SUDOBLARK
Platform Engineering Whitepaper
May 2026

Platform Transformation Patterns: From Bottleneck to Force Multiplier

SUDOBLARK
Benjamin Clark bclark@sudoblark.com

Executive Summary

Infrastructure bottlenecks significantly hinder organisational competitiveness. DORA Research [10] identifies a 1,000-fold disparity in deployment frequency between elite and low-performing organisations. This whitepaper contends that this gap is primarily due to the failure of low performers to treat infrastructure as a first-class software engineering deliverable. Although documented transformation patterns are limited, they are essential. This whitepaper analyses two transformations conducted by Sudoblark: (1) a multi-national enterprise improved from one release every four months to twelve releases per day (see Section 2.1); (2) a startup launched revenue-generating products within 90 days, starting from no infrastructure (see Section 2.2).

This analysis identifies two core technical patterns to support organisational transformation: data-driven Terraform (see Section 3.1) and CI/CD at scale (see Section 3.2). These patterns enable organisations to manage infrastructure changes efficiently and optimise software delivery processes. Both are defined using a Gang of Four (GoF)-style pattern definition structure [5]. The appendix (see Appendix A) illustrates how data-driven Terraform can simplify the management of users and teams within a GitHub Organisation.

Contents

1 The Problem **3**

2 Case Studies **5**
2.1 Case Study A: Enterprise Bottleneck 5
2.2 Case Study B: Starting from zero 7
2.3 Key Insights 8

3 Two Core Patterns **9**
3.1 Pattern 1: Data-Driven Terraform 9
3.1.1 Intent 9
3.1.2 Motivation 9
3.1.3 Applicability 9
3.1.4 Structure 10
3.1.5 Participants 11
3.1.6 Constraints and Tradeoffs 11
3.1.7 Implementation 12
3.2 Pattern 2: CI/CD at Scale 13
3.2.1 Intent 13
3.2.2 Motivation 13
3.2.3 Applicability 13
3.2.4 Structure 14
3.2.5 Participants 18
3.2.6 Constraints and Tradeoffs 19
3.2.7 Implementation 19
3.3 Safe Adoption and Governance 20

4 Conclusion **20**

A Appendix A: Data-Driven Terraform — Variant B (GitHub Management) **22**
A.1 Step 1: Identify Primitives 22
A.2 Step 2: Sketch out the data structure 23
A.3 Step 3: Decide on the variant to use 24
A.4 Step 4: Build your Infrastructure tier 24
A.5 Step 5: Test the Infrastructure-tier 27
A.6 Conclusion 29

1 The Problem

DevOps emerged as a synthesis of decades of Software Engineering innovation. In 2001, the Agile Manifesto [1] presented an alternative to traditional Waterfall approaches, focusing on short development iterations, emphasising adaptability and a human-centric approach to Software. In 2003, the Lean movement [2] transposed lessons from Toyota's manufacturing methods from the 1980s to Software Engineering, focusing on Systems Thinking and waste elimination. In 2009, Flickr stunned the Industry by showcasing 10 deployments a day [3], emphasising the coordination of Ops and Dev teams and the use of automated deployments to achieve it. The rapid deployments enabled by such methodologies placed a strain on critical non-functional requirements (NFRs), such as security and quality assurance, leading to the Rugged Computing movement in 2010 [4]. By 2013, the definitive literature on the subject - the Phoenix Project [7] - was produced. This piece of literature focused on close cooperation between Dev and Ops teams and contrasted this with the archetype of "Brent": the irreplaceable individual upon whom everything depends, whose departure would paralyse productivity. The singular insight which unifies all of these movements is that Infrastructure automation was an essential part of increased delivery velocity, and must be the shared responsibility of both traditional Dev and Ops silos. Whilst a rich ecosystem of tooling arose in this space, most organisations simply adopted the tooling and not the cultural change they represented.

The DevOps Research and Assessment Team (DORA) conduct yearly Industry surveys. Their 2023 survey [10] demonstrated that top performers outperformed their peers by an exponential margin across every metric (See Table 1). They're faster to market, respond to customer feedback more quickly, and recover from failures in minutes. By the time a competitor ships a product, they've already deployed several iterations and/or new products to an audience that knows and trusts them.

DORA Metric	**Elite Performers**	**Low Performers**
Deployment Frequency	1,640 per year	1.5 per year
Lead Time for Changes	$<$ 1 hour	Weeks to months
Time to Restore Service	$<$ 1 hour	1–8 hours
Change Failure Rate	$\sim$15%	$\sim$45%

Table 1: DORA metrics: Elite vs Low performers

DORA metrics

This whitepaper uses the four core DORA metrics from the 2023 State of DevOps Report [10]. More recent research has included deployment rework rate as a new metric and has used failed deployment recovery time in place of time to restore service. These refinements do not change the fundamental delivery performance gap illustrated in Table 1.

