Q1: What is Terraform?

Answer:
Terraform is an open-source Infrastructure as Code (IaC) tool developed by HashiCorp. It allows you to define, provision, and manage infrastructure across various cloud providers like Microsoft Azure, AWS, and Google Cloud using a declarative language called HCL (HashiCorp Configuration Language).

Instead of manually setting up cloud resources through the portal or CLI, with Terraform, we can write infrastructure configurations in code. This makes infrastructure repeatable, version-controlled, and auditable — just like application code.

For example, in Azure, we can use Terraform to provision virtual machines, networking, storage accounts, and more — all through `.tf` files. Once the code is ready, we use `terraform plan` to preview the changes and `terraform apply` to provision the resources.

It's especially useful in teams and DevOps environments where automation, repeatability, and scalability are key. It also integrates well with CI/CD tools like Azure DevOps or GitHub Actions for continuous provisioning.

Q2: What does `terraform init` do?

Answer:
`terraform init` is the **first command** you run when working with a new or existing Terraform configuration.

It does the following:

1. **Initializes the working directory** – It sets up the local directory to use Terraform. It looks for `.tf` files and prepares them.
2. **Downloads provider plugins** – Based on the provider block in your configuration (e.g., Azure, AWS, Google Cloud), it downloads the required provider plugin versions from the Terraform Registry.
3. **Sets up the backend** – If you're using a remote backend (like Azure Storage or S3 for state files), `terraform init` configures and connects to that backend.
4. **Installs modules** – If you're using modules (local or remote), it downloads and prepares them.

In short, `terraform init` prepares everything needed to start using Terraform safely in your project directory.

👉 Without running `terraform init`, you **cannot** proceed with commands like `terraform plan` or `terraform apply`.

Q3: What is the Terraform state file and what is its role?

Answer:

The **Terraform state file** (usually named `terraform.tfstate`) is a **critical file** that stores the current state of your infrastructure as known to Terraform.

🔍 What it does:

1. **Maps your configuration to real resources**
   It keeps track of what resources Terraform has created and what their current values are (e.g., VM names, IP addresses, disk sizes, etc.).
2. **Enables change tracking**
   When you run `terraform plan`, Terraform compares the desired state (from your `.tf` code) with the **actual state** (from the `.tfstate` file) to determine what needs to change.
3. **Speeds up operations**
   Terraform doesn't query the cloud provider for everything on every run—it uses the state file as a cached source of truth.

📂 Where is it stored?

- **By default**: Stored **locally** in the same folder (`terraform.tfstate`)
- **Best practice**: Store it **remotely** using backends (e.g., Azure Blob Storage, AWS S3) for:
  - Team collaboration
  - Locking to prevent conflicts
  - Backup and versioning

🔒 Why it’s important:

- If the state file is lost or corrupted, Terraform **won’t know what exists**, and you risk resource duplication or deletion.
- It can contain **sensitive data** like passwords, connection strings, etc., so it must be protected.

Summary:

The Terraform state file is like a **database of your infrastructure**. It helps Terraform understand the difference between what you **want** (your code) and what you **have** (real resources), so it can apply changes accurately and safely.

Q3: What are Terraform `lifecycle` blocks and what do they do?

Answer:

In Terraform, a `lifecycle` block is used **inside a resource block** to **customize the behavior** of how Terraform handles that resource during its lifecycle — creation, updates, and deletion.

It allows you to override the default behavior of Terraform.

☐ Basic Syntax:

```
hcl
CopyEdit
resource "azurerm_storage_account" "example" {
  name                     = "mystorageacct"
  resource_group_name      = "my-rg"
  location                 = "East US"
  account_tier             = "Standard"
  account_replication_type = "LRS"

  lifecycle {
    create_before_destroy = true
    prevent_destroy       = true
    ignore_changes        = [tags]
  }
}
```

🛠 Types of Lifecycle Arguments:

1. **`create_before_destroy`**
   - Forces Terraform to create the new resource **before destroying** the old one.
   - Useful for resources where downtime is unacceptable (e.g., load balancers, VMs).
   - Default behavior is *destroy first, then create*.
2. **`prevent_destroy`**
   - Protects a resource from being accidentally deleted.
   - If someone runs `terraform destroy`, it will **error out** instead of destroying that resource.
   - Good for production databases or storage accounts.
3. **`ignore_changes`**
   - Tells Terraform to **ignore changes** to specified attributes even if they drift from the code.
   - Useful for fields like `tags`, where other teams/tools might make changes outside Terraform.

📌 Real-Life Use Case:

- **`prevent_destroy`** for a production database to avoid accidental wipe.
- **`ignore_changes`** on VM size in dev environment where sizing may be done manually.
- **`create_before_destroy`** for switching from one VM image to another with zero downtime.

Summary:

The `lifecycle` block in Terraform gives you **fine control** over how resources are handled during updates and deletes — helping avoid downtime, protect critical infrastructure, and allow flexibility for external changes.

Q4: What is `terraform taint` and when is it used?

Answer:

`terraform taint` is a command that **manually marks a resource for destruction** and **forces it to be recreated** during the next `terraform apply`.

Even if there is no change in the configuration, Terraform will treat the tainted resource as needing to be replaced.

🛠 Why use `terraform taint`?

- When a resource is **unhealthy or misbehaving** (e.g., corrupted VM, bad config).
- When a resource was **changed outside of Terraform** and you want to reset it.
- For **testing**, if you want to simulate a change or replacement.

📌 Example Command:

```
bash
CopyEdit
terraform taint azurerm_virtual_machine.example
```

This marks the VM named `example` for recreation.

