spacelift

EBOOK

Terraform State at Scale

The Three–Stage Maturity Model for Platform Teams at Scale

Contents

From remote state to collaboration to dependency-driven reuse

Infrastructure as code doesn't stay single-player for long.

The moment multiple engineers start shipping changes across multiple environments in parallel, Terraform, OpenTofu, and Terragrunt become multiplayer systems.

Once that happens, state management stops being a backend detail and becomes a coordination problem.

Teams scaling their infrastructure hit the same wall:

- ⚠️ Conflicting applies
- ⚠️ State drift
- ⚠️ Unclear ownership
- ⚠️ Surprise breakage from upstream changes
- ⚠️ No reliable answer to "who changed this and why?"

The problem doesn't just relate to where state is stored; it's about how change is coordinated.

In this guide, we'll walk you through the three real-world stages teams need to advance through to master multiplayer state complexity. We'll explain how you can assess the maturity of your state management and move successfully to the next stage. You'll also find a handy quick-reference checklist with a summary of the stages.

The problem: Multiplayer changes everything

IaC is inherently stateful, so tools like Terraform must keep a persistent record (a "state file") of the infrastructure they have previously provisioned to manage it effectively. Terraform state is the data Terraform uses to map your configuration to the actual resources it manages, telling it what exists and what must change. However, once multiple engineers start interacting with the same pipelines, the state file becomes the central, mutable database of record, and that's when the trouble starts.

What is multiplayer?

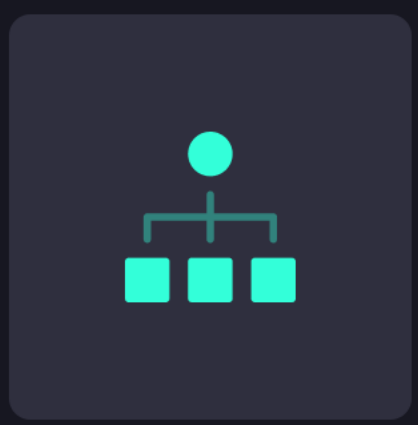
Multiplayer describes the shift that happens when more than one engineer, team, or automated system is interacting with the same infrastructure state at the same time. As with the world of gaming from which the term is borrowed, rules that hold in single-player mode break down when multiple actors are operating concurrently, making moves that can conflict, overwrite, or invalidate each other's work.

What multiplayer state complexity means in IaC

As different users start making changes across different environments, challenges in ownership, mutation control, and trust arise with the state file. If two engineers run `terraform apply` against the same infrastructure, one engineer's changes overwrites the other's, causing unauthorized rollback or dangling resources. Or worse — they don't overwrite them. Suddenly, you have drift, ghost resources, and a creeping fear of running apply at all. At this stage, teams are not concerned about operating models. They are simply looking for stability.

The 3-step adoption pattern is real

To secure that stability, teams might decide to move state files off laptops, add locking, and consider the job done. But remote state is really just the first step in effective state management. At scale, the issue isn't state files: It's coordinating people and systems. You have to align change across teams, manage dependencies across stacks, preserve accountability, and enforce ordering. Here's what it looks like in an organized process:



Stage 1: Remote Backend

Centralizing state management using a remote backend instead of storing state locally resolves the first challenge of multiplayer state complexity — state location. Locking is natively supported, the state file is loaded automatically, and encryption ensures it is secure on disk and in transit.

Stage 2: Collaboration

Providing centralized access, RBAC, visibility and auditability through policy enforcement and standardized workflows addresses the second challenge of multiplayer state management: people.

Stage 3: Dependencies

In the final stage of state management maturity, infrastructure management is coordinated by structure. Modules facilitate reuse, explicit relationships ensure safer change propagation, and first-class stack dependencies minimize blast radius.

⚠️ Teams might be tempted to skip to Stage 3 before completing Stage 2, but resolving the issue of state management processes before you have addressed the problem of how your people interact with those processes is doomed to failure. Without robust access management and clear visibility into who changed what and why, you leave glaring gaps in your security and undermine governance and compliance. Your backend may be stable, but your workflow certainly is not.

What state management maturity buys you

- ✅ Faster parallel work without collisions: Because infrastructure is divided into purposefully scoped stacks with locking enforced at the backend, engineers can work on separate stacks simultaneously without write conflicts or unexpected collisions.
- ✅ Lower risk deployments: The shift from manual to automated and predictable, code-based states reduces the chance of errors, allowing for more confident, faster releases.
- ✅ State versioning and a centralized run record make it straightforward to track when and why infrastructure changed.
- ✅ Compliance: Mature, standardized processes allow for easier auditing and documentation of infrastructure changes.
- ✅ Operational confidence: A mature state architecture (often combined with IaC and CI/CD) enables reliable, repeatable, and scalable operations.

The Maturity Model at a Glance

Stage Summaries: What "Done" Looks Like

Stage 1 — Remote Backend:

State is stored in a remote backend with locking, versioning, and encryption in place. No engineer runs Terraform against a local state file for a shared environment. The backend is treated as the authoritative record of infrastructure, and the team has a documented procedure for recovering from a bad apply or a broken lock.

Stage 2 — Collaboration:

All plans and applies run through a single centralized path. Every engineer authenticates with their own identity, production apply rights are restricted, and a review or approval step is enforced by tooling rather than convention. For any apply in the last 90 days, the team can answer — without digging — who triggered it, what the plan showed, and what the outcome was.

Stage 3 — Dependencies:

Shared infrastructure components are defined as versioned modules and consumed from a registry rather than copy-pasted. Cross-stack dependencies are explicit in code. A bad apply in one stack cannot cascade into an unrelated stack. New environments can be stood up by consuming existing modules, and large-scale changes can be rolled out in a deliberate, ordered sequence.

