# 50 most asked Terraform questions in interviews

## Terraform Fundamentals

**1. What is Terraform and how does it differ from CloudFormation or Ansible?**
Terraform is an open-source Infrastructure as Code (IaC) tool by HashiCorp that automates infrastructure provisioning across multiple clouds. Unlike **AWS CloudFormation**, which is AWS-specific, Terraform supports **multi-cloud and hybrid environments**. Compared to **Ansible**, which is configuration management and procedural, Terraform focuses on **declaratively defining infrastructure state** and managing dependencies automatically.

---

**2. Explain the Terraform workflow (init → plan → apply → destroy).**

- **Init:** Initializes the working directory, downloads provider plugins, and prepares the backend.
- **Plan:** Compares the desired configuration with the current state and shows what changes will be made.
- **Apply:** Executes the plan, creating or modifying resources to match the declared state.
- **Destroy:** Removes all resources defined in the configuration safely.

---

**3. What is declarative IaC vs imperative? Why Terraform is declarative?**
In **declarative IaC**, you describe *what* the infrastructure should look like, and the tool figures out *how* to get there. In **imperative IaC**, you define *step-by-step commands* to reach that state. Terraform is **declarative** because users define the desired end state in `.tf` files, and Terraform automatically computes the necessary actions to reach it.

---

**4. What is a provider in Terraform?**
A **provider** is a plugin that allows Terraform to interact with APIs of external platforms like AWS, Azure, GCP, or Kubernetes. Providers handle CRUD (Create, Read, Update, Delete) operations for specific resources. Each provider must be declared in the configuration to enable Terraform to manage those resources.

---

**5. What is HCL? What advantages does it bring?**
HCL (HashiCorp Configuration Language) is Terraform's domain-specific language designed for human-readable and machine-friendly IaC definitions. Its main advantages include

**readability**, **type safety**, and **support for complex expressions**, making it easier to maintain and reuse configurations across teams.

---

**6. What is a resource vs data source?**
A **resource** defines infrastructure objects that Terraform manages — for example, creating an AWS EC2 instance or S3 bucket.
A **data source** retrieves existing information from providers (like a VPC ID or AMI image) **without creating or changing it**, allowing you to reference pre-existing infrastructure.

---

**7. What are variables and outputs used for?**
**Variables** make Terraform configurations flexible and reusable by allowing parameterization of values (like region or instance type).
**Outputs** expose information after deployment — for instance, printing an IP address or database endpoint for use in other modules or tools.

---

**8. What is terraform.tfvars and when is it used?**
`terraform.tfvars` is a file used to define variable values separately from the main configuration. It's useful when managing different environments (e.g., dev, staging, prod) where the same code should run with different inputs. Terraform automatically loads it during runtime.

---

**9. What are .auto.tfvars files?**
Files ending in `.auto.tfvars` or `.auto.tfvars.json` are automatically loaded by Terraform **without needing to specify them** with the `-var-file` flag. They're ideal for storing environment-specific or shared variable sets that should always be applied.

---

**10. Explain locals — when would you use them?**
**Locals** define local values within a Terraform module using expressions. They're useful for simplifying complex expressions, avoiding repetition, or making code more readable. For example, you can define a computed name prefix once and reuse it throughout your configuration.

## State Management

**11. What is Terraform state and why is it needed?**
Terraform state is a file (`terraform.tfstate`) that tracks the real-world infrastructure managed by Terraform. It maps resource configurations in your code to actual cloud resources.
It's essential because Terraform uses this state to determine what changes to make during `plan` and `apply`, enabling efficient updates and dependency management.

---

**12. Where is state stored by default?**
By default, Terraform stores state **locally** in a file called `terraform.tfstate` in the current working directory. This is fine for single-user use, but not for shared environments or production setups.

---

**13. Why is storing state locally dangerous in teams?**
Local state is unsafe in teams because:

- It can easily get out of sync if multiple people modify resources.
- It risks **data loss** if the file is deleted or corrupted.
- It doesn't provide **state locking**, which can lead to race conditions when multiple users apply changes simultaneously.

