Top 100 Senior-Level Terraform Interview Questions

Compiled from FAANG, Startups, Cloud-Native Companies

Table of Contents

Core Concepts

Q1: What is the difference between `terraform import` and `terraform taint`?

Answer: `terraform import` brings existing infrastructure into Terraform state without recreating it. `terraform taint` marks a resource for recreation on next apply (deprecated in favor of `-replace` flag).

Q2: Explain the Terraform workflow and lifecycle.

Answer: Write → Init → Plan → Apply → Destroy

- **Write**: Create .tf files
- **Init**: Download providers and modules
- **Plan**: Preview changes
- **Apply**: Execute changes
- **Destroy**: Remove infrastructure

Q3: What happens during `terraform init`?

Answer:

- Downloads provider plugins
- Initializes backend
- Downloads modules
- Creates `.terraform` directory
- Generates dependency lock file

Q4: Difference between `count` and `for_each`? When to use which?

Answer:

- `count`: Numeric indexing (0,1,2...) - use for identical resources
- `for_each`: Map/set keys - use when resources need unique identifiers
- `for_each` is more stable when list items change

Q5: What are Terraform providers? Name 5 popular ones.

Answer: Plugins that interact with APIs of cloud platforms.

- AWS, Azure, GCP, Kubernetes, Docker, Datadog, GitHub, Vault, Cloudflare, PostgreSQL

Q6: Explain Terraform's dependency graph.

Answer: DAG (Directed Acyclic Graph) that determines resource creation order based on dependencies (implicit via references, explicit via `depends_on`).

Q7: What is the difference between `variable` and `local`?

Answer:

- `variable`: Input parameters (can be passed externally)
- `local`: Computed values used within module (internal only)

Q8: How do you reference outputs from one module in another?

Answer: `module.<module_name>.<output_name>`

```hcl
module.vpc.vpc_id
```

Q9: What are data sources in Terraform?

Answer: Read-only queries to fetch information from providers without creating resources.

```hcl
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"]
}
```

Q10: Explain implicit vs explicit dependencies.

Answer:

- **Implicit**: Automatic via resource references (`aws_instance.web.id`)
- **Explicit**: Manual using `depends_on` (use sparingly)

Q11: What is the purpose of `terraform.tfvars` vs `variables.tf`?

Answer:

- `variables.tf`: Declares variables (structure)
- `terraform.tfvars`: Provides values (data)

Q12: How does Terraform handle resource lifecycle?

Answer: Using lifecycle blocks:

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

Q13: What is `terraform validate` and when do you use it?

Answer: Checks configuration syntax and internal consistency. Run before plan/apply. Doesn't check provider credentials or API availability.

Q14: Explain `terraform.tfstate.backup`.

Answer: Automatically created backup of previous state before modifications. Useful for manual rollback in emergencies.

Q15: What are provisioners? Why are they discouraged?

Answer: Execute scripts on local/remote machines. Discouraged because:

- Not idempotent
- Error handling is poor
- Better to use configuration management tools (Ansible, cloud-init)

Q16: Difference between `terraform refresh` and `terraform apply -refresh-only`?

Answer: Both update state with real infrastructure, but `refresh` is deprecated. Use `apply -refresh-only` which is safer and clearer in intent.

Q17: What is the `-target` flag and when should it be used?

Answer: Applies changes to specific resources only. Use for emergencies or breaking circular dependencies. **Avoid in normal workflow**.

Q18: How do you handle conditional resource creation?

Answer: Using `count` with ternary operator:

```hcl
count = var.create_resource ? 1 : 0
```

Q19: What are dynamic blocks?

Answer: Generate nested configuration blocks dynamically:

```hcl
dynamic "ingress" {
  for_each = var.ingress_rules
  content {
    from_port = ingress.value.from_port
    to_port   = ingress.value.to_port
  }
}
```

Q20: Explain `terraform fmt` and `terraform show`.

Answer:

- `terraform fmt`: Auto-formats code to canonical style
- `terraform show`: Displays human-readable state or plan

State Management

Q21: What is Terraform state and why is it critical?

Answer: JSON file mapping Terraform config to real resources. Critical for tracking resource metadata, performance, and collaboration.

Q22: What are the dangers of manual state file editing?

Answer: Can corrupt state, cause data loss, create inconsistencies. Always use `terraform state` commands.

Q23: Explain state locking and its importance.

Answer: Prevents concurrent state modifications. Uses DynamoDB (S3 backend), Azure Blob locks, or native locking. Prevents race conditions.