The proliferation of tools resulting from earlier cultural innovations has expanded the capabilities of key individuals, such as the archetypal 'Brent,' without addressing the underlying dependency on them. Organisations frequently experience an initial surge in delivery velocity upon adopting DORA metrics, Infrastructure as Code (IaC) tools like Terraform, and Continuous Integration/Continuous Deployment (CI/CD) practices. However, in the absence of foundational cultural change, progress soon stagnates. Often, only a single individual possesses the expertise to interpret Terraform code or develop new pipelines, necessitating their involvement in every project initiation to ensure proper resource provisioning. As demonstrated in Table 1, most organisations have implemented advanced tooling but have not achieved the cultural transformation necessary to fully leverage these innovations and attain market leadership.

The Industry has already acknowledged the failure of the cultural transformation promised by DevOps and has instead converged on a new idea: Platform Engineering [9]. Team Topologies defines a platform team as one that provides internal services to reduce cognitive load for stream-aligned (product) teams, enabling them to deliver value autonomously. Aligned with this, the SPACE framework [14] highlights that productivity includes employee satisfaction, well-being, communication flow, and efficiency in addition to quantitative activity metrics.

By approaching Infrastructure as a Software Engineering problem, rather than an Operational one, we may refactor Terraform implementations that resemble the Big Ball of Mud anti-pattern [6] - haphazardly structured, tightly coupled, and impossible to extend without breaking something - into composable data structures. The same concept may be applied to our CI/CD platforms, our source control platforms, and the environments we deploy code to. By approaching Infrastructure through the lens of Software Engineering, organisations can outperform their competitors by orders of magnitude, translating directly into higher productivity, market share, and profits. Multiple SPACE dimensions are addressed simultaneously; efficiency improves due to reduced friction in deployment processes; satisfaction increases as developers are fully autonomous for their infrastructure needs; and performance improves due to rapid iteration facilitated by infrastructure as a first-class software deliverable.

2 Case Studies

2.1 Case Study A: Enterprise Bottleneck

A global media and entertainment company operated in over 21 countries with real-time broadcast needs. It adopted the squad model [8], organising teams into small, autonomous units, but Infrastructure remained centralised. Each squad included an Infrastructure specialist responsible for resource provisioning and deployments. During sprint planning, application developers refined business requirements into concrete deliverables. Infrastructure specialists did not attend, considering sprint planning solely an application concern. Development teams experimented in isolated sandbox environments, refining solutions before handing work to the Infrastructure specialist. The specialist reverse-engineered the business requirements and architecture choices, then redesigned the solution in Terraform to align with the broader ecosystem.

After several sprints, this process resulted in a monolithic repository that merged application and infrastructure code into a single module. This module, instantiated in each environment, required over 50 inputs and was accompanied by a 10,000-line policy.tf file that defined all IAM roles and policies. As a result, releases took months and sprints frequently stalled, leaving the programme's strategic objectives unfulfilled. Subsequent interviews with technical leads, senior stakeholders, and programme sponsors revealed several key pain points (see Table 2).

Pain Point	**Description**
Monolithic coupling	Single module with over 50+ variables and a 10,000-line policy file. Adding or modifying resources risked cascading failure.
Lack of interface documentation	No documented interface for inputs, optionals, or interdependencies. Required "code archaeology" to understand setup; tracing through 1000s of lines of code, and reconstructing the original intent, to find the information required.
Infrastructure as Ops concern	All changes required Terraform expertise. The Infrastructure specialist was a bottleneck on every change.
Lack of multiplicity or mutability	Difficult to duplicate or amend resources without reverse-engineering the monolithic setup.
Brittle CI/CD	Infrastructure changes bypassed quality gates. Manual, laptop-based deployments from specialists are common.
Tribal knowledge	Undocumented codebase. Understanding requires only direct, long-term empirical knowledge.

Table 2: Case Study A: Technical pain points constraining infrastructure velocity

The implementation resulted in a Big Ball of Mud anti-pattern. Infrastructure, treated only as operations, became a bottleneck isolated from application delivery. Table 2 highlighted two core failures:

1. The Terraform codebase lacked composability.
2. Infrastructure changes lacked quality and safety gates.

Technical leadership from each vertical was engaged to identify the fundamental primitives essential for project delivery. Based on these insights, a reusable, data-driven Terraform solution was developed to enable developer self-service. Each primitive was provided as a versioned git artefact, accessible to developers. Comprehensive, publicly documented interfaces were established for each primitive, specifying requirements, optional parameters, constraints, dependencies, and outputs. Collections were selected as the primary data structure to facilitate multiplicity. A unified index across modules enabled the creation of multiple logically connected resources, abstracted into single primitives for developer consumption. Invariants were enforced at the abstraction level. Developers could provision required infrastructure using straightforward, well-documented data structures. When edge cases or new requirements emerged, new or updated primitives were introduced and adopted through version updates and revised data structures.

With infrastructure established as a first-class software deliverable and developers empowered to address their own infrastructure requirements, CI/CD processes were restructured to align with these practices. Common CI/CD components were centralised into a versioned library of reusable, composable actions, each representing a discrete task within CI/CD pipelines. An internal website was developed to document the interface—including inputs, outputs, and usage instructions—for each action, mirroring the documentation standards applied to versioned Terraform primitives. This approach facilitated the specification and enforcement of best practices for code quality, security scanning, and deployment. Rigorous code quality metrics were applied to infrastructure code, paralleling those used for application code, thereby fostering developer trust and adoption. Deployment processes transitioned from trunk-based (main branch) to version-based (tagged releases), eliminating manual deployments, ensuring auditability and reproducibility, and reducing both deployment and remediation lead times.