Which stage are you at?

You're probably at Stage 1 if...

- State files live on individual laptops or a shared server, not in a managed remote backend.
- Two engineers have tried to apply at the same time and had to work out whose changes "won".
- You are not sure whether state is encrypted, or whether locking is actually enforced.
- A team member leaving would take infrastructure knowledge with them that isn't recorded anywhere.

You're probably at Stage 2 if...

- State is safely stored remotely, but you couldn't tell someone who ran the last apply without checking Slack.
- Credentials are shared, or individuals have broader permissions than they need.
- The review process for production changes depends on people remembering to ask, not on tooling enforcing it.
- You have had at least one "invisible apply" — a change that appeared in state with no clear origin.

You're probably at Stage 3 if...

- Modules exist but are copy-pasted between environments rather than consumed from a shared source.
- You have had a change to one stack unexpectedly break another.
- Module versions are not pinned, or upgrading a shared module requires manually updating many stacks.
- The dependency structure of your infrastructure lives in someone's head rather than in code.

Key Metrics by Stage

These metrics give you a quantitative signal for where you are and whether your improvements are working. Baseline each one before making changes, and track them over time.

METRIC	STAGE 1 TARGET	STAGE 2 TARGET	STAGE 3 TARGET
Lock contention rate	Zero concurrent write conflicts	Near zero; any contention is visible and resolved quickly	Near zero; stack boundaries mean contention is isolated and rare
Drift frequency	Detected on a schedule; no undetected drift older than 24 hours	Drift alerts fire automatically; all drift is investigated and resolved	Drift is rare by design; module reuse and ordered rollouts prevent most out-of-band changes
Failed apply rate	Declining as backend configuration stabilises	Low; failures surface in the pipeline with full logs and are traceable to a commit	Low; stack isolation means failures are contained and do not cascade
Apply lead time	Not yet measured	Time from commit to production apply is consistent and predictable	Lead time is low even for complex, multi-stack changes due to ordered rollout tooling
Rollback frequency	Possible but manual; relies on state versioning being in place	Infrequent; review gates catch most problematic changes before they reach production	Rare; blast-radius control and phased rollouts limit the impact of any single bad change
Attribution rate	Not yet tracked	100% of applies to shared environments have a recorded identity and commit	100% of applies are attributed, including cross-stack propagation events
Unplanned outages from IaC changes	Baseline being established	Declining; approval gates and run history make root cause analysis faster	Low and stable; dependency isolation prevents a single change from destabilising multiple systems

Stage 1: Remote Backend

The default backend for Terraform is local storage, with the state file stored and managed in the local machine that runs Terraform. The state is stored on the server disk, so it is tied to the lifecycle of that specific machine. During Stage 1 of the state management maturity model, you enable remote backends instead. This introduces enhanced collaboration, security, accessibility, and state locking.

Stage 1 explained: Remote Backend

What changes in Stage 1?

Once you have successfully completed Stage 1:

- [x] Your state will have moved from local storage to a robust remote backend (such as AWS S3, Azure Storage, or GCS) with state locking enabled. This means state is no longer tied to a single machine or operator, concurrent writes are prevented, and the backend itself (not a local `.tfstate` file) is the authoritative record of your infrastructure.
- [x] State will be encrypted at rest and in transit using your backend's server-side encryption and TLS. Because Terraform state can contain sensitive values written out by providers, this ensures that credentials, tokens, and other secrets are not exposed to anyone with raw access to the storage layer.
- [x] Your Terraform code will be version-controlled, so you can roll back safely if required and keep state changes scoped and predictable. With code in version control, every change to your infrastructure configuration carries a commit history, so you can monitor what changed, when, and why, and you can revert cleanly if an apply produces an unexpected result.

Common failure modes

Terraform state failures tend to cluster around a few well-known patterns. Most can be prevented entirely with the right backend configuration, access controls, and team norms.

⚠️ **State corruption, manual edits, and drift spirals:** Because state is a JSON document, anyone with write access to the backend can edit it directly. A misplaced resource address or a truncated write can leave Terraform producing plans that look correct but act on the wrong resources. Drift spirals start smaller (e.g., an instance type adjusted during an incident but never reconciled back into code), but if the team routinely ignores drift warnings, the gap between state and reality widens with each cycle.

⚠️ **Lock contention and broken locks:** Remote backends that support locking prevent concurrent writes by acquiring a lock at the start of each plan or apply and releasing it on completion, but this process introduces its own failure class. Broken locks occur when a Terraform process is interrupted before it can release its lock.

⚠️ **"Apply from laptop" as an operational hazard:** The risk is not the command itself, but everything surrounding it, from selecting wrong workspaces to not committing local changes to mismatched provider credentials.

Recommended patterns

All of the failure modes above are most damaging when state is treated as a mere side effect of running Terraform rather than as critical infrastructure in its own right. The patterns below address that directly:

- **Remote backend primitives:** A well-configured remote backend is the foundation everything else depends on. At a minimum, it should provide versioning to enable rollbacks, encryption to protect state at rest and in transit, locking to prevent concurrent writes, and documented recovery.
- **Minimal pipeline integration:** The goal of pipeline integration is consistency, so every apply to a shared environment should travel the same path, every time, with no exceptions. This establishes a single source of truth for apply.

Stage 1 checklist

Storage and durability

- [ ] All state is stored in a remote backend — no local `.tfstate` files in shared environments.
- [ ] State versioning is enabled so previous versions can be restored after a bad apply.
- [ ] State is encrypted at rest using server-side encryption.
- [ ] State is encrypted in transit (TLS enforced on all backend communication).
- [ ] Access to the state storage bucket or container is restricted by IAM policy.