Q24: How do you migrate state between backends?

Answer:

```hcl
# 1. Update backend config
# 2. terraform init -migrate-state
# 3. Confirm migration
```

Q25: What is `terraform state mv` used for?

Answer: Rename resources or move them between modules without destroying/recreating:

```bash
terraform state mv aws_instance.old aws_instance.new
```

Q26: How do you remove a resource from state without destroying it?

Answer: `terraform state rm <resource_address>`

Q27: What is remote state data source?

Answer: Read outputs from another Terraform state:

```hcl
data "terraform_remote_state" "vpc" {
  backend = "s3"
  config = {
    bucket = "terraform-state"
    key    = "vpc/terraform.tfstate"
  }
}
```

Q28: Explain state file security concerns.

Answer: Contains sensitive data (passwords, keys). Use:

- Encryption at rest
- Access controls
- Never commit to Git
- Remote backends with encryption

Q29: How do you handle state file conflicts in teams?

Answer:

- Remote backend with locking
- CI/CD automation
- State locking timeout configuration
- Clear team processes

Q30: What is `terraform state pull` and `terraform state push`?

Answer:

- `pull`: Download remote state (backup/inspection)
- `push`: Upload local state (emergency recovery only)

Q31: How do you split monolithic state files?

Answer:

1. Use modules with separate backends
2. `terraform state mv` to migrate resources
3. Update remote state references
4. Gradual migration with `-target`

Q32: What happens if state file is lost?

Answer: Terraform loses resource tracking. Options:

- Restore from backup
- Use `terraform import` for each resource
- Prevention: versioned remote backends

Q33: Explain workspace state isolation.

Answer: Each workspace has separate state file. Same code, different states (dev/staging/prod).

```bash
terraform workspace new prod
```

Q34: How do you handle sensitive data in state?

Answer:

- Mark variables as `sensitive = true`
- Use remote backends with encryption
- Restrict state file access
- Consider external secret management

Q35: What is state file versioning and why is it important?

Answer: Backend stores multiple state versions. Allows rollback to previous known-good state. Essential for disaster recovery.

Modules & Code Organization

Q36: What makes a good Terraform module?

Answer:

- Single responsibility
- Clear inputs/outputs
- Documentation
- Versioning
- Minimal dependencies
- Reusability

Q37: How do you version Terraform modules?

Answer: Git tags with semantic versioning:

```hcl
module "vpc" {
  source = "git::https://github.com/org/modules.git//vpc?ref=v1.2.0"
}
```

Q38: Explain module sources: local vs remote.

Answer:

- **Local**: `./modules/vpc` - development
- **Remote**: Git, Terraform Registry, S3, HTTP - production

Q39: What is the Terraform Registry?

Answer: Public repository of verified modules and providers (registry.terraform.io). Simplifies module sharing.

Q40: How do you pass complex objects to modules?

Answer: Using object variables:

```hcl
variable "vpc_config" {
  type = object({
    cidr    = string
    azs     = list(string)
    private = bool
  })
}
```

Q41: What are module composition patterns?

Answer:

- Nested modules (modules calling modules)
- Wrapper modules (standard configurations)
- Root modules (entry points)

Q42: How do you handle module dependencies?

Answer:

- Implicit via output references
- Explicit `depends_on` in module blocks
- Careful ordering in root module

Q43: Explain mono-repo vs multi-repo for Terraform code.

Answer:

- **Mono-repo**: All infrastructure in one repo (easier refactoring, versioning complexity)
- **Multi-repo**: Separate repos per service/team (isolation, coordination overhead)

Q44: How do you organize Terraform code for multiple environments?

Answer: Three approaches:

1. Workspaces (same code)
2. Directory structure (separate configs)
3. Branches (Git-based)

Best: Directory structure with modules

Q45: What are Terraform module best practices?

Answer:

- Pin provider versions
- Use semantic versioning
- Document variables/outputs
- Include examples
- Add README.md

- CI/CD testing
- Minimize nested modules

Security & Best Practices

Q46: How do you manage secrets in Terraform?

Answer:

- Environment variables (`TF_VAR_*`)
- Vault integration
- AWS Secrets Manager/Parameter Store
- Terraform Cloud sensitive variables
- Never hardcode secrets

Q47: What is `sensitive = true` in variables?

Answer: Prevents Terraform from showing the value in logs/output. Doesn't encrypt in state.

Q48: How do you implement least privilege in Terraform?