Then:

```
bash
```

```
CopyEdit
terraform apply
```

Terraform will destroy and re-create that VM — even if the `.tf` config hasn't changed.

🔄 What happens behind the scenes?

- It updates the **state file**, not the actual resource.
- Marks that resource as tainted (needs replacement).
- During `apply`, Terraform will destroy the tainted resource and recreate it.

⚠️ Note:

- In Terraform v0.15+, the `terraform taint` command is being **deprecated** in favor of:

  ```
  hcl
  CopyEdit
  lifecycle {
    replace_triggered_by = [something]
  }
  ```

  Or use the newer method:

  ```
  bash
  CopyEdit
  terraform apply -replace=azurerm_virtual_machine.example
  ```

✅ Summary:

`terraform taint` is used to **force Terraform to recreate** a specific resource by marking it as "tainted", which is helpful for fixing broken resources without changing the code.

Let me know if you want to see how it works with Azure resources.

You said:

mean now we are not using terraform taint for current versions , as we have lifecycle blocks

ChatGPT said:

✅ Exactly — you're spot on!

Let’s clarify this clearly for interview purposes:

🔄 Is `terraform taint` still used in newer versions?

Answer:

- **`terraform taint` is deprecated** in newer Terraform versions (v0.15+).
- Instead, HashiCorp recommends using the `-replace` flag during `terraform apply` or using **`lifecycle` blocks** for controlled behavior.

🆕 Modern Alternatives to `terraform taint`:

1. **`terraform apply -replace=resource.name`**
    - On-demand replacement without marking in state manually.

```
bash
CopyEdit
terraform apply -replace=azurerm_virtual_machine.example
```

2. **Use lifecycle blocks to handle auto-replacement**
    - Example:

```
hcl
CopyEdit
resource "azurerm_virtual_machine" "example" {
  # ... config ...

  lifecycle {
    replace_triggered_by = [azurerm_resource_group.example]
  }
}
```

    - If the resource group changes, the VM will be recreated.

📝 Summary for Interviews:

Earlier, we used `terraform taint` to mark resources for recreation, but in recent Terraform versions, it's deprecated. Now we use `terraform apply -replace=...` or define intelligent rules using the `lifecycle` block (e.g., `replace_triggered_by`) for better control and clarity.

Q5: How to create a VM via terraform with external disk attached?

✅ High-Level Steps:

To create an Azure VM with an external data disk attached using Terraform, you'll need:

1. Resource group
2. Virtual network + subnet
3. Network interface
4. Managed data disk
5. Virtual machine
6. Attach the data disk to the VM

☐ Terraform Example: Azure VM with External Disk

```
hcl
CopyEdit
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "rg" {
  name     = "rg-example"
  location = "East US"
}

resource "azurerm_virtual_network" "vnet" {
  name                = "vnet-example"
  address_space       = ["10.0.0.0/16"]
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
}

resource "azurerm_subnet" "subnet" {
  name                 = "subnet-example"
  resource_group_name  = azurerm_resource_group.rg.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = ["10.0.1.0/24"]
}

resource "azurerm_network_interface" "nic" {
  name                = "nic-example"
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name

  ip_configuration {
    name                          = "ipconfig1"
    subnet_id                     = azurerm_subnet.subnet.id
    private_ip_address_allocation = "Dynamic"
  }
}

resource "azurerm_managed_disk" "data_disk" {
  name                 = "example-data-disk"
  location             = azurerm_resource_group.rg.location
  resource_group_name  = azurerm_resource_group.rg.name
  storage_account_type = "Standard_LRS"
  create_option        = "Empty"
  disk_size_gb         = 20
}

resource "azurerm_linux_virtual_machine" "vm" {
  name                  = "example-vm"
  resource_group_name   = azurerm_resource_group.rg.name
  location              = azurerm_resource_group.rg.location
  size                  = "Standard_B1s"
  admin_username        = "azureuser"
  network_interface_ids = [azurerm_network_interface.nic.id]
  disable_password_authentication = false
  admin_password        = "YourStrongPassword123"

  os_disk {
    name              = "example-os-disk"
    caching           = "ReadWrite"
    storage_account_type = "Standard_LRS"
  }

  source_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "18.04-LTS"
    version   = "latest"
  }

  computer_name  = "examplevm"
  custom_data    = filebase64("init-script.sh")  # Optional cloud-init
script

  tags = {
    environment = "dev"
  }

  storage_data_disk {
    lun                    = 0
    caching                = "ReadWrite"
    managed_disk_id        = azurerm_managed_disk.data_disk.id
    disk_size_gb           = 20
    create_option          = "Attach"
  }
}
```

🔍 Explanation:

- **`azurerm_managed_disk`** creates an empty external disk.
- In the **VM block**, the `storage_data_disk` section attaches the disk using:
    - `managed_disk_id`: Links the external disk
    - `create_option = "Attach"`: Tells Terraform to attach an existing disk
    - `lun`: Logical unit number (0, 1, 2...)

💡 Tips:

- Use **different LUNs** if you plan to attach multiple disks.
- Make sure the disk is in the **same region and resource group** as the VM.
- Use `caching = "None"` or `"ReadOnly"` for data disks depending on work

2. How you migrate data from cloud init to terraform

ChatGPT said:

Great — this is a strong DevOps/Infra-as-Code interview question. Let's break it down clearly and practically.

✅ Q6: How do you migrate from cloud-init to Terraform?

Answer:

Cloud-init and Terraform serve different but overlapping purposes:

- **Cloud-init**: Used to configure instances **at boot time** — install packages, write files, run scripts.
- **Terraform**: Used to **provision infrastructure** (VMs, disks, networks, etc.), but it can **pass user data** to VMs — often cloud-init scripts.

