Terraform Plan & Apply Failure Runbook Pack

A senior engineer's incident runbook for the ten ways `terraform plan` and `terraform apply` go wrong — and how to recover state without making it worse. DevOps AI ToolKit · devopsaitoolkit.com

Terraform failures fall into a handful of families: your config won't validate, the backend/state is unreachable or locked, a provider won't authenticate, variables are missing or invalid, the graph has a cycle, or a resource operation failed mid-apply and left reality out of sync with state. The golden rule during any incident: **read `plan` before you `apply`, and treat state as production data.** Most damage in Terraform incidents is self-inflicted — a blind `force-unlock`, a `state rm` on the wrong address, or an `apply` that replaced a database because nobody read the `-/+ forces replacement` lines. Work the error family, confirm with a plan, then act narrowly.

> Examples are sanitized (`module.vpc`, `aws_instance.web`, `registry.example.com`). Always run `terraform plan` and read it in full before applying. Back up state before any `state` surgery. Prefer a locked, versioned remote backend and a second engineer on anything destructive.

1. Safe Workflow (the order that prevents incidents)

Run these in sequence. Each gate catches a different class of failure before it reaches real infrastructure.

```
terraform fmt -recursive -check      # style only; -check exits non-zero if unformatted
terraform validate                   # config is internally consistent (needs init first)
terraform plan -out=tfplan           # save the exact plan you reviewed
terraform show tfplan                # re-read it; count adds/changes/destroys
terraform apply tfplan               # apply the SAVED plan, not a fresh one
```

- `validate` checks syntax and internal references — it does **not** call providers or read state. If `validate` fails, fix the config; don't touch state.
- Applying a saved `tfplan` guarantees you apply exactly what you reviewed. `terraform apply` with no plan file re-plans and may do more than you saw.
- Scope risky changes with `-target` to shrink blast radius: `terraform apply -target=module.vpc.aws_subnet.private`. Treat `-target` as a surgical tool, not a habit — it can skip dependency updates and leave state partially converged.
- `terraform plan -detailed-exitcode` returns `0` (no changes), `1` (error), `2` (changes present) — use it in CI to gate.

2. Error Acquiring State Lock

```
Error: Error acquiring the state lock
Lock Info:
  ID:        84e3f1a2-...   Operation: OperationTypeApply
  Who:       ci@runner-7    Created:   2026-07-05 14:02:11 UTC
```

```
# First: is someone (or a CI job) legitimately mid-apply? Check the "Who"/"Created" above.
# DynamoDB-backed S3 lock — inspect the actual lock item before doing anything:
aws dynamodb get-item --table-name terraform-locks \
  --key '{"LockID":{"S":"my-bucket/env/prod/terraform.tfstate-md5"}}'

# Only if you have CONFIRMED no apply is running (stale lock from a crashed run):
terraform force-unlock 84e3f1a2-...        # use the exact Lock ID from the error
```

- `force-unlock` does **not** cancel a running apply — it only removes the lock record. If you unlock while another apply is genuinely in flight, two writers can corrupt state.
- Confirm the holder is dead (crashed CI runner, killed laptop session) before unlocking. Ping the `Who` first.
- Never pass `-force` to `apply`/`destroy` to dodge a lock; fix the lock, don't bypass it.

3. Backend / Remote State Issues

```
Error: Backend initialization required, please run "terraform init"
Error: Failed to get existing workspaces: S3 bucket does not exist
Error: Error loading state: AccessDenied
```

```
terraform init                       # first-time backend setup / provider install
terraform init -reconfigure          # backend config changed; ignore saved backend, don't migrate
terraform init -migrate-state        # move existing state to a NEW backend (copies state)
terraform state pull > backup.tfstate  # ALWAYS back up before backend changes
terraform workspace list             # confirm you're in the intended workspace
```

- `-reconfigure` discards the cached backend settings and re-initializes **without** copying state — use when the backend block changed but you don't want a migration.

- `-migrate-state` copies state from the old backend to the new one; Terraform prompts before overwriting. This is how you move local → S3 or bucket → bucket.
- `AccessDenied` / `bucket does not exist` is IAM or a wrong `bucket`/`key`/`region` in the backend block — not a Terraform bug. Verify credentials with the cloud CLI first.
- Wrong workspace is a classic "why is prod planning to destroy everything" cause. Always confirm with `terraform workspace show`.

4. Provider Config Not Present / Auth Failed / Version Constraints

```
Error: Failed to query available provider packages
Error: no valid credential sources for AWS Provider found
Error: Incompatible provider version / provider registry.example.com/... does not match constraint
```

```
terraform {
  required_version = ">= 1.6.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.40"      # allow 5.40.x, block 6.0 — pin deliberately
    }
  }
}

provider "aws" {
  region = var.region
  # credentials come from the environment / profile, NOT hardcoded
}
```

```
terraform init -upgrade               # re-resolve providers within constraints; rewrite the lock file
terraform providers                   # show the provider tree and where each is required
aws sts get-caller-identity           # prove your credentials actually work before blaming Terraform
```

- "no valid credential sources" is almost always the environment: unset `AWS_PROFILE`, expired SSO token, or a missing assume-role. Confirm with the cloud CLI, then re-run.
- Version constraint conflicts come from `.terraform.lock.hcl` vs `required_providers`. `terraform init -upgrade` re-resolves and updates the lock file; commit the result.
- For multi-region/multi-account, use provider `alias` and pass `providers = { aws = aws.us_east }` into modules — a missing alias raises "provider configuration not present."