Answer:

- Separate IAM roles per environment
- Scoped provider credentials
- Backend access controls
- State encryption
- Audit logging

Q49: Explain Terraform Cloud Sentinel policies.

Answer: Policy-as-code framework for governance:

```hcl
```

```
# Prevent public S3 buckets
rule "s3_bucket_public_access" {
  condition = all s3_buckets as _, bucket {
    bucket.acl != "public-read"
  }
}
```

Q50: How do you implement GitOps with Terraform?

Answer:

- Git as single source of truth
- PR-based workflows
- Automated plan on PR
- Apply only from CI/CD
- Branch protection rules

Q51: What are the security risks of `terraform.tfstate`?

Answer: Contains:

- Passwords in plaintext
- API keys
- Private IPs
- Resource metadata

Mitigate: encryption, access control, remote backend

Q52: How do you implement cost controls in Terraform?

Answer:

- Use `terraform-cost-estimation`
- Infracost tool in CI/CD
- Tag resources for billing
- Set resource limits in modules
- Regular drift detection

Q53: What is Checkov and how is it used with Terraform?

Answer: Static code analysis for security/compliance. Scans IaC for misconfigurations:

```bash
checkov -d . --framework terraform
```

Q54: How do you handle compliance (SOC2, HIPAA) in Terraform?

Answer:

- Policy-as-code (Sentinel, OPA)
- Automated scanning (Checkov, tfsec)
- Audit logging
- Encrypted state
- Access controls
- Documentation

Q55: What is the principle of immutable infrastructure with Terraform?

Answer: Never modify running resources, always replace. Achieved via:

- `create_before_destroy`
- Blue-green deployments
- Avoiding provisioners

Q56: How do you implement MFA for Terraform operations?

Answer:

- AWS MFA for assume role
- Terraform Cloud/Enterprise SSO
- Break-glass procedures
- Audit all changes

Q57: Explain `terraform plan -out=tfplan` security.

Answer: Saves plan to file. Can contain sensitive data. Use:

- Encrypt plan files

- Short TTL
- Delete after apply
- Access controls

Q58: How do you rotate credentials used by Terraform?

Answer:

- Short-lived credentials (assume role)
- Dynamic credentials (Vault)
- Automated rotation policies
- Update provider config
- No long-lived keys

Q59: What is state file encryption at rest?

Answer: S3 backend example:

```hcl
backend "s3" {
  encrypt      = true
  kms_key_id   = "arn:aws:kms:..."
}
```

Q60: How do you implement change approval workflows?

Answer:

- PR-based reviews
- CODEOWNERS file
- Manual approval in CI/CD
- Sentinel policies
- Terraform Cloud runs with approval

Advanced Scenarios

Q61: How do you handle circular dependencies?

Answer:

- Refactor to remove circularity
- Split into separate apply stages
- Use `-target` (last resort)
- Data sources instead of direct references

Q62: Explain zero-downtime deployments with Terraform.

Answer:

- `create_before_destroy = true`
- Blue-green with load balancers
- Rolling updates via count/for_each
- Health checks before old resource deletion

Q63: How do you implement custom providers?

Answer: Using Terraform Plugin SDK (Go):

```go
func Provider() *schema.Provider {
  return &schema.Provider{
    Schema: map[string]*schema.Schema{},
    ResourcesMap: map[string]*schema.Resource{},
  }
}
```

Q64: What are Terraform provider plugins and how do they work?

Answer: Separate binaries that communicate via gRPC. Terraform core orchestrates, providers handle API calls.

Q65: How do you handle API rate limiting?

Answer:

- Parallelism control: `terraform apply -parallelism=1`
- Provider-specific rate limiting configs
- Retry logic in custom providers
- Batch operations

Q66: Explain Terraform's graph command.

Answer: Generates DOT format dependency graph:

```bash
terraform graph | dot -Tpng > graph.png
```

Q67: How do you migrate from one provider version to another?

Answer:

1. Read upgrade guide
2. Update version constraints
3. Run `terraform init -upgrade`
4. Test in non-prod
5. Fix deprecations
6. Roll out gradually

Q68: What is `moved` block (Terraform 1.1+)?

Answer: Refactor resources without destroy/recreate:

```hcl
moved {
  from = aws_instance.old
  to   = aws_instance.new
}
```

Q69: How do you implement canary deployments?

Answer:

- Separate resource sets
- Traffic splitting (load balancer)
- Gradual rollout using count
- Monitoring integration
- Automated rollback

Q70: Explain Terraform's replace-triggered-by.