So the migration is about **moving cloud-init logic** into Terraform while **preserving the configuration behavior**.

☐ Migration Steps:

🔹 1. Extract your cloud-init script

If you're currently using a VM with cloud-init, you might already have a YAML file like:

```
yaml
CopyEdit
#cloud-config
packages:
  - nginx
runcmd:
  - systemctl enable nginx
  - systemctl start nginx
```

---

2. Encode cloud-init script for Terraform

Terraform doesn't use raw YAML directly. You must pass cloud-init as **base64-encoded user data** to the VM.

In Terraform:

```
hcl
CopyEdit
resource "azurerm_linux_virtual_machine" "example" {
  name                = "vm1"
  resource_group_name = azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  size                = "Standard_B1s"
  admin_username      = "azureuser"
  network_interface_ids = [azurerm_network_interface.example.id]
  disable_password_authentication = false
  admin_password      = "YourStrongPassword123"

  os_disk {
    caching              = "ReadWrite"
    storage_account_type = "Standard_LRS"
    name                 = "vm1-os-disk"
  }

  source_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "18.04-LTS"
    version   = "latest"
  }

  custom_data = filebase64("cloud-init.yaml")  # <--- Key part
}
```

- The file `cloud-init.yaml` should contain your cloud-init script.
- `filebase64()` encodes it properly.

---

3. (Optional) Break it into modules/variables

If your cloud-init was dynamic (per environment), move parts into variables or templates using `templatefile()`.

🔄 Summary:

To migrate from **cloud-init to Terraform**, you don’t remove cloud-init logic — instead, you **integrate it** by passing the cloud-init script as **`custom_data`** in your Terraform VM resource.

Terraform becomes the orchestrator, and cloud-init still does the instance-level configuration.

✅ Bonus Tip (Best Practice):

- Use `templatefile()` to generate cloud-init from `.tpl` files if you need dynamic behavior:

```
hcl
CopyEdit
custom_data = base64encode(templatefile("${path.module}/cloud-
init.tpl", {
  hostname = "webserver1"
}))
```

Would you like an example with **dynamic cloud-init templating**?

✅ Top Terraform Built-in Functions — Commonly Used / Asked

Function	Purpose	Example	When/Why Used
`file()`	Reads raw file content (string)	`file("init.sh")`	When you need to pass raw script content (e.g., `user_data`)
`filebase64()`	Reads + encodes file to base64	`filebase64("cloud-init.yaml")`	Used in `custom_data` for VMs (Azure, AWS, GCP)
`base64encode()`	Converts a string to base64	`base64encode("hello")`	Encode dynamic strings if needed
`base64decode()`	Converts base64 to plain text	`base64decode(var.b64_string)`	Used for debugging or outputs
`templatefile()`	Renders a `.tpl` template with variables	`templatefile("cloud.tpl", { name = "vm1" })`	Dynamic scripts/cloud-init with variables
`join()`	Joins list elements into a string	`join(",", ["a", "b"])` → `"a,b"`	When passing comma-separated lists (e.g., IPs, tags)
`split()`	Splits a string into a list	`split(",", "a,b")` → `["a", "b"]`	Parsing values from variables

Function	Purpose	Example	When/Why Used
`lookup()`	Looks up a key in a map	`lookup(var.vm_sizes, "dev", "Standard_B1s")`	For environment-specific defaults
`length()`	Returns length of a list or string	`length(var.subnets)`	Count how many items, use in loops or conditions
`element()`	Gets item at a specific index in a list	`element(["a", "b", "c"], 1)` → `"b"`	Useful when iterating in loops
`toset()` / **`tolist()`**	Convert between sets and lists	`tolist(var.tags)`	Needed when working with for_each and count

🧠 Summary Table type of Terraform Block :

Block Type	Purpose
`provider`	Connect to cloud/service
`resource`	Create infrastructure
`data`	Read existing data
`variable`	Take inputs
`output`	Show values after apply
`locals`	Define reusable expressions
`module`	Call other Terraform configs
`terraform`	Set backend, versions, providers
`lifecycle`	Control resource behavior (nested)

Summary Table: Terraform Meta-Arguments

Meta-Argument	Purpose
`count`	Create multiple unnamed resources (by index)
`for_each`	Create multiple resources from map/set (by key)
`provider`	Use a specific provider or alias
`depends_on`	Explicit dependency declaration
`lifecycle`	Control creation, destruction, updates

🧠 Interview Tip:

"Both `provider` and `lifecycle` are **meta-arguments**, but `provider` can exist as a top-level block to configure providers, while `lifecycle` only exists **inside resources** to control behavior."

✅ Q: How can we apply RPO and RTO using Terraform?

Answer:

Terraform **itself doesn't directly define RPO or RTO**, but you can **design and provision infrastructure** in such a way that your **disaster recovery goals** (RPO/RTO) are met.

Let's break it down:

🔄 First, Understand RPO and RTO:

Term	Meaning	Example
RPO (Recovery Point Objective)	How much **data loss** is acceptable	E.g., backups every 15 mins → max 15 mins data loss
RTO (Recovery Time Objective)	How much **downtime** is acceptable	E.g., recover in 30 mins after failure

---

🛠 How Terraform helps with RPO & RTO:

📏 *1. Automated Backup Setup (RPO)*

Use Terraform to provision resources that **automate backups** to reduce data loss:

- Azure: `azurerm_backup_protected_vm`, `azurerm_backup_policy_vm`
- AWS: `aws_backup_plan`, `aws_backup_vault`

Example (Azure VM Backup Policy):

```
hcl
CopyEdit
resource "azurerm_backup_policy_vm" "example" {
  name                = "daily-backup-policy"
  resource_group_name = azurerm_recovery_services_vault.example.name
  recovery_vault_name = azurerm_recovery_services_vault.example.name

  timezone = "UTC"
  backup {
    frequency = "Daily"
    time      = "23:00"
  }

  retention_daily {
    count = 7
  }
}
```

🎯 **This supports RPO** by ensuring backups run every X hours/minutes.