---

**14. How do you store remote backend state? Which backends have you used?**
You can configure a **remote backend** in Terraform to store state securely and centrally.
Common options include:

- **AWS S3** (often paired with DynamoDB for locking)
- **Terraform Cloud/Enterprise**
- **Google Cloud Storage (GCS)**
- **Azure Blob Storage**
  These remote backends allow versioning, encryption, collaboration, and automatic state locking.

---

**15. What is state locking and why is it critical?**
**State locking** prevents multiple Terraform processes from writing to the same state file simultaneously. Without it, concurrent changes can corrupt the state or cause inconsistent infrastructure.

Backends like **Terraform Cloud** and **S3+DynamoDB** provide automatic locking mechanisms to avoid conflicts.

---

**16. What happens if two engineers apply Terraform at the same time?**
Without locking, both engineers might attempt to modify the same resources concurrently, leading to **race conditions**, **state corruption**, or **unexpected infrastructure changes**.
With proper state locking enabled, one apply waits until the other completes.

---

**17. What is terraform refresh?**
`terraform refresh` updates the state file with the current real-world infrastructure data without changing resources.
It's useful when external changes (like manual edits in the cloud console) may have occurred, and you need to sync Terraform's view with reality.

---

**18. When do you use terraform state rm or mv?**

- `terraform state rm`: Removes a resource from state without deleting it in the cloud (used if a resource is managed outside Terraform).
- `terraform state mv`: Renames or moves a resource within the state file, often during refactoring (e.g., moving to a new module).

---

**19. What is drift? How do you detect and fix it?**
**Drift** occurs when real infrastructure diverges from Terraform's configuration (for example, manual edits in the cloud console).
You can detect drift by running `terraform plan` or `terraform refresh`.
To fix it, either:

- Revert manual changes to match code, or
- Update Terraform configuration to reflect reality.

---

**20. How do you migrate state from S3 → Terraform Cloud or Consul etc.?**
You can migrate state using:

- `terraform state pull` → retrieves the current state from S3.
- `terraform login` and update backend configuration for Terraform Cloud.

- `terraform init -migrate-state` → automatically transfers the existing state to the new backend.
  This ensures all resources are retained and tracked under the new backend safely.

## Modules & Reusability

**21. What is a Terraform module?**
A **Terraform module** is a container for multiple resources used together. It allows you to group related resources (like VPCs, EC2s, and IAM roles) into a reusable package.
Modules help enforce consistency, reduce code duplication, and simplify complex infrastructure by abstracting reusable logic.

---

**22. How do you structure reusable modules across environments?**
A common structure is:

```
/modules
   /vpc
   /ec2
   /rds
/environments
   /dev
   /stage
   /prod
```

Each environment uses the same module definitions with different variable values (e.g., instance size, region). This allows consistency while enabling customization.

---

**23. What is the difference between root module vs child module?**
The **root module** is the main entry point where Terraform commands (`apply`, `plan`, etc.) are executed — typically your environment folder.
**Child modules** are the reusable building blocks called from the root or other modules using the `module` block.

---

**24. When should you NOT use modules?**
Avoid using modules when:

- The infrastructure is **small or one-off**, where a module adds unnecessary complexity.
- You're still experimenting or prototyping.
- Reusability adds more **maintenance overhead** than benefit (for example, very dynamic or temporary stacks).

---

**25. How do you version Terraform modules?**
Modules can be versioned via the **`version`** argument in the module block when pulling from registries or Git repositories.
Example:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.19.0"
}
```

Versioning ensures stability and reproducibility across deployments.

---

**26. How do you pass outputs from one module to another?**
You can **reference a module's outputs** directly:

```
module "vpc" {
  ...
}

module "ec2" {
  vpc_id = module.vpc.vpc_id
}
```

This allows data sharing between modules while maintaining encapsulation and clean architecture.

---

**27. What are count and for_each? When would you choose one over the other?**

- **count**: Creates multiple instances of a resource or module based on an index (`count.index`).
- **for_each**: Iterates over maps or sets, allowing **named instances** (key-based access).
  Use **`count`** for simple lists, **`for_each`** when resources must be uniquely identified (e.g., per subnet name).

---

**28. Explain module sources (local, GitHub, Terraform Registry).**
Modules can be pulled from several sources:

- **Local path:** `source = "../modules/vpc"`
- **GitHub repo:** `source = "github.com/org/repo//modules/vpc?ref=v1.0.0"`
- **Terraform Registry:** `source = "terraform-aws-modules/vpc/aws"`
  Each provides flexibility for collaboration and version control.

---

**29. How do you handle multiple environments (dev/stage/prod) — workspaces vs separate state?**
Two main strategies:

- **Separate state files** (recommended for isolation): each environment has its own backend/state file.
- **Workspaces:** a single configuration but separate logical states.
  For production systems, separate state backends are safer and give better access control.

---

**30. What are Terraform workspaces? Pros/cons?**
Workspaces allow multiple **state files** for the same configuration, useful for lightweight environment separation (e.g., `dev`, `prod`).
**Pros:** Easy to set up, share the same code.
**Cons:** Harder to isolate environments, can cause confusion or accidental cross-environment applies.

## Plans, Apply & Lifecycle

**31. What is the purpose of `terraform plan`?**
`terraform plan` is a dry-run command that shows what changes Terraform *will* make without actually applying them.
It compares the desired configuration (in `.tf` files) with the current state to display actions like create, update, or destroy.
This helps validate infrastructure changes before execution and prevents unintended modifications.

---

**32. What does the `-target` flag do? When should it not be used?**
`-target` limits Terraform's actions to a specific resource or module. Example:

```
terraform apply -target=aws_instance.app
```

It's useful for debugging or partial applies — but **should not** be used regularly because it bypasses dependency resolution, leading to **drift** or incomplete infrastructure creation.

---

**33. Explain `create_before_destroy`.**
`create_before_destroy` is a lifecycle rule that ensures Terraform creates a new resource **before destroying** the old one.

Example use case: when replacing instances or load balancers in production where downtime must be avoided.

```
lifecycle {
  create_before_destroy = true
}
```

---

**34. What does `depends_on` do? Why is it sometimes overused?**
`depends_on` explicitly defines dependencies between resources, forcing Terraform to build or destroy them in a specific order.
Example:

```
resource "aws_instance" "app" {
  depends_on = [aws_iam_role.app_role]
}
```

However, it's often overused — Terraform's dependency graph usually handles relationships automatically. Overuse makes code harder to maintain.

---

**35. What happens if a resource is deleted manually from the AWS console?**
Terraform's state still believes the resource exists.
When you run `terraform plan`, Terraform will detect the missing resource and plan to **recreate** it to match the configuration.
You can also use `terraform refresh` to sync state manually.

---

**36. How do you ignore specific attribute changes (like random IDs or tags)?**
You can use the `ignore_changes` lifecycle rule to tell Terraform to skip updates for certain attributes:

```
lifecycle {
  ignore_changes = [tags, name_prefix]
}
```

This prevents unnecessary updates caused by automatically generated or external values.

---

**37. What are lifecycle rules in Terraform?**
Lifecycle rules modify how Terraform manages resource creation, replacement, and destruction. Common ones include:

- `create_before_destroy` — ensures zero downtime.
- `prevent_destroy` — blocks accidental deletions.

- `ignore_changes` — avoids unnecessary updates.
  They're critical for controlling sensitive or production resources.

---

**38. Can Terraform import existing infra? How?**
Yes. Terraform can import real-world resources into its state using:

```
terraform import aws_instance.example i-1234567890abcdef0
```

After importing, you must manually add the corresponding configuration in `.tf` files so Terraform can manage it moving forward.

---

**39. Explain `terraform taint` and why it's deprecated.**
`terraform taint` used to mark a resource for recreation during the next `apply`.
It's deprecated in favor of:

```
terraform apply -replace=aws_instance.app
```

The new command is safer and integrates better with Terraform's plan/apply workflow.

---

**40. Difference between `terraform destroy` and just deleting the state file?**

- `terraform destroy`: Safely destroys all managed resources *and updates the state* to reflect deletion.
- Deleting the state file: Terraform loses track of resources but does **not** delete them in the cloud — leading to **orphaned resources** and billing leaks.
  Always use `destroy` for clean teardowns.

## Security, CI/CD & Real-World Use

**41. How do you manage secrets and avoid hardcoding credentials?**
Never store secrets (like AWS keys) in `.tf` files or Git.
Instead:

- Use **environment variables** (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`)
- Integrate with secret managers like **AWS Secrets Manager**, **Azure Key Vault**, or **HashiCorp Vault**
- Use **Terraform Cloud** or **Workspaces** with sensitive variable storage
- Employ **OIDC** or **IAM roles** for temporary credentials instead of static keys.

---

**42. How do you integrate Terraform with GitHub Actions / Jenkins / GitLab CI?**
Integrate Terraform as part of the CI/CD pipeline to automate plan/apply:

- **GitHub Actions:** use actions like `hashicorp/setup-terraform` and run `terraform fmt`, `validate`, `plan`, `apply`
- **Jenkins / GitLab CI:** create pipeline stages for `init`, `plan`, and `apply`
- Store remote state in a backend and use service accounts or OIDC for authentication
- Add manual approval gates before `apply` in production.

---

**43. What's the difference between `terraform validate` vs `terraform fmt`?**

- `terraform validate` checks syntax and logic errors in configuration files (ensures valid HCL).
- `terraform fmt` automatically formats `.tf` files to standard HCL style.
  Together, they keep Terraform code **consistent** and **error-free** in pipelines.

---

**44. What linters or policies have you used? (TFLint / TFsec / OPA / Sentinel)**

- **TFLint:** static analysis for Terraform code (detects unused variables, typos, invalid references).
- **TFsec:** security scanner that detects misconfigurations (e.g., open S3 buckets).
- **OPA / Conftest:** enforces custom policies as code (e.g., mandatory tags).
- **Sentinel:** HashiCorp's policy engine for governance in Terraform Cloud/Enterprise.
  These tools help enforce compliance, security, and code quality in CI/CD.

---

**45. How do you implement approval workflows before apply?**

- Use **manual approval steps** in CI/CD tools (GitHub "environment protection rules", GitLab `manual` jobs).
- In **Terraform Cloud**, use "Run Tasks" or "Sentinel policies" to enforce approval gates.
- Require peer reviews of `terraform plan` output via pull requests before merging.

---

**46. How would you test Terraform changes safely?**

- Run `terraform plan` in CI to preview all changes.
- Use **Terraform workspaces** or **temporary environments** for testing.

- Apply to **sandbox or staging** accounts before production.
- For automated tests: use **Terratest (Go)** or **Checkov** for static validation.

---

**47. How do you handle provider upgrades or version drift?**

- Pin exact provider versions in `required_providers` block to ensure reproducibility.
- Test upgrades in a non-prod environment first.
- Use `terraform init -upgrade` to refresh providers.
- Regularly review `terraform providers lock` file for drift.
  Automation pipelines can flag outdated providers via Dependabot or Renovate.

---

**48. How do you deploy the same exact module to AWS & GCP?**
Use **cloud-agnostic modules** and pass provider configurations dynamically:

```
module "storage" {
  source = "./modules/storage"
  providers = {
    aws = aws.us
    google = google.eu
  }
}
```

Inside the module, use conditional logic (`count`, `for_each`, or `terraform.workspace`) to handle cloud-specific differences.
Alternatively, use **Terragrunt** for multi-cloud orchestration.

---

**49. Terraform vs Pulumi — what's your take?**

- **Terraform:** Declarative, HCL-based, widely adopted, huge ecosystem, easier for teams.
- **Pulumi:** Imperative, uses real programming languages (Python, Go, TS), allows loops/conditions directly.
  Terraform = more standard & predictable; Pulumi = more flexible & developer-friendly.
  I prefer Terraform for infra teams, Pulumi for app-centric DevOps workflows.

---

**50. What's the biggest Terraform disaster you've seen — and how did you fix/prevent it?**
A team ran `terraform destroy` on the wrong workspace due to misconfigured backend.
It wiped shared infrastructure.
Fixes:

- Added **backend environment isolation** (separate states per env)
- Implemented **approval gates before destroy**
- Enforced `prevent_destroy = true` on critical resources
- Added `IAM least privilege` to block destructive operations in prod.