Answer: Forces replacement when related resource changes:

```hcl
lifecycle {
  replace_triggered_by = [
    aws_iam_role.example.id
  ]
}
```

Q71: How do you handle Terraform in multi-cloud scenarios?

Answer:

- Separate providers in same config
- Abstract common patterns in modules
- Be aware of cloud-specific features
- Avoid tight coupling
- Consider cost/complexity

Q72: What are Terraform tests (1.6+)?

Answer: Built-in testing framework:

```hcl
# tests/example.tftest.hcl
run "verify_bucket" {
  assert {
    condition = aws_s3_bucket.test.versioning[0].enabled == true
    error_message = "Versioning must be enabled"
  }
}
```

Q73: How do you implement disaster recovery for Terraform?

Answer:

- State versioning and backups
- Infrastructure as Code in Git

- Documented runbooks
- Regular restore testing
- Multi-region considerations

Q74: Explain Terraform's experimental features flag.

Answer: Opt-in to pre-release features:

```hcl
terraform {
  experiments = [module_variable_optional_attrs]
}
```

Q75: How do you optimize Terraform performance for large infrastructures?

Answer:

- Split state files
- Use `-target` for specific changes
- Reduce parallelism if hitting limits
- Minimize data source queries
- Cache remote state reads

CI/CD & Automation

Q76: How do you integrate Terraform with Jenkins?

Answer:

```groovy
```

```
pipeline {
  stages {
    stage('Plan') {
      steps {
        sh 'terraform init'
        sh 'terraform plan -out=tfplan'
      }
    }
    stage('Apply') {
      when { branch 'main' }
      steps {
        sh 'terraform apply tfplan'
      }
    }
  }
}
```

Q77: What is Atlantis and how does it work?

Answer: Self-hosted Terraform pull request automation. Comments on PRs with plan output, applies on approval.

Q78: How do you implement GitOps for Terraform?

Answer:

- Git = source of truth
- PR-based workflow
- Automated testing in CI
- Auto-plan on PR
- Manual/auto apply on merge
- Drift detection scheduled

Q79: Explain Terraform Cloud workflow.

Answer:

1. VCS integration
2. Automatic plan on commit
3. Policy checks (Sentinel)
4. Manual approval

5. Apply execution
6. State storage

Q80: How do you handle Terraform in GitHub Actions?

Answer:

```yaml
name: Terraform
on: [push, pull_request]
jobs:
  terraform:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: hashicorp/setup-terraform@v1
      - run: terraform init
      - run: terraform plan
      - run: terraform apply -auto-approve
        if: github.ref == 'refs/heads/main'
```

Q81: What is infrastructure testing with Terraform?

Answer: Multiple layers:

- **Syntax**: `terraform validate`
- **Policy**: Sentinel, OPA
- **Unit**: terraform-compliance, Terratest
- **Integration**: Terratest with real resources
- **Security**: Checkov, tfsec

Q82: How do you implement approval gates in Terraform pipelines?

Answer:

- Manual approval steps in CI/CD
- Terraform Cloud manual approval
- Slack notifications
- JIRA integration
- CODEOWNERS reviews

Q83: Explain drift detection automation.

Answer:

```bash
# Scheduled CI/CD job
terraform plan -detailed-exitcode
# Exit code 2 = drift detected
# Send alert/create ticket
```

Q84: How do you handle Terraform upgrades in CI/CD?

Answer:

- Pin Terraform version in pipelines
- Test new version in sandbox
- Gradual rollout
- Version in container images
- Rollback capability

Q85: What is Terraform Cloud Run Tasks?

Answer: Integrations that run at specific stages:

- Pre-plan: cost estimation
- Post-plan: security scanning
- Pre-apply: change management systems

Troubleshooting & Debugging

Q86: How do you debug Terraform issues?

Answer:

- Enable logging: `TF_LOG=DEBUG`
- Specific logs: `TF_LOG_PATH=./terraform.log`
- Review state: `terraform show`
- Graph visualization
- Provider debug logs

Q87: What does "Error acquiring state lock" mean?

Answer: Another Terraform operation is running or crashed. Solutions:

- Wait for completion
- Force unlock (dangerous): `terraform force-unlock <lock-id>`
- Check for hung processes

Q88: How do you fix "Resource already exists" errors?

Answer:

- Import existing resource: `terraform import`
- Rename in config to match
- Remove from state if duplicate
- Check for naming conflicts

Q89: What is "Error: Provider configuration not present"?