📏 *2. Infrastructure as Code for Rapid Reprovisioning (RTO)*

Store all infra (VMs, disks, network, databases) in Terraform. In case of disaster:

- Run `terraform apply` in DR region
- Spin up entire infra fast
- Use restored backups to rehydrate data

🎯 **This supports RTO** by making recovery infrastructure fast and repeatable.

📏 *3. Use Multi-region Design with Terraform*

Provision **standby infrastructure** or **geo-redundant resources** using Terraform:

- Azure: `geo-redundant storage`, `Availability Zones`, `Traffic Manager`
- AWS: `multi-AZ`, `Route53 failover`, `RDS read replicas`

You can define **primary and secondary** regions in your code using variables:

```
hcl
```

```
CopyEdit
variable "region" {
  default = "eastus"
}
```

🎯 This minimizes downtime (lower RTO) in case of region failure.

📏 *4. Automate Failover Setup*

Use Terraform to configure:

- DNS failover
- Load balancers with health probes
- Azure Traffic Manager or AWS Route53 failover routing

🎯 This ensures **automatic switching** to healthy infrastructure.

✅ Summary: Terraform + RPO/RTO

Goal	Terraform Role
RPO	Automate frequent backups, use geo-redundant storage
RTO	Use IaC to rapidly re-create infra, multi-region design, failover automation

---

🧠 Interview Tip:

"Terraform doesn't enforce RPO/RTO directly, but by designing resilient infrastructure, backup policies, and fast redeployment strategies in Terraform, we can **achieve and automate DR goals**.

Q7: Can we create and deploy policies using Terraform? Or are they predefined, and not good practice to deploy via Terraform?

Answer:

Yes, you **can and should** create and deploy **custom policies** using Terraform — it's actually a best practice in **enterprise-scale environments** to manage **governance** and **compliance** as code.

This applies to **Azure**, **AWS**, **GCP**, etc.

☁ Cloud-wise Answer:

In Azure

Yes, Terraform can manage **Azure Policy Definitions**, **Policy Assignments**, and **Initiatives (Policy Sets)**.

☑ Supported resources:

- `azurerm_policy_definition`
- `azurerm_policy_assignment`
- `azurerm_policy_set_definition`

💡 Example:

```
hcl
CopyEdit
resource "azurerm_policy_definition" "deny_public_ip" {
  name         = "deny-public-ip"
  policy_type  = "Custom"
  mode         = "All"
  display_name = "Deny Public IP"

  policy_rule = <<POLICY
{
  "if": {
    "field": "Microsoft.Network/publicIPAddresses/ipAddress",
    "exists": "true"
  },
  "then": {
    "effect": "deny"
  }
}
POLICY

  metadata = <<METADATA
{
  "category": "Network"
}
METADATA
}
```

This denies the creation of public IPs — a typical security policy.

You can then **assign it** using `azurerm_policy_assignment`.

In AWS

You can manage **SCPs (Service Control Policies)**, **IAM policies**, and **Backup Plans**.

- Use `aws_iam_policy`, `aws_organizations_policy`, `aws_backup_plan`

✅ It's common to enforce tagging, backup, region restrictions via Terraform.

🎯 Why it's a good practice:

- ✅ **Policy-as-Code**: Store policies in Git, version control them
- ✅ **Consistency**: Same policies across all environments (dev, QA, prod)
- ✅ **Compliance**: Audit trail of who changed what
- ✅ **Automation**: No manual policy assignment from portal

⚠ When to be careful:

- Don't accidentally **override existing policies** if managed by another team or tool.
- In multi-team environments, use Terraform workspaces or separate states.
- Use `mode = "Incremental"` for assignments so it **doesn't conflict** with other policies.

🧠 Interview Answer (Sample):

"Yes, Terraform supports deploying policies — in Azure, AWS, and GCP. It's actually a best practice to manage policy-as-code. I've used `azurerm_policy_definition` and `azurerm_policy_assignment` to enforce things like denying public IPs or requiring tags. This approach improves governance, traceability, and automation."

Let me know if you'd like:

- A full example of policy assignment
- Terraform module to manage all policies
- GitOps flow for policy deployment

Q: What is Terraform state file management? Why is it important?

📦 What is a State File in Terraform?

- Terraform uses a file called **`terraform.tfstate`** to **track the real-world state** of your infrastructure.
- It stores:
  - Resource IDs
  - Attributes
  - Dependencies
  - Metadata (e.g., module paths, provider versions)

Without the state file, Terraform wouldn't know what exists or needs to change.

🔐 Types of State File Storage

Type	Description
Local State (default)	`terraform.tfstate` is saved in the working directory
Remote State (recommended for teams)	State is saved in a remote backend like Azure Blob, AWS S3, Terraform Cloud, etc.

🗂 Local State Management (default)

- Easy to get started
- Not suitable for teams (risk of corruption or version conflict)
- Located in the same folder where Terraform is run

☁ Remote State Management (recommended)

Use Terraform `backend` block to configure this:

📘 *Azure Example:*

```
hcl
CopyEdit
terraform {
  backend "azurerm" {
    resource_group_name    = "rg-tfstate"
    storage_account_name   = "tfstatestorage"
    container_name         = "tfstate"
    key                    = "terraform.tfstate"
  }
}
```

Terraform stores the state in an Azure Storage Blob instead of locally.