Locking and concurrency

- [ ] State locking is enabled and enforced for all plans and applies.
- [ ] The locking mechanism is verified (e.g. DynamoDB table confirmed active for S3 backends).
- [ ] A procedure is documented for safely releasing a broken lock.

Operational hygiene

- [ ] All Terraform code is version-controlled.
- [ ] Local applies against shared environments are restricted or explicitly discouraged.
- [ ] A recovery procedure is documented and tested for restoring state from a previous version.

Graduation criteria → Stage 2

You are ready to move to Stage 2 when the following are true:

- **Your state is stored remotely and is consistently locked.** You are using a remote backend such as AWS S3, Azure Storage, or GCS with state locking enabled, and you have separate state files per environment or a logical system to reduce blast radius.
- **You have a controlled, repeatable execution path.** You are managing state through repeatable, policy-controlled processes, rather than manual command-line interface (CLI) steps.

- **The collision rate between engineers is low.** With automated state locking, If a second engineer tries to run an operation while a stack is locked, their run is queued until the first run completes.

CASE STUDY

PAR Retail

STAGE 1 → STAGE 2

PAR Retail's DevOps team of ten supports 42 developers across five teams, managing a large Terraform estate of 30+ environments, each with over a dozen stacks. At that scale, the challenge was coordinating parallel work without collisions, enforcing consistency across all teams, and eliminating the operational drag of managing hundreds of stacks manually.

> "We're not stepping on each other's toes in terms of state. The stability saves us time that cannot easily be measured. It's not that it saves me an hour a week — it saves me from being interrupted by any instability caused by trying to manage Terraform."
>
> **Rick Arroues**
> Senior DevOps Engineer at PAR Retail

Stage Assessment

PAR Retail is operating solidly at **Stage 2**, with clear Stage 3 indicators in their stack management approach. Spacelift provides a centralized run environment with full run history, an ordered run queue, and OPA-enforced policy guardrails. The deliberate use of scoped stacks, bulk stack operations, and local preview without state locking suggests a team that has also begun to address the blast-radius and dependency concerns central to Stage 3.

Checklist Mapping

Stage 1 & 2 — Complete

- [x] State stored remotely with locking enforced
- [x] Run history queryable by the whole team
- [x] Centralized apply path via Spacelift
- [x] Policy guardrails enforced via OPA
- [x] Local preview enables parallel work without state contention
- [x] Ordered run queue with DevOps prioritization

Stage 3 — Partially Evidenced

- [x] Infrastructure split into purposeful, scoped stacks
- [x] Parallel work across 42 developers without state collisions
- [x] Blast radius controlled via selective bulk stack operations

Stage 2: Collaboration

In Stage 2 of the state management maturity model, your aim is to centralize access and make changes observable and governable.

Why Stage 1 isn't enough

Stage 1 of state management maturity solves the problem of state surviving beyond a single machine, but it does not solve the problem of how to coordinate the people who use it.

Remote state prevents corruption, not chaos. A locked remote backend stops two operators from writing simultaneously, but it has no opinion on who those operators are, whether their changes were reviewed, or whether the apply they just ran was intentional. As team size grows,

the backend becomes a shared resource with no governance layer on top of it.

Visibility, review, and "who changed this?" becomes the bottleneck. Without a centralized run environment, answering basic operational questions such as what was the last apply, what did the plan show, which commit triggered it requires trawling through individual machines, shell histories, and chat logs. At scale, this is simply unsustainable, and in an incident it is dangerous.

What does collaboration really mean?

Collaboration in a Terraform context is less about tooling and more about making the apply path clear to everyone on the team.

- ✅ **Centralized runs and a shared operational interface.** All plans and applies for shared environments should execute in a single, visible location, such as a CI pipeline, Atlantis, Spacelift, Terraform Cloud, or equivalent. Operators should be able to see what is running, what ran recently, and what the outcome was, without needing direct backend access.
- ✅ **RBAC and least privilege.** Not everyone who needs to read infrastructure state needs to write it, and not everyone who can write development state should be able to write production state. A mature collaboration setup maps roles to permissions explicitly: read-only for observers, plan-only for contributors, apply rights gated on approval or environment.
- ✅ **Visibility into plan and apply history.** Every run should produce a durable record that can be queried: the triggering commit, the plan diff, the apply result, and any error output. This is what makes post-incident reviews tractable and what allows new team members to understand how infrastructure has evolved.
- ✅ **Audit trail as a first-class artifact.** The audit trail should be baked in from the start. Who triggered the run, who approved it, and what changed should be answerable from the tooling itself, not reconstructed from memory or Slack threads after the fact.

Common failure modes

⚠️ **Invisible applies.** Applies that run from local machines, personal credentials, or ad-hoc scripts leave no trace in the shared operational record. The state changes, but the team has no visibility into what happened or why.

⚠️ **Permission sprawl and shared credentials.** As teams grow, the path of least resistance is often to share a single set of highly-privileged credentials broadly. This eliminates attribution entirely because every apply looks identical regardless of who triggered it, and it means that revoking access for one person requires rotating credentials for everyone.

⚠️ **No consistent review gate for risky changes.** Without a defined review step, the decision about whether a change is safe to apply is made unilaterally by whoever happens to be running it. Tooling should enforce reviews for high-risk operations.

Stage 2 checklist

Access and governance

- [ ] Individual identities are used for all Terraform operations — no shared credentials.
- [ ] RBAC is configured with environment-scoped permissions (dev / staging / production).
- [ ] Apply rights for production are restricted to a CI service principal or an approved operator role.