5. Variables — Missing / Invalid Value, Precedence, Validation

```
Error: No value for required variable
Error: Invalid value for variable  (failed validation condition)
```

```
variable "instance_type" {
  type        = string
  default     = "t3.micro"
  description = "EC2 size for the web tier"

  validation {
    condition     = can(regex("^t3\\.", var.instance_type))
    error_message = "instance_type must be a t3.* size."
  }
}
```

```
terraform plan -var="region=us-east-1" -var-file=prod.tfvars
export TF_VAR_region=us-east-1        # env-var form of an input variable
terraform console                     # evaluate expressions: > var.instance_type
```

- **Precedence (later wins):** environment `TF_VAR_*` → `terraform.tfvars` / `*.auto.tfvars` (alphabetical) → explicit `-var-file` → explicit `-var` on the command line.
- A failing `validation` block is *your* guardrail doing its job — read the `error_message`, fix the input, don't delete the block.
- Missing required variable in CI usually means an un-passed `-var-file` or a renamed variable. Non-interactive runs need every required var supplied; there's no prompt.

6. Missing Required Argument & Schema Errors

```
Error: Missing required argument: "vpc_id" is required
Error: Unsupported argument: an argument named "tag" is not expected here
Error: Unsupported block type
```

```
terraform validate                    # catches most schema errors offline
terraform providers schema -json | jq '.provider_schemas | keys'   # inspect valid args/blocks
```

- "Missing required argument" / "Unsupported argument" almost always follows a **provider major-version bump** where arguments were renamed, removed, or promoted from argument to block (e.g. `tag` → `tags`, inline block → nested block). Read that provider's upgrade guide.
- "Unsupported block type" means the block name is wrong for this provider version — check the resource's registry docs for the exact schema you're pinned to.
- Fix schema errors in config; they never require state changes. If `validate` passes but `plan` doesn't, the error is data/reference-driven, not schema.

---

7. Unexpected Drift & "forces replacement"

The single most dangerous plan output. Read every symbol:

```
  ~ update in-place
  -/+ destroy and then create replacement   # THIS destroys the resource
  +/- create replacement, then destroy      # create_before_destroy is set
```

```
resource "aws_instance" "web" {
  # ...
  lifecycle {
    create_before_destroy = true            # build the new one before killing the old
    ignore_changes        = [tags["LastScan"]]   # stop drift on externally-managed fields
    prevent_destroy       = true            # hard stop: apply fails rather than destroy
  }
}
```

```
terraform plan | grep -E "forces replacement|must be replaced|destroy"
terraform state show aws_instance.web       # compare live attributes vs config
terraform apply -refresh-only               # reconcile drift into state WITHOUT changing infra
```

- `-/+ forces replacement` means Terraform will **destroy then recreate**. On a database, load balancer, or EBS volume that is data loss — stop and confirm intent.
- A replacement caused only by a resource being *renamed in code* should be a `moved` block, not a destroy/create:

```
moved {
  from = aws_instance.app
  to   = aws_instance.web
}
```

- `ignore_changes` silences drift on attributes some other system mutates. `prevent_destroy` turns an accidental destroy into a failed apply — a good guard on stateful resources.
- Persistent drift with no code change means something outside Terraform edited the resource. Use `-refresh-only` to sync state, then decide whether config or reality is authoritative.

---

8. Cycle / Dependency Errors

```
Error: Cycle: aws_security_group.a, aws_security_group.b
```

```
terraform graph | grep -A2 "aws_security_group"     # inspect the dependency edges
terraform plan                                       # cycles surface at plan time
```

- A cycle means two resources depend on each other (classic case: two security groups that each reference the other's ID in a rule). Break it by extracting the mutual reference into a standalone resource.

```
# Instead of SG a -> SG b -> SG a, use separate rule resources:
resource "aws_security_group_rule" "a_from_b" {
  type                     = "ingress"
  security_group_id        = aws_security_group.a.id
  source_security_group_id = aws_security_group.b.id
  # ...
}
```

- `depends_on` fixes *ordering*, not cycles — adding it to a cyclic pair makes the cycle worse. Only use `depends_on` for hidden dependencies Terraform can't infer.
- Over-broad `depends_on` (module-to-module) can also create surprise cycles; depend on the narrowest resource that actually matters.