🔁 Why Manage State Properly?

Reason	Explanation
✅ Track existing infrastructure	Terraform compares state with your `.tf` code to plan changes
✅ Prevents drift	Ensures actual cloud infra matches what’s declared
✅ Enables collaboration	Remote state lets teams safely work together
✅ Enables data source lookups	Output from one resource can be used by another

---

□ Best Practices for State Management

Practice	Why it matters
🔐 Use remote backend with locking	Avoid multiple users editing state simultaneously
🔄 Enable versioning on remote state	Helps recover from mistakes
🔍 Keep `terraform.tfstate` secure	It may contain sensitive info (like passwords, keys)
💥 Don’t edit the state manually	Use `terraform state` commands if needed
🔄 Use `terraform refresh` or `terraform import`	To sync drift or bring existing infra into state

---

⚙️ Useful State Commands

Command	Purpose
`terraform state list`	Show all resources tracked in state
`terraform state show <resource>`	View details of a specific resource
`terraform state rm <resource>`	Remove a resource from state (but not from cloud)

Command	Purpose
`terraform state mv`	Move resource within modules or rename
`terraform refresh`	Sync state with real infrastructure
`terraform import`	Add existing cloud resources to Terraform state

🧠 Interview Summary Answer:

"Terraform uses a `terraform.tfstate` file to track the infrastructure it manages. While it defaults to local storage, for team environments and production, it's best to store it remotely using backends like Azure Blob, AWS S3, or Terraform Cloud. Proper state management is essential for collaboration, security, and consistency. I also use locking, versioning, and Terraform CLI state commands when needed."

Q-Best Practices to Write Terraform Code (Top 10)

---

1. Use Modules to Reuse Code

☑ Split reusable logic into modules — VMs, networks, storage, etc.

```
hcl
CopyEdit
module "vnet" {
  source        = "./modules/vnet"
  vnet_name     = "my-vnet"
  address_space = ["10.0.0.0/16"]
}
```

Helps you avoid repetition and improves maintainability.

---

2. Follow a Logical Folder Structure

☑ Keep code organized.

```
text
CopyEdit
.
├── main.tf
├── variables.tf
├── outputs.tf
├── backend.tf
├── terraform.tfvars
├── modules/
│   ├── vnet/
│   └── vm/
```

---

3. Use `terraform.tfvars` or `*.auto.tfvars` for Values

✅ Separate config values from logic.

```
hcl
CopyEdit
location    = "East US"
vm_size     = "Standard_B1s"
```

Avoid hardcoding values in `main.tf`.

---

4. Use Meaningful Names for Resources and Variables

✅ Keep it readable and understandable to new team members.

```
hcl
CopyEdit
resource "azurerm_network_interface" "web_nic" {
  name = "web-nic"
}
```

---

5. Use Remote Backend for State

✅ Store state in a safe place like Azure Blob, AWS S3, or Terraform Cloud.

Enables collaboration and protects your infra state.

---

6. Lock Provider and Module Versions

✅ Avoid unexpected upgrades.

```
hcl
CopyEdit
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0"
    }
  }
}
```

---

7. Write Outputs for Key Values

✅ Make your Terraform code friendly for automation and pipelines.

```
hcl
CopyEdit
output "vm_ip" {
  value = azurerm_public_ip.vm.ip_address
}
```

---

8. Use `locals` for Repeated Values

✅ Simplify changes and improve readability.

```
hcl
CopyEdit
locals {
  tags = {
    environment = var.env
    owner       = "DevOps"
  }
}
```

---

9. Follow Naming Conventions and Formatting

✅ Use `terraform fmt` to auto-format code.
✅ Use consistent naming for variables, outputs, modules, etc.

Helps readability and standardization.

---

10. Use `for_each` or `count` for Repetitive Resources

✅ Dynamic infrastructure.

```
hcl
CopyEdit
resource "azurerm_network_interface" "nic" {
  for_each = var.subnets

  name = "nic-${each.key}"
  # ...
}
```

🧠 Bonus: Use Policy-as-Code & CI/CD

- Run `terraform validate`, `tflint`, `tfsec`, and `checkov` in pipelines.
- Use GitOps-style workflows with approvals and plan output.

🎯 Interview Summary Answer:

“The best approach to writing Terraform is to follow best practices like modular design, clean structure, using variables and outputs, separating config from logic, and managing state remotely. I also use tools like `terraform fmt`, `tfsec`, and `tflint` for code quality, and version control the entire infrastructure codebase.”

✅ Q11: What is UDR (User-Defined Route) and how can you define rules in it using Terraform?

🚦 What is UDR (User-Defined Route)?

- UDR stands for **User Defined Route** in **Microsoft Azure**.
- It allows you to create **custom routing rules** for traffic within a **Virtual Network (VNet)**.
- Azure has system/default routes by default, but with UDR, you **override default routing** — for scenarios like:
    - Sending traffic to a firewall or NVA (network virtual appliance)
    - Forcing internet-bound traffic via a custom path
    - Peered VNets, hybrid setups

🔧 How to Define UDR in Terraform?