Visibility and traceability

- [ ] All plans and applies run in a centralized, shared environment.
- [ ] Each run records the triggering commit, plan diff, and apply outcome.
- [ ] Run history can be queried by the whole team without requiring backend access.

Operational controls

- [ ] A review or approval step exists for applies to production.

- [ ] Alerts or notifications fire on apply failure or unexpected drift.
- [ ] A documented process exists for emergency break-glass access, including post-hoc logging.

Graduation Criteria → Stage 3

You are ready to move to Stage 3 when the following are true:

- **Parallel work is safe and observable.** Multiple engineers or teams can work against the same infrastructure concurrently without stepping on each other, and the state of any in-flight operation is visible to the whole team in real time.
- **Changes are attributable and reviewable.** For any apply in the last 90 days, you can answer (without digging) who triggered it, what the plan showed, and who approved it. Changes do not happen outside the shared run environment.
- **Teams trust the system of record.** Operators do not feel the need to verify state by inspecting cloud consoles directly, and the state file is treated as the authoritative description of infrastructure rather than a best-effort approximation of it.

CASE STUDY

Affinity

STAGE 2 → STAGE 3

Affinity's infrastructure team prioritized small workspace workflows and maximum stack segregation, but their existing Terraform platform couldn't keep up. Blocked runs were hard to diagnose, permissions management was cumbersome, and developers had no clear visibility into groupings or run status. Even low-risk changes required individual approval across all five environments. This process was slow, error-prone, and deeply frustrating for a team already divided on whether Terraform was worth the effort.

affinity

"Even our Terraform haters are more willing to work in it without audibly groaning. We've saved a heap of time and annoyance just by using Spacelift policies."

Chris Nantau

Senior Software Engineer at Affinity

Stage Assessment

Affinity entered the maturity journey with Stage 1 basics in place, but it was stalled at the Stage 2 bottleneck. It had S3 for state storage and version-controlled code but no reliable centralized run environment, no automated approval gates, and permissions that required manual overhead on every change. Post-migration, they have advanced firmly into Stage 3. OPA policies automate approval for low-risk changes, eliminating the five-environment manual approval loop. Spacelift's Terraform provider manages all dependency trees, enabling ephemeral environments to be spun up and torn down on a cadence. This is a clear sign of mature blast-radius control and ordered rollout capability. Bulk stack actions and API-driven refactoring workflows signal an estate that is being managed structurally, not manually. Drift detection is actively planned as the next operational investment.

Checklist Mapping

Stage 1 & 2 — Complete

- [x] State stored remotely in Amazon S3, managed via Spacelift
- [x] Centralized apply path — no manual applies to shared environments
- [x] OPA policies enforce automated approval for low-risk changes
- [x] Run visibility and groupings surfaced across all 239 stacks
- [x] Permissions managed via Spacelift — no manual role-handling overhead
- [x] VCS integration triggers runs on commit and PR

Stage 3 — In Progress

- [x] Spacelift Terraform provider manages all dependency trees
- [x] Ephemeral environments spun up and torn down on a cadence
- [x] Bulk stack actions enable structured, efficient rollouts at scale
- [x] API-driven refactoring workflows replace manual intervention
- ⚠️ Drift detection planned but not yet active

Stage 3: Dependencies

By the end of Stage 3 you should have a mature state management model in place that accommodates scaling through explicit dependencies, reuse, and blast-radius control.

The scaling ceiling without dependencies

Stage 2 makes collaboration safe and observable. What it does not solve is what happens when the infrastructure itself becomes too large and interconnected to reason about as a single unit.

Duplication, drift, and inconsistent rollouts. Without a deliberate dependency model, teams copy-paste configurations across environments and stacks. Those copies inevitably diverge: A security group rule is updated in one place; a variable default changes in another. Ultimately, environments that are supposed to be identical are meaningfully different. Rollouts become manual coordination exercises because there is no shared source of truth for what a given component should look like.

Unpredictable change propagation. When relationships between stacks and modules are implicit (encoded in convention, tribal knowledge, or comments), it is easy to change something upstream without understanding what it affects downstream. A modification to a shared networking module might cascade into dozens of stacks. Without an explicit dependency graph, the only way to find out is to run a plan and hope you noticed everything in the diff.

What dependencies unlock

A mature dependency model shifts infrastructure management from coordination by convention to coordination by structure.

- ✅ **Reuse.** Modules become the canonical definition of a component such as a VPC, a Kubernetes cluster, or a standard IAM role set, and every stack that needs that component consumes the module rather than reimplementing it. Changes made to the module propagate to consumers in a controlled, versioned way rather than requiring manual updates in every copy.
- ✅ **Safer change propagation.** Explicit terraform_remote_state references or data sources make

upstream-to-downstream relationships legible in code. When a shared output changes, downstream stacks surface the impact at plan time, not after an apply has already run.

- ✅ **Blast-radius control.** Splitting infrastructure into purposeful stacks with well-defined interfaces means a bad apply can only affect what is within that stack's scope. A mistake in the application layer cannot cascade into the networking layer if they are managed as separate state files with explicit, read-only connections between them.
- ✅ **Ordered rollouts and partial deploys.** With an explicit dependency graph, it becomes possible to apply changes in a deliberate sequence — networking before compute, shared services before consumers — and to deploy to a subset of the estate without triggering unrelated stacks. This makes large-scale changes tractable and reduces the coordination overhead for every release.