---

9. Timeout While Creating + Failed Resource Operations

```
Error: timeout while waiting for state to become 'available'
Error: creating EC2 Instance: ... InsufficientInstanceCapacity
│ ... aws_instance.web will be created  (tainted after failure)
```

```
terraform apply                     # re-running often completes a transient/capacity failure
terraform plan                      # a half-created resource shows as tainted → will be replaced
terraform apply -replace=aws_instance.web    # modern, explicit replacement (preferred)
```

- When a create/provisioner fails mid-apply, Terraform marks the resource **tainted**; the next apply destroys and recreates it. Confirm that recreation is safe.
- `terraform apply -replace=ADDR` is the current way to force one resource to be recreated. The old `terraform taint` / `terraform untaint` still work but are deprecated in favor of `-replace`.
- Real timeouts (RDS, EKS) can be tuned per-resource, but a too-short timeout usually hides a real provisioning problem — check the cloud console/events before raising it:

```
resource "aws_db_instance" "main" {
  # ...
  timeouts {
    create = "40m"
    delete = "60m"
  }
}
```

- Capacity/quota errors (`InsufficientInstanceCapacity`, `LimitExceeded`) are provider-side — retry, change AZ/instance type, or raise the quota. No state surgery needed.

---

10. Import & State Surgery

The riskiest tools in Terraform. **Back up state first, every time.**

```
terraform state pull > backup.$(date +%s).tfstate   # non-negotiable before any state op
terraform state list                                 # see exactly what state manages
terraform state show aws_instance.web                # inspect one resource
```

```
# Preferred: declarative import block (plan/apply reviews it like any change)
import {
  to = aws_instance.web
  id = "i-0abc123def456"
}
```

```
terraform plan -generate-config-out=generated.tf     # scaffold config for the imported resource
terraform import aws_instance.web i-0abc123def456    # imperative form (older, no plan preview)

terraform state mv aws_instance.old aws_instance.web # rename/move WITHIN state (no infra change)
terraform state rm aws_instance.web                  # forget a resource; DOES NOT destroy it
```

- Import brings existing infra **under management** — it never creates anything. After import, run `plan`; a non-empty plan means your config doesn't yet match reality.
- `state mv` is for renames/refactors and moving resources between modules — it changes the address, not the infrastructure. For code-driven renames, a `moved` block is safer because it's reviewed in the plan and committed.
- `state rm` makes Terraform *forget* a resource without destroying it — useful to hand ownership to another config, dangerous if you then re-apply and Terraform recreates a duplicate. Double-check before and after.
- Never hand-edit `terraform.tfstate`. Use the `state` subcommands; they keep the serial/lineage consistent.

Decision Tree

```
terraform plan / apply failed?
 ├─ "Error acquiring the state lock"? ──────── confirm holder is dead → force-unlock <ID> (§2); NEVER while an apply runs
 ├─ "Backend init required" / AccessDenied? ── init / init -reconfigure / -migrate-state (§3); check IAM + workspace
 ├─ credential / provider version error? ───── sts get-caller-identity + init -upgrade (§4)
 ├─ "No value for required variable" / bad var? ── pass -var-file, check precedence, read validation msg (§5)
 ├─ "Missing/Unsupported argument/block"? ──── provider major-bump schema change → validate + upgrade guide (§6)
 ├─ plan shows "-/+ forces replacement"? ───── STOP. Rename? use moved (§7). Real? confirm data loss before apply
 ├─ "Cycle:" in output? ────────────────────── break mutual reference into a standalone rule resource (§8)
 ├─ timeout / tainted / failed create? ─────── retry, then apply -replace=<addr>; tune timeouts if genuine (§9)
 └─ resource exists but not in state? ──────── back up state → import block → plan until clean (§10)
```

Copy/Paste One-Shot Triage

```
echo "== version =="        ; terraform version
echo "== workspace =="      ; terraform workspace show
echo "== fmt drift =="      ; terraform fmt -recursive -check || echo "  (unformatted files above)"
echo "== validate =="       ; terraform validate
echo "== providers =="      ; terraform providers 2>/dev/null | head -20
echo "== creds =="          ; aws sts get-caller-identity 2>&1 | head -3
echo "== state count =="    ; terraform state list 2>/dev/null | wc -l
echo "== plan (read-only) ==" ; terraform plan -input=false -lock=false -detailed-exitcode \
  | grep -E "forces replacement|must be replaced|will be destroyed|Plan:" || echo "  no destructive changes"
```

Incident Notes Template

```
INCIDENT: Terraform plan/apply failure
Started (UTC):          ____   Detected by: ____   Sev: ____
Command:                [ ] plan  [ ] apply  [ ] destroy  [ ] import   Workspace: ____
Error family:           [ ] lock  [ ] backend/state  [ ] provider/auth  [ ] variables
                        [ ] schema  [ ] drift/replace  [ ] cycle  [ ] timeout/tainted  [ ] import
Exact error (first line): ____
Plan summary:           add __  change __  destroy __   (any "forces replacement"? Y/N: ____)
State backup taken:     [ ] yes  path: ____    Lock ID (if unlocked): ____
Action taken:           (init flag / -replace / state mv|rm / var fix + who + when) ____
Result:                 [ ] plan clean @ ____   [ ] applied @ ____   [ ] escalated to ____
Root cause:             ____
Follow-ups:             (pin versions, prevent_destroy, CI var-files, quota, moved blocks) ____
```