You define UDR in **two steps**:

1. **Create a Route Table**
2. **Add Routes to the Route Table**

☐ Step 1: Create a Route Table

```
hcl
CopyEdit
resource "azurerm_route_table" "example" {
  name                          = "custom-route-table"
  location                      = azurerm_resource_group.example.location
  resource_group_name           = azurerm_resource_group.example.name
  disable_bgp_route_propagation = false
  tags = {
    environment = "dev"
  }
}
```

☐ Step 2: Add Routes to the Route Table

```
hcl
CopyEdit
resource "azurerm_route" "example" {
  name                = "route-to-firewall"
  resource_group_name = azurerm_resource_group.example.name
  route_table_name    = azurerm_route_table.example.name

  address_prefix      = "0.0.0.0/0"  # All traffic
  next_hop_type       = "VirtualAppliance"
  next_hop_in_ip_address = "10.0.1.4"  # IP of firewall or NVA
}
```

✅ Other values for `next_hop_type` include:

- `VirtualNetworkGateway`
- `Internet`
- `VnetLocal`
- `None`

🔗 Step 3: Associate UDR with a Subnet

```
hcl
CopyEdit
resource "azurerm_subnet" "example" {
  name                 = "app-subnet"
  resource_group_name  = azurerm_resource_group.example.name
  virtual_network_name = azurerm_virtual_network.example.name
  address_prefixes     = ["10.0.2.0/24"]
  route_table_id       = azurerm_route_table.example.id
}
```

🧠 Summary (Interview Answer):

"UDR or User Defined Routes are custom routing rules we create in Azure to control how traffic flows within or outside the VNet. Using Terraform, we define UDRs with `azurerm_route_table`, add rules using `azurerm_route`, and associate them with subnets. This is useful for forcing traffic through NVAs, firewalls, or custom gateways to improve security and control.

✅ Q15: What is the purpose of creating a Terraform module?

◈ 1. Reusability

Write once, use many times (e.g., VM, VNet modules).

◈ 2. Organization

Break big configs into smaller, manageable parts.

◈ 3. Consistency

Ensure same standards across all environments (dev/prod).

◈ 4. Scalability

Deploy many resources with different inputs easily.

◈ 5. Team Collaboration

Helps teams work on separate modules in parallel.

✅ Q16: How to manage 3 environments (dev, test, prod) in a single Azure subscription with dedicated state files?

🎯 Goal:

- One subscription
- Three environments: `dev`, `test`, `prod`
- Separate state files
- Clean isolation

🔍 Best Practice Approach: Use Workspaces + Remote Backend

✅ Option 1: Using Remote Backend + Workspaces

Use **Azure Blob Storage** as backend and create **Terraform workspaces** for each environment.

🔧 Backend Setup (common for all envs)

```
hcl
CopyEdit
terraform {
  backend "azurerm" {
    resource_group_name  = "tfstate-rg"
    storage_account_name = "tfstateaccount"
    container_name       = "tfstate"
    key                  = "env-${terraform.workspace}.tfstate"
  }
}
```

✅ Here, each workspace (dev, test, prod) will automatically get its **own state file**:

- `env-dev.tfstate`
- `env-test.tfstate`
- `env-prod.tfstate`

🔄 Workspace Commands

```
bash
CopyEdit
terraform workspace new dev
terraform workspace new test
terraform workspace new prod

terraform workspace select dev
terraform apply
```

Repeat for `test` and `prod`.

🗀 Folder Structure (optional)

All code shared, controlled by `terraform.workspace`

```
bash
CopyEdit
infra/
├── main.tf
├── variables.tf
├── backend.tf
```

✅ Option 2: Separate Folders per Environment (also valid)

```
bash
CopyEdit
environments/
├── dev/
│   └── backend.tf (key = dev.tfstate)
├── test/
│   └── backend.tf (key = test.tfstate)
├── prod/
│   └── backend.tf (key = prod.tfstate)
```

This also gives strict separation, useful in CI/CD pipelines.

🧠 Summary Answer (for Interview):

"I'll use remote state via Azure Blob Storage with either workspaces or separate folders per environment. Each environment will have its own isolated state file like `dev.tfstate`, `test.tfstate`, and `prod.tfstate`. This keeps infra separate, safe, and manageable within one subscription."

✅ Q18: What is the command to update the Terraform provider?

"To update a Terraform provider, I modify the version in the `required_providers` block and then run `terraform init -upgrade` to fetch the updated provider.

19.what is terraform refresh command

`terraform refresh` updates the Terraform state file to reflect the actual real-world infrastructure, useful when manual changes are made. It doesn't apply any changes. In newer versions, it's replaced with `terraform apply -refresh-only`.

3. How you review terraform Code

"I review Terraform code by checking best practices, ensuring secure and correct resource definitions, validating remote state and locking, using linters like tflint and tfsec, reviewing terraform plan outputs carefully, and confirming proper documentation and formatting."

Question. Which configuration you are working in terraform

“I work with provider setups, resource declarations, variables, outputs, and remote backends. I also create and use modules for reusable code and sometimes include lifecycle blocks and provisioners for advanced resource control.”

Question. SQL instance created by terraform but some attributes you don't want to expose with terraform, how you will do it

“To avoid exposing sensitive SQL instance attributes, I don’t output them unless necessary. If I must output, I mark them `sensitive = true` so Terraform hides them. I also use secure backends with encryption to protect state, and I avoid storing secrets in plain text variables

8- How did you use terraform import and have you heard about terraformer?

“I use `terraform import` to bring single existing resources into Terraform state, linking them with pre-written resource blocks. For large-scale import, I’ve heard about Terraformer — a tool that automates generating Terraform code and state for existing infrastructure, saving manual effort.”
terraformer import azure --resources=resourceGroups,virtualNetworks --connect=true

Question. Execution order of terraform blocks.