Common failure modes

- ⚠️ **Implicit coupling.** The most common form is two stacks that share a resource (a security group, a DNS zone, an S3 bucket) managed in one stack but assumed to exist in another, with the relationship documented nowhere in code. When the owning stack changes, the dependent stack breaks in ways that are hard to trace and harder to prevent.
- ⚠️ **Accidental cascading changes.** Modules that are consumed without version pinning are a frequent source of unintended cascades. Any upstream module change propagates immediately to every consumer on the next plan, potentially triggering large, unexpected diffs across the estate simultaneously.
- ⚠️ **Refactors that break half the estate.** Renaming a module output, changing a resource address, or restructuring a stack's internal layout can silently invalidate the assumptions of every downstream consumer. Without a clear contract between modules and the stacks that use them, refactors are high-risk operations that teams learn to avoid. However, this causes the codebase to calcify over time.

Stage 3 checklist

Explicit relationships

- [ ] All cross-stack dependencies use `terraform_remote_state` or data sources — no out-of-band coordination.
- [ ] Module inputs and outputs are documented and treated as a versioned interface.
- [ ] The dependency graph for critical infrastructure paths is mapped and visible to the team.

Reuse and standardization

- [ ] Shared components (networking, IAM, compute baselines) are defined as modules consumed from a registry or repository, not copied per-environment.
- [ ] Module versions are pinned at consumption sites, with a documented process for upgrading.
- [ ] Environment-to-environment parity is enforced through shared modules, not manual synchronization.

Blast radius and rollout safety

- [ ] Infrastructure is split into stacks with scoped, purposeful state boundaries.
- [ ] A bad apply in one stack cannot directly affect resources in an unrelated stack.
- [ ] Rollout order for changes that span multiple stacks is defined and automated where possible.

"You're Done" Criteria

If you have progressed through all the stages of of state management maturity, here's how you know you have an effective model for dealing with multiplayer state complexity:

- **You can reuse without chaos.** Shared modules are the default path for common components, and updating a module propagates changes in a controlled, reviewable way rather than requiring manual updates across dozens of copies. New environments can be stood up by consuming existing modules, not by duplicating existing configurations.

- **Propagation is intentional.** When something upstream changes, the downstream impact is visible before any apply runs, and the decision to propagate that change is explicit and deliberate. Surprises in plan output are rare, and when they occur they are investigated rather than accepted.
- **Large estates remain manageable.** The size of the infrastructure estate no longer dictates the complexity of any individual change. Stacks are small enough to reason about independently, dependencies are explicit enough to navigate confidently, and the team can onboard new members or new workloads without a weeks-long apprenticeship in undocumented conventions.

CASE STUDY

commercetools

STAGE 3

commercetools is a leading enterprise digital commerce platform operating across multicloud environments (AWS and Google Cloud). Infrastructure automation is central to its SRE processes, but the team was managing Terraform through an entirely manual workflow: Plans were run locally, added to PRs by hand, reviewed by SREs, and applied locally to targeted environments. A single SRE could spend an entire workday on PR queues, with spillover into the next day due to apply failures and retries.

commercetools

"Delivery timelines for developers were not guaranteed because every opened PR fell into a queue to be actioned by an SRE, and, unless it was urgent, this work usually took days."

Chuka Durugo

Sr. SRE at commercetools

Stage Assessment

commercetools is operating at Stage 3. The pre-Spacelift workflow shows a team that had the Stage 1 fundamentals in place but was stuck at a Stage 2 bottleneck: They had version-controlled code and environment-targeted applies, but no centralized run environment, no automated approval gates, and SRE availability as the single point of failure for every deployment.
Post-adoption, the picture is fully Stage 3: A private module registry with semantic versioning enables controlled reuse across the codebase; approval policies enforce team-scoped ownership of stack changes; and developer self-service removes the SRE coordination overhead entirely.
The explicit use of space-based ownership, push policies scoped to project roots, and OIDC-based multicloud authentication all reflect a mature, dependency-aware infrastructure model.

Checklist Mapping

Stage 1 & 2 — Complete

- [x] State managed remotely via Spacelift per stack
- [x] Centralized apply path — no local applies to shared environments
- [x] Plan output surfaced on every PR automatically
- [x] Approval policies enforce review before apply
- [x] Team-scoped access — changes require approval from the owning team
- [x] Individual cloud authentication via private worker pool (AWS) and OIDC (Google Cloud)

Stage 3 — Complete

- [x] Private module registry with semantic versioning
- [x] Module version pinning with automatic patch-version resolution
- [x] Push policies scoped to stack project roots — no unintended cross-stack triggers
- [x] Developer self-service removes SRE as a coordination bottleneck
- [x] Space-based ownership enforces blast-radius control by team

The Assessment: What Stage Are You At?

Answer the questions below honestly, based on how your team operates today (not how you intend to operate). Score each answer using the rubric beneath the questionnaire, then total your points to find your current stage:

Diagnostic Questions

State storage and locking

1. **Where is your Terraform state stored for shared environments?**
    a. Locally on individual machines
    b. In a remote backend (S3, GCS, Azure Storage, etc.)
    c. In a remote backend with versioning explicitly enabled

2. **Is state locking enabled and actively verified for all shared environments?**
    a. No, or I'm not sure
    b. Yes, locking is enabled
    c. Yes, locking is enabled and we have a documented procedure for broken locks

3. **Is your state encrypted at rest and in transit?**
    a. No, or only partially
    b. Yes, server-side encryption and TLS are both in place
    c. Yes, with customer-managed keys and access auditing

Access and credentials

4. **How do engineers authenticate to run Terraform against shared environments?**
    a. Shared credentials or personal long-lived keys
    b. Individual identities, but permissions are inconsistent across environments
    c. Individual identities with environment-scoped RBAC enforced