Answer: Module uses provider not configured in root. Solutions:

- Add provider in root
- Pass provider explicitly to module
- Use provider aliases

Q90: How do you handle "context deadline exceeded"?

Answer: Timeouts during API calls. Solutions:

- Increase timeout in resource config
- Check API rate limits
- Network/firewall issues
- Provider configuration

Q91: Explain "Error: Cycle" in Terraform.

Answer: Circular dependency detected. Solutions:

- Refactor dependencies
- Use data sources
- Remove unnecessary `depends_on`

- Split into separate applies

Q92: How do you recover from corrupted state?

Answer:

- Restore from backup: `terraform.tfstate.backup`
- Pull previous version from backend
- Manual state reconstruction (last resort)
- Use `terraform import`

Q93: What does "insufficient IAM permissions" mean?

Answer: Provider credentials lack required permissions. Solutions:

- Review CloudTrail/activity logs
- Check IAM policy
- Use `--debug` for exact API calls
- Least privilege analysis

Q94: How do you troubleshoot slow Terraform operations?

Answer:

- Check parallelism setting
- Review API rate limits
- Analyze state file size
- Network latency
- Enable profiling: `TF_LOG=TRACE`

Q95: What are common causes of failed applies?

Answer:

- Resource limits (quotas)
- Conflicting changes (drift)
- Provider bugs
- Dependency issues
- Timeout problems
- API throttling

Architecture & Design

Q96: Design a multi-region, multi-account AWS setup with Terraform.

Answer:

```
Structure:
├── modules/
│   ├── vpc/
│   ├── eks/
│   └── rds/
├── accounts/
│   ├── prod/
│   │   ├── us-east-1/
│   │   └── eu-west-1/
│   └── dev/
└── shared/

- Assume role for accounts
- Regional provider aliases
- Remote state per region
- Cross-region references via data sources
```

Q97: How would you design a Terraform structure for microservices?

Answer:

```
├── infrastructure/ (shared: VPC, EKS)
├── services/
│   ├── service-a/
│   │   ├── terraform/
│   │   └── k8s/
│   └── service-b/
└── modules/ (reusable)

- Separate state per service
- Shared infrastructure layer
- Service teams own their Terraform
- Remote state references
```

Q98: Explain your approach to Terraform code review.

Answer: Check for:

- Security (hardcoded secrets, public access)
- State management (backend config)
- Module usage (DRY principle)
- Naming conventions
- Documentation
- Testing coverage
- Compliance with policies

Q99: How do you handle Terraform at scale (1000+ resources)?

Answer:

- Split state files by service/team
- Use modules extensively
- Implement strict naming conventions
- Automated policy enforcement
- Dedicated platform team
- Self-service with guardrails
- Comprehensive documentation

Q100: What are your Terraform anti-patterns to avoid?

Answer:

- Hardcoded values
- God modules (too complex)
- No state locking
- Manual state edits
- Provisioners instead of cloud-init
- Overusing `-target`
- No version constraints
- Secrets in state
- No testing
- Tightly coupled modules

Bonus Questions by Company Type

FAANG-Style Questions

- **Meta**: "Design Terraform automation for 100+ teams with different cloud accounts"
- **Amazon**: "How would you implement Terraform for a multi-region disaster recovery setup?"
- **Google**: "Design a Terraform CI/CD pipeline with policy enforcement and cost optimization"
- **Netflix**: "Explain your approach to canary deployments and rollback strategies"
- **Apple**: "How do you ensure security and compliance in Terraform at scale?"

Startup Questions

- "You have 3 engineers and need to set up AWS infrastructure. What's your Terraform strategy?"
- "How do you implement cost controls when resources are growing rapidly?"
- "Design a simple but scalable Terraform structure for a startup"

Cloud-Native/DevOps Companies

- "Implement zero-downtime deployments for a Kubernetes cluster"
- "Design a GitOps workflow for infrastructure"
- "How do you handle secrets rotation in a Terraform-managed environment?"

Study Tips

1. **Practice coding**: Set up real infrastructure in free-tier AWS/GCP
2. **Read official docs**: Terraform documentation is excellent
3. **Join communities**: Reddit r/Terraform, HashiCorp forums
4. **Build projects**: Create modules, contribute to open source
5. **Use Terratest**: Write tests for your modules
6. **Follow best practices**: HashiCorp's style guide
7. **Stay updated**: Follow Terraform releases and changelogs

Common Interview Formats

- **Live coding**: Implement a module or fix broken