"Terraform processes the configuration in this order: first the terraform block for settings, then provider configurations, followed by variable declarations and locals. It fetches data sources before creating resources. Modules are evaluated recursively with the same rules, and finally outputs are calculated after resource creation.

How do you handle the sensitive data like passwords or access keys in Terraform configurations?

"I handle sensitive data by marking variables as sensitive, never hardcoding secrets, and using secure secret stores like Azure Key Vault or AWS Secrets Manager. I keep state files encrypted in remote backends and avoid outputting sensitive info unless necessary, marking outputs sensitive to hide them."

Q23: Two resources created two weeks ago got destroyed and recreated. Apart from `terraform taint`, what scenarios cause this?

Common scenarios causing resource destroy & recreate:

1. **Immutable fields changed**
    - Some resource properties **cannot be updated in-place** and require resource replacement.

    - Example: Changing VM size in Azure or changing a subnet’s address prefix.
2. **Provider or Terraform version upgrade**
    - Sometimes, provider updates change resource behavior causing recreation.
3. **Changes in dependent resources or references**
    - If a resource depends on another that’s replaced, it might cascade.
4. **Manual changes outside Terraform (drift)**
    - If infra is modified manually and Terraform plans to reconcile, sometimes recreation happens.
5. **Changes to resource identifiers or names**
    - Renaming resources or IDs may cause destroy and recreate.
6. **Lifecycle rules with `create_before_destroy = false`**
    - By default, Terraform destroys then creates; without `create_before_destroy`, downtime or recreation occurs.
7. **Switching resource types or moving resources between modules**
    - Moving resources or changing resource types can trigger replacement.
8. **Changes in backend or workspace causing state mismatch**
    - If state is corrupted or not properly synced, Terraform may think resources need recreation.

🧠 Interview Summary:

“Resources get destroyed and recreated when immutable properties change, provider upgrades cause differences, manual drift happens, or naming changes occur. Lifecycle blocks and state management also influence this behavior, beyond explicit `terraform taint` commands.”

25blob strorage for state file then someone delete and donot have backup file . and we apply terraform apply

“If the state file is deleted with no backup, Terraform will treat all resources as missing and try to recreate them, potentially causing conflicts or downtime. Recovery involves restoring backups if available or using terraform import to rebuild state. To prevent this, I enable blob versioning, soft delete, and enforce state backups.”

10. Did you face any problem while working on terraform

Yes, I have faced issues like state conflicts, unintended resource recreation, and managing sensitive data. I solved these by implementing remote state locking, lifecycle policies, securing secrets properly, and using tools like Terraformer to ease imports.”

4. How do you connect your Terraform to your Azure portal?

- Answer: You connect Terraform to Azure by authenticating using Azure CLI, a service principal, or a managed identity, and configuring the Azure provider in your Terraform code.

Q: Are you create a AKS Cluster using terraform? How your creating worker node?

```
resource "azurerm_kubernetes_cluster" "aks" {
  name                = "myAKSCluster"
  location            = var.location
  resource_group_name = var.resource_group_name
  dns_prefix          = "myaks"

  default_node_pool {
    name       = "default"
    node_count = 2              # Number of worker nodes
    vm_size    = "Standard_DS2_v2"  # Node VM size
    os_disk_size_gb = 30
  }

  identity {
    type = "SystemAssigned"
  }

  kubernetes_version = "1.28.5"
}
```

Q – Write config for AKS ,ACR & NSG

```
provider "azurerm" {
  features {}
}
resource "azurerm_resource_group" "rg" {
  name     = "example-rg"
  location = "eastus"
}
resource "azurerm_container_registry" "acr" {
  name                = "exampleacr123456"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  sku                 = "Basic"
  admin_enabled       = true
}

# NSG (Network Security Group)
resource "azurerm_network_security_group" "nsg" {
  name                = "example-nsg"
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
}

# AKS (Azure Kubernetes Service)
resource "azurerm_kubernetes_cluster" "aks" {
  name                = "example-aks"
  location            = azurerm_resource_group.rg.location
  resource_group_name = azurerm_resource_group.rg.name
  dns_prefix          = "exampleaks"
  default_node_pool {
    name       = "default"
    node_count = 1
    vm_size    = "Standard_DS2_v2"
  }

  identity {
    type = "SystemAssigned"
  }

  # Optional: Attach ACR permission to AKS
  depends_on = [azurerm_container_registry.acr]
}

# Output
output "acr_login_server" {
  value = azurerm_container_registry.acr.login_server
}

output "aks_kube_config" {
  value     = azurerm_kubernetes_cluster.aks.kube_config_raw
  sensitive = true
}


resource "azurerm_network_security_rule" "allow_blob_storage" {
  name                        = "AllowBlobStorageHTTPS"
  priority                    = 100
  direction                   = "Outbound"
  access                      = "Allow"
  protocol                    = "Tcp"
  source_port_range           = "*"
  destination_port_range      = "443"
  source_address_prefix       = "*"
  destination_address_prefix  = "Storage"
  resource_group_name         = azurerm_resource_group.rg.name
  network_security_group_name = azurerm_network_security_group.nsg.name
}
```

Q7. Write the key vault creation code in terraform ?

```
resource "azurerm_key_vault" "kv" {
  name                        = "examplekeyvault123"
  location                    = azurerm_resource_group.rg.location
  resource_group_name         = azurerm_resource_group.rg.name
  tenant_id                   = data.azurerm_client_config.current.tenant_id
  sku_name                    = "standard"
  soft_delete_enabled         = true
  purge_protection_enabled    = true
}
```

What is a dynamic block in terraform and when they used?