5. **Who can run `terraform apply` against production?**
    a. Anyone with credentials
    b. A restricted group, enforced by convention
    c. Only a CI service principal; human applies require a documented break-glass process

6. **Can you revoke one engineer's access to Terraform without affecting others?**
    a. No — credentials are shared
    b. Yes, but it requires manual effort across multiple places
    c. Yes, immediately, via IAM or the platform's access controls

Visibility and traceability

7. **If something in production changed unexpectedly yesterday, how would you find out what happened?**
    a. I would ask around or check the cloud console
    b. I could piece it together from Git history and chat logs
    c. I would query the run history in our centralized pipeline tool

8. **Does every apply produce a durable, queryable record of what changed?**
    a. No
    b. Sometimes: It depends on who ran it and how
    c. Yes, always — commit, plan diff, and apply outcome are all recorded

9. **Is there a mandatory review or approval step before applies reach production?**
    a. No
    b. It's expected, but not enforced by tooling
    c. Yes, enforced by the pipeline — applies cannot proceed without approval

Dependencies and reuse

10. **How do shared infrastructure components (networking, IAM, compute baselines) get reused across stacks?**
    a. They are copy-pasted and maintained independently
    b. There are some shared modules, but consumption is inconsistent
    c. Canonical modules are consumed from a registry or repository with pinned versions

11. **When a shared module or upstream stack changes, how do you know what else will be affected?**
    a. We don't; we find out when something breaks
    b. We check manually by tracing references in the code
    c. Cross-stack dependencies are explicit and impact is visible at plan time

12. **Has a change to one stack ever unexpectedly broken another?**
    a. Yes, regularly
    b. Occasionally; we manage it when it happens
    c. Rarely; stacks have well-defined interfaces and state boundaries

13. **Are module versions pinned at the point of consumption?**
    a. No; we reference modules directly from source
    b. Sometimes, inconsistently
    c. Yes, always, with a documented upgrade process

14. **Can you deploy to a subset of your infrastructure estate without triggering unrelated stacks?**
    a. No — changes often have wider blast radius than intended
    b. Sometimes, with careful use of `-target`
    c. Yes — stacks are scoped deliberately and rollout order is defined

15. **Could a new engineer understand the dependency structure of your infrastructure within a day?**
    a. No — it relies on tribal knowledge
    b. With help, yes
    c. Yes — the dependency graph is documented and explicit

Scoring Rubric

Score each answer: **a = 0 points**, **b = 1 point**, **c = 2 points**

TOTAL POINTS	STAGE
0–8	**Stage 1 or below** Focus on remote backend fundamentals before anything else.
9–16	**Stage 2** Your foundation is solid; the gap is collaboration and governance.
17–24	**Stage 3** You are managing dependencies, but look for gaps in reuse and blast-radius control.
25–30	**Stage 3 (mature)** Your state management model is operating as intended; focus on continuous improvement.

Most Valuable Next Upgrade

If you scored 0–8:

Your most impactful immediate action is to move state off local machines and into a remote backend with locking enabled. Everything else in this guide depends on that foundation being in place. Work through the Stage 1 checklist before considering any collaboration or dependency tooling.

If you scored 9–16:

Your state is safe, but your workflow is not yet trustworthy at scale.The single most valuable change you can make is to establish a centralized apply path — a pipeline or orchestration tool that is the only authorised route to production. This alone will give you attribution, review gates, and an audit trail without requiring significant architectural changes.

If you scored 17–24:

Your collaboration model is working, but your stacks are likely more coupled than they appear. The most valuable next step is to audit your cross-stack relationships and make implicit dependencies explicit — either through `terraform_remote_state` outputs or documented data source references. Start with the stacks that have caused the most surprise breakage.

If you scored 25–30:

You have a mature model. The highest-value activity now is investing in developer experience: faster plan feedback, better drift alerting, and lower friction for onboarding new teams and workloads into the existing structure.

Upgrade Playbooks

Stage 1 → Stage 2

The transition from Stage 1 to Stage 2 is fundamentally organisational, not technical. The backend is already in place; what is missing is a governance layer on top of it.

Minimal viable changes

You do not need a sophisticated platform to reach Stage 2. The minimum viable set of changes is:

- **Centralize the apply path.** Pick one tool such as GitHub Actions, Atlantis, Spacelift, Terraform Cloud, or even a simple CI pipeline and commit to it as the only authorized route for applies to shared environments. The specific tool matters less than the consistency.
- **Replace shared credentials with individual identities.** Each engineer should authenticate with their own identity, and the CI pipeline should use a dedicated service principal with the minimum permissions required. This single change makes attribution possible.
- **Add a production approval step.** Even a mechanism as lightweight as a required reviewer on a pull request, a manual approval gate in the pipeline, is dramatically better than none. The goal is to make "did this get reviewed?" answerable with evidence rather than memory.

- **Enable run logging.** Ensure that every plan and apply records the triggering commit and its outcome somewhere queryable by the whole team. Most CI tools do this by default if you route runs through them.

Common traps and anti-patterns

⚠️ **Over-engineering the pipeline before establishing the norm.** Teams sometimes spend weeks building an elaborate, policy-heavy CI system while engineers continue to apply locally in the meantime. Establish the behavioural norm — all applies go through the pipeline — before investing in sophistication.

⚠️ **RBAC that exists on paper but not in practice.** Defining roles without actually restricting credentials means the governance layer is cosmetic. Verify that engineers cannot bypass the pipeline by testing whether a direct `terraform apply` with personal credentials actually works against production.

