sign in to your AWS Management Console.

Open the AWS Organizations console.

Create a Service Control Policy (SCP):

In the Organizations console, select "Service control policies" from the left navigation pane.

Choose "Create policy."

Define the SCP by specifying allowed or denied actions for specific AWS services or actions.

Attach the SCP:

After creating the SCP, you can attach it to the root of your organization or to specific OUs.

Attaching an SCP at the root level affects all accounts in the organization, while attaching it to an OU applies only to accounts within that OU.

Review and Apply:

Review the SCP to ensure it meets your organization's security and compliance requirements.

Attach the SCP to the desired organizational unit or root.

Use Cases:

Service Control Policies can be used in various areas to enforce security, compliance, and access control in AWS Organizations:

Compliance and Security: SCPs can be used to enforce compliance with security standards, industry regulations, and internal security policies. For example, you can use SCPs to restrict access to certain AWS services or actions that may pose security risks.

Resource Separation: SCPs can help separate resources and access controls in multi-account architectures. For example, you can use SCPs to limit the use of certain AWS services to specific accounts or OUs.

Isolation of Workloads: If you're using AWS accounts to isolate workloads, SCPs can be used to control the interaction between accounts and ensure that they don't inadvertently access or modify resources in other accounts.

Enforcing Budgets: You can use SCPs to control resource provisioning and spending by limiting the use of costly services and actions.

Enforcing Least Privilege: SCPs allow you to apply the principle of least privilege by limiting the permissions of AWS accounts to only what they need to perform their intended functions.

Centralized Governance: SCPs enable centralized governance and policy enforcement across your AWS organization. You can create and manage policies from a central location.

1. **How can you convert a public subnet to private subnet?**

Converting a public subnet to a private subnet in the (AWS) typically involves modifying the subnet's routing configuration and network access control. A public subnet is one with a route table that directs traffic to the internet via a Network Address Translation (NAT) gateway or instance, while a private subnet typically routes traffic to internal resources without direct internet access. Here are the steps to convert a public subnet to a private subnet:

Create a New Route Table for the Private Subnet:

In the AWS Management Console, navigate to the VPC Dashboard.

Select "Route Tables" from the left navigation pane.

Create a new custom route table for your private subnet.

Modify the Route Table Associations:

Associate the newly created route table with the private subnet(s) that you want to convert.

Disassociate the public subnet from its current route table.

Update the Routing Configuration:

In the route table associated with the private subnet, add the following route:

Destination: 0.0.0.0/0

Target: The ID of a security appliance or other resource that can forward traffic to the internet if necessary.

Note that this route doesn't necessarily route traffic to the internet directly but rather to a component that can do so.

Network ACLs and Security Groups:

Adjust network ACLs and security groups as needed to control inbound and outbound traffic to and from the private subnet.

Make sure that your network ACLs and security group rules allow necessary communication between resources in the private subnet and other parts of your VPC or with external services.

(Optional) Replace NAT Gateway/Instance:

If your public subnet relies on a NAT gateway or instance, you can either terminate or disassociate it from the public subnet to avoid unnecessary costs.

Update Resource Configurations:

Update the configurations of resources in the private subnet (e.g., EC2 instances, RDS databases) to use the new route table for communication. Make sure they can reach any required resources within or outside your VPC.

Test Connectivity:

After making the changes, test connectivity from resources in the private subnet to ensure that they can communicate as expected.

1. **What is the default route for any newly created route table?**

For any newly created custom route table in Amazon Virtual Private Cloud (VPC) on AWS, the default route entry is an implicit local route. This local route allows for communication between resources within the same VPC.

The local route has the following characteristics:

Destination: The destination for this route is the VPC's CIDR block, which encompasses all IP addresses in the VPC.

Target: The target for this route is always "local." This means that traffic to any IP address within the VPC, including resources in different subnets, is routed directly within the VPC.

The local route is automatically created for every custom route table, and you cannot delete it or modify its destination or target. It ensures that resources within the VPC can communicate with each other, even if you don't explicitly define any other custom routes in the route table.

In addition to the local route, you can add custom routes to direct traffic to specific destinations through different targets, such as a network appliance or a Virtual Private Gateway for connecting to on-premises networks or other VPCs. These custom routes allow you to control the routing of traffic in your VPC to meet your network architecture and connectivity requirements.

1. **How would you ensure routes in the route table DO NOT use the local routes?**

In VPC on AWS, you cannot remove or modify the local route (the route to the VPC's CIDR block) in a custom route table. This route is automatically created and cannot be deleted or altered.

However, if you want to ensure that routes in the route table do not use the local route or want to prevent specific resources or subnets from using the local route for communication, you have a few options:

Create a New Custom Route Table: Instead of modifying the default route table, create a new custom route table for your VPC. You can then associate specific subnets or resources with this custom route table, which allows you to have more control over the routes without the implicit local route.

Control Route Associations: When you create a new custom route table, you can associate specific subnets with it. These subnets will use the routes defined in the custom route table. Subnets associated with the default route table continue to use the implicit local route.

Control Route Priorities: You can define custom routes with different priorities in your custom route table. Custom routes with more specific destination CIDR blocks will take precedence over the local route. In other words, if you have a custom route for a more specific CIDR block, it will be used instead of the local route.

For example, if you have a custom route for a specific subnet, and that route specifies a more specific destination (e.g., a smaller CIDR block) than the local route, the traffic to that specific destination will follow the custom route. However, if there is no custom route defined for a destination, the traffic will use the local route.