"Dynamic blocks make Terraform configurations scalable and maintainable when handling nested structures that need to be repeated based on variable inputs."

7. How you can back up the terraform configuration files?

Terraform configuration files can be backed up like any source code — using version control systems like Git and storing them in remote repositories such as GitHub, GitLab, Azure Repos, or Bitbucket. For state files, separate strategies are used.

9. Types of input in terraform

☑ **Summary:**

Type	Description	Example
string	Single text value	`"eastus"`
number	Numeric value	`8080`
bool	Boolean (true/false)	`true`
list	Ordered collection	`["a", "b", "c"]`
map	Key-value pairs	`{key1 = "value1", key2 = "v2"}`
set	Unique unordered collection	`["tag1", "tag2"]`
object	Structured attribute collection	`{name=string, port=number}`
tuple	Fixed-length heterogeneous list	`["string", 123]`

8. Terraform provisioner and null resource

```
resource "azurerm_linux_virtual_machine" "example" {
  # VM config here

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx"
    ]

    connection {
      type     = "ssh"
      user     = "azureuser"
      private_key = file(var.private_key_path)
      host     = self.public_ip_address
```

- scripts or commands **during resource creation or destruction** (local or remote).
- Use **remote-exec** for remote commands and **local-exec** for local machine commands.
- **null_resource** is a dummy resource used to attach provisioners or triggers when no real resource is managed.
- Provisioners should be used sparingly and replaced with better config management tools when possible.

5- Assume Manager A and Manager B uses Subscription A and Subscription B and we need to create VM A in Subscription A and VM B in Subscription B in single terraform file and after VM creation I have to do ssh from VM A to VM B. How and What approach will you follow?

onfigure Multiple Providers with Aliases

Terraform allows configuring multiple providers for different subscriptions using `alias`.

Example:

hcl
CopyEdit

```
provider "azurerm" {
  alias           = "subA"
  subscription_id = var.subscription_id_A
  features        = {}
}

provider "azurerm" {
  alias           = "subB"
  subscription_id = var.subscription_id_B
  features        = {}
}
```

---

2 Create VM A Using Provider for Subscription A

hcl
CopyEdit

```
resource "azurerm_network_interface" "nic_A" {
  provider = azurerm.subA

  name                = "nicA"
  location            = var.location_A
  resource_group_name = var.rg_name_A
  # subnet etc.
}

resource "azurerm_linux_virtual_machine" "vm_A" {
  provider              = azurerm.subA
  name                  = "vmA"
  resource_group_name   = var.rg_name_A
  location              = var.location_A
  size                  = "Standard_B1s"
  network_interface_ids = [azurerm_network_interface.nic_A.id]
  # other config
}
```

---

3 Create VM B Using Provider for Subscription B

```
hcl
CopyEdit
resource "azurerm_network_interface" "nic_B" {
  provider = azurerm.subB

  name                = "nicB"
  location            = var.location_B
  resource_group_name = var.rg_name_B
  # subnet etc.
}

resource "azurerm_linux_virtual_machine" "vm_B" {
  provider              = azurerm.subB
  name                  = "vmB"
  resource_group_name   = var.rg_name_B
  location              = var.location_B
  size                  = "Standard_B1s"
  network_interface_ids = [azurerm_network_interface.nic_B.id]
  # other config
}
```

---

4 Enable Network Connectivity

- Ensure both VMs are reachable via **public IP** or VPN/Peering if private IPs are used.
- Create **Public IPs** and assign to both VMs' NICs or set up VNet peering between subscription VNets.

Example:

```
hcl
CopyEdit
resource "azurerm_public_ip" "pubip_A" {
  provider = azurerm.subA
  name                = "pubipA"
  location            = var.location_A
  resource_group_name = var.rg_name_A
  allocation_method   = "Static"
}

resource "azurerm_network_interface" "nic_A" {
  provider = azurerm.subA
  # ...
  ip_configuration {
    name                          = "ipconfig1"
    subnet_id                     = azurerm_subnet.subnet_A.id
    private_ip_address_allocation = "Dynamic"
    public_ip_address_id          = azurerm_public_ip.pubip_A.id
  }
}
```

---

5 SSH from VM A to VM B

- Store VM B’s **public IP** as output or variable.
- Use a **remote-exec provisioner** on VM A or a **null_resource** to SSH into VM B.

Example with `null_resource` and `local-exec` or `remote-exec`:

```
hcl
CopyEdit
resource "null_resource" "ssh_from_A_to_B" {
  depends_on = [azurerm_linux_virtual_machine.vm_A,
azurerm_linux_virtual_machine.vm_B]

  provisioner "remote-exec" {
    connection {
      host        = azurerm_linux_virtual_machine.vm_A.public_ip_address
      user        = "azureuser"
      private_key = file(var.private_key_path)
    }

    inline = [
      "ssh -o StrictHostKeyChecking=no
azureuser@${azurerm_linux_virtual_machine.vm_B.public_ip_address} hostname"
    ]
  }
}
```

To deploy VMs in two different Azure subscriptions within a single Terraform configuration, I configure two `azurerm` providers with aliases, each targeting one subscription. I create the VMs specifying the relevant provider. For SSH communication, I ensure network connectivity using public IPs or peering and use provisioners or external scripts with SSH keys to allow VM A to SSH into VM B."

12. Where provisioners are defined into terraform

```
resource "azurerm_linux_virtual_machine" "example" {
  # resource configuration...

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx"
    ]

    connection {
      type        = "ssh"
      user        = "azureuser"
      private_key = file(var.private_key_path)
      host        = self.public_ip_address
    }
  }
}
```