⚠️ **Treating the audit trail as optional.** Run logs that are ephemeral, inaccessible to the whole team, or only produced for some environments provide false confidence. If you cannot answer "who applied what yesterday?" from the tooling alone, the audit trail is not yet working.

⚠️ **Skipping drift alerting.** Stage 2 is the right time to introduce scheduled drift detection. Teams that do not add it here consistently find themselves diagnosing mysterious state divergence in Stage 3, when the dependency surface is much larger.

Stage 2 → Stage 3

The transition from Stage 2 to Stage 3 is an architectural one. The challenge is not adding tooling but restructuring how stacks and modules relate to each other — without breaking a live, collaborative estate in the process.

Mapping domains and ownership

Before drawing dependency graphs, establish ownership boundaries. Group your infrastructure into logical domains — networking, identity, data, compute, application — and identify which team or squad owns each. Ownership and state boundaries should align: If two teams routinely modify the same stack, that is a signal the stack should be split.

Document the current state honestly, including the implicit dependencies that exist only in convention or tribal knowledge. An informal dependency map — even a whiteboard diagram — is a useful forcing function for surfacing assumptions that nobody has written down.

Designing dependency graphs without over-coupling

The goal is explicit dependencies, not maximum dependencies. A common mistake is to wire every stack to every other via `terraform_remote_state`, creating a graph so dense that a change anywhere ripples everywhere. Instead:

- Identify the genuine consumers of each output. If only one downstream stack reads a value, the dependency is real. If no downstream stack reads it, do not expose it.
- Prefer data sources over remote state references where the relationship is read-only and the producing stack is stable. Data sources are less brittle when the producing stack's structure changes.
- Define clear interface contracts for shared modules: documented inputs, documented outputs, a version number. Treat breaking changes to that interface the same way you would treat a breaking API change — with a deprecation period and communication to consumers.
- Keep stacks small enough to reason about independently. If a stack's plan output regularly runs to hundreds of resources, it is a candidate for further decomposition.

Introducing reuse safely: versioning and rollout controls

Introducing shared modules into an estate that has relied on copy-paste is high-risk if done abruptly. A safer approach:

1. **Extract, don't rewrite.** Take the most stable, most-copied configuration and extract it into a module without changing its behavior. Validate the module against one existing stack before propagating it.

2. **Pin from day one.** Every consumer of the module should reference a specific version tag, not a branch or a floating source reference. This ensures that a change to the module does not cascade to all consumers simultaneously.

3. **Use a phased rollout.** When releasing a new module version, update consumers one environment at a time — development first, then staging, then production — with a plan review at each stage before applying.

4. **Preserve the old path temporarily.** During migration, keep the copy-pasted configuration in place alongside the module reference until the module is validated. Removing the fallback too early turns a controlled migration into a high-stakes cutover.

5. **Automate dependency updates with review gates.** Tools like Renovate or Dependabot can open pull requests when a new module version is available, making upgrade decisions visible and deliberate rather than forgotten until something breaks.

Stage 1 checklist

Storage and durability

- [ ] All state is stored in a remote backend — no local `.tfstate` files in shared environments.
- [ ] State versioning is enabled so previous versions can be restored after a bad apply.
- [ ] State is encrypted at rest using server-side encryption.
- [ ] State is encrypted in transit (TLS enforced on all backend communication).
- [ ] Access to the state storage bucket or container is restricted by IAM policy.

Locking and concurrency

- [ ] State locking is enabled and enforced for all plans and applies.
- [ ] The locking mechanism is verified (e.g. DynamoDB table confirmed active for S3 backends).
- [ ] A procedure is documented for safely releasing a broken lock.

Operational hygiene

- [ ] All Terraform code is version-controlled.
- [ ] Local applies against shared environments are restricted or explicitly discouraged.
- [ ] A recovery procedure is documented and tested for restoring state from a previous version.

Stage 2 checklist

Access and governance

- [ ] Individual identities are used for all Terraform operations — no shared credentials.
- [ ] RBAC is configured with environment-scoped permissions (dev / staging / production).
- [ ] Apply rights for production are restricted to a CI service principal or an approved operator role.

Visibility and traceability

- [ ] All plans and applies run in a centralized, shared environment.
- [ ] Each run records the triggering commit, plan diff, and apply outcome.
- [ ] Run history can be queried by the whole team without requiring backend access.

Operational controls

- [ ] A review or approval step exists for applies to production.
- [ ] Alerts or notifications fire on apply failure or unexpected drift.
- [ ] A documented process exists for emergency break-glass access, including post-hoc logging.

Stage 3 checklist

Explicit relationships

- [ ] All cross-stack dependencies use `terraform_remote_state` or data sources — no out-of-band coordination.
- [ ] Module inputs and outputs are documented and treated as a versioned interface.
- [ ] The dependency graph for critical infrastructure paths is mapped and visible to the team.

Reuse and standardization

- [ ] Shared components (networking, IAM, compute baselines) are defined as modules consumed from a registry or repository, not copied per-environment.
- [ ] Module versions are pinned at consumption sites, with a documented process for upgrading.
- [ ] Environment-to-environment parity is enforced through shared modules, not manual synchronization.

Blast radius and rollout safety

- [ ] Infrastructure is split into stacks with scoped, purposeful state boundaries.
- [ ] A bad apply in one stack cannot directly affect resources in an unrelated stack.
- [ ] Rollout order for changes that span multiple stacks is defined and automated where possible.

Glossary

Apply / `terraform apply`

The Terraform command that executes a planned set of changes against real infrastructure. An apply reads the current state, computes the diff against the desired configuration, and makes the necessary API calls to reconcile the two.

Atlantis

An open-source tool that runs Terraform plan and apply operations via pull request automation, providing a centralized, auditable apply path without requiring a full SaaS platform.

Audit trail

A durable, queryable record of who triggered a Terraform run, what the plan showed, who approved it, and what the outcome was. Treated in this guide as a first-class operational requirement rather than a side effect.

Backend

The storage layer Terraform uses to persist state. The default backend is local (a `.tfstate` file on disk); remote backends include AWS S3, Azure Storage, Google Cloud Storage, and Terraform Cloud.

Blast radius

The scope of infrastructure that can be affected by a single bad apply. Reducing blast radius — typically by splitting infrastructure into smaller, purposefully scoped stacks — limits how much damage any one mistake can cause.

Break-glass procedure

A documented, audited process for emergency access that bypasses normal controls. In the context of this guide, it refers to the steps an operator follows when a manual apply or force-unlock is genuinely necessary outside the standard pipeline.

Drift / state drift

The condition that arises when real infrastructure diverges from what Terraform state records — typically because a change was made out-of-band (e.g. directly in a cloud console) and never reconciled back into code.

Force-unlock / `terraform force-unlock`

A Terraform command that manually releases a stuck state lock. Should only be used after confirming the original locking process is no longer running, as releasing a live lock can allow concurrent writes and corrupt state.

IAM (Identity and Access Management)

The access control framework used by cloud providers (AWS, GCP, Azure) to define who can perform which actions on which resources. In this guide, IAM policies are used to restrict who can read or write Terraform state.

Infrastructure as Code (IaC)

The practice of defining and managing infrastructure through machine-readable configuration files rather than manual processes. Terraform, OpenTofu, and Terragrunt are all IaC tools.

Lock / state lock

A mechanism that prevents two Terraform processes from writing to the same state file simultaneously. A lock is acquired at the start of a plan or apply and released on completion.

Module

A reusable, self-contained unit of Terraform configuration that encapsulates a set of related resources. In Stage 3, modules become the canonical definition of shared infrastructure components, consumed by stacks rather than copy-pasted.

OpenTofu

An open-source fork of Terraform, maintained by the Linux Foundation. The state management concepts in this guide apply equally to OpenTofu and Terraform.

Pipeline

An automated CI/CD workflow that executes Terraform plans and applies. In this guide, the pipeline is the single authorised apply path for shared environments, providing consistency, attribution, and an audit trail.

Plan / `terraform plan`

The Terraform command that computes the difference between current state and desired configuration and outputs a preview of what changes would be made. A plan does not modify infrastructure; it produces a diff for review before an apply.

RBAC (Role-Based Access Control)

A permissions model that assigns access rights based on defined roles rather than individual identities. In a mature Terraform setup, RBAC maps roles (observer, contributor, approver) to environment-scoped permissions.

Remote state

State stored in a shared, remote backend rather than on a local machine. Remote state is accessible to multiple operators and pipelines, and typically supports locking and versioning.

`terraform_remote_state`

A Terraform data source that allows one stack to read the outputs of another stack's state file. Used in Stage 3 to make cross-stack dependencies explicit and legible in code.

Serial counter

A monotonically increasing number embedded in the Terraform state file that tracks the version of the state. A corrupted or mismatched serial can cause Terraform to reject or misapply state.

Stack

A unit of Terraform configuration managed as a single state file. Splitting infrastructure into multiple stacks with well-defined boundaries is a core Stage 3 practice for blast-radius control and parallel work.

State file / `.tfstate`

The JSON document Terraform uses to record the current state of managed infrastructure. It maps configuration resources to real cloud resources and is the source of truth Terraform uses to compute plans.

State versioning

A backend feature that retains previous versions of the state file, enabling rollback to a known-good state after a bad apply or corruption event.

Terragrunt

A thin wrapper around Terraform that adds features for managing multiple stacks, enforcing DRY configuration, and orchestrating dependency ordering across a large infrastructure estate.

TLS (Transport Layer Security)

The cryptographic protocol used to encrypt data in transit. In this guide, TLS ensures that state data is protected when being read from or written to a remote backend.

Workspace

A Terraform feature that allows multiple state files to be managed within a single configuration, typically used to separate environments (development, staging, production). Selecting the wrong workspace before an apply is a common "apply from laptop" hazard.

Resources

Stage 1 — Remote Backend

Core explainer / patterns

Managing Terraform State – Best Practices & Examples

How to Set Up and Manage Terraform Remote State

Terraform Backends – Local and Remote Explained

Terraform State Lock: How It Works & Best Practices

Terraform Remote State with Spacelift

Video: Terraform State Explained and How Spacelift Helps You

Failure-mode playbook + state operations

Terraform State List: How to List Resources in State File

Terraform State Show Command: Showing Resource Details

Terraform State Mv Command: How to Move Resources

Terraform State Rm: Removing a Resource From State File

Terraform State Rollback Guide: Step-by-Step Recovery

How to Migrate Terraform State Between Different Backends

Terraform Force-Unlock Command : Unlocking TF State File

Security & encryption

13 Terraform Security Best Practices (& 8 Common Risks)

Stage 2 — Collaboration

How to Manage Multiple Terraform Environments Efficiently

What are Terraform Workspaces? Overview with Examples

Audit trail & drift

Terraform Drift Detection and Remediation [Guide]

The Compliance Cost of Drift: Why Auditors Don't Trust Your Terraform

Stage 3 — Dependencies

How to Manage Dependencies Between Terraform Resources

Terraform Private Registry: Setup, Publishing & Best Practices

10 Best Practices for Managing Terraform Modules at Scale

Introducing Stack Dependencies v2

Video: Intro to Stack Dependencies

spacelift