The transformation these changes brought to the programme and the business as a whole is both substantial and quantifiable. Table 3 (see overpage) documents the measurable improvements across infrastructure velocity, deployment frequency, and operational reliability.

How we calculated metrics

Table 3 compares Case Study A metrics before and after the platform transformation. Values from 6 months prior to our engagement were sourced from release notes, CI/CD logs, and incident records. After values were taken from the same sources, 2 months following the transformation, with metric definitions unchanged.

Metric	Before	After
Deployment Frequency	1 release every 4 months	12 releases per day
Lead Time for Changes	2–3 weeks	Under 1 hour
Time to Restore Service	1–3 days	5 minutes
Average hotfix turnaround	1 week	15 minutes
Change Failure Rate	66%	5%
Average Build Time	1 hour	5 minutes
Average Deployment Time	2 hours	5 minutes
Builds per Month	300	8000+

Table 3: Case Study A: Infrastructure transformation metrics (measured 2 months post-transformation)

2.2 Case Study B: Starting from zero

A 10-person consultancy had built its business on spreadsheet-based analytics. Consultants received data in Excel, manually analysed patterns, created custom outputs for each client, and presented their findings via PowerPoint or limited Streamlit/Power BI dashboards. Each client engagement required repeated work as outputs were not discrete nor reproducible in a systematic fashion. Their CEO wished to transition to an AI-enabled platform that could ingest structured and unstructured data in any format and provide real-time outputs, both by emulating existing analysis and via modern interfaces such as an AI chat interface. Furthermore, they had a 90-day deadline to launch to secure further funding. However, the business had no in-house technical expertise beyond a bare-bones proof-of-concept created via an AI coding agent. Following the same data-driven Terraform approach outlined in the Enterprise transformation case study (see Section 2.1), we identified the key business needs, broke these down into their technical primitives, and delivered a foundational digital platform to enable the client's future success.

First, the client required a code storage solution. They needed git repositories, standardised quality rulesets, and an efficient process for implementing team-based repository access. A two-tiered Terraform solution was implemented: one repository managed organisational primitives such as rulesets, teams, and memberships, while another established a repository factory pattern with automated team assignments. This approach provided self-service for designated persons: the CTO and IT administrators managed the organisational repository, while other users could only contribute to the repository factory.

Second, the client needed a code deployment environment, so they were set up with a new AWS Organisation. This included management, development, staging, and production accounts, all secured via Single Sign-On (SSO). The AWS infrastructure was split into primitives and assembled with reusable, modular data structures. New accounts, SSO profiles, and Service Control Policies (SCP) were provisioned through these data structures.

Third, the client needed a way to deploy their code. Since GitHub was their source control platform, GitHub Actions was chosen for CI/CD. The AWS Terraform implementation enabled a new OpenID Connect (OIDC) integration with GitHub, providing temporary security credentials for the CI/CD pipeline from the start.

Finally, new operational practices were established to incorporate a data-driven approach. The proof-of-concept was decomposed into a microservice architecture, demonstrating the benefits by deploying the backend API, authentication mechanisms, ETL pipelines, and frontend website as distinct components. Additionally, DNS was migrated from the existing registrar and managed using a data-driven methodology, reducing the lead time for DNS changes from an average of three days to one minute.

The initial platform setup required three weeks of client engagement, with the majority of time allocated to administrative tasks such as securing payment details for AWS Organisation creation. This transformation produced substantial, quantifiable business impacts, as detailed in Table 4.

Following platform delivery, continued collaboration enabled the launch of the first version of the client's Software-as-a-Service (SaaS) product within an additional 30 days (eight weeks from project initiation). The product was subsequently adopted internally to streamline consultant workflows and externally to support sales efforts.

Metric	**Day 0**	**Day 30**
Revenue-generating digital products	0	1
Lead Time for Changes	Indefinite	Half a day
Builds per Month	0	2383

Table 4: Case Study B: Infrastructure and development metrics (measured 30 days after engagement start)

2.3 Key Insights

Despite the contrasting requirements and contexts of the two use cases, both transformation engagements demonstrate a fundamental set of principles.

1. Infrastructure must be treated as a first-class Software deliverable to unlock exponential improvements in delivery velocity.

2. Self-service and governance must be made as frictionless as possible.

These principles now guide all Sudoblark work, and underpin the two core transformation patterns outlined in Section 3.

3 Two Core Patterns

The two principles detailed in Section 2.3 require concrete implementations to be of use, which we describe as two core patterns below. These are declared using the Gang of Four (GoF) pattern definition structure [5], providing readers with rigorous clarity. This structure enables understanding of why the pattern exists and when to apply it, rather than just understanding what it is. Section 3.1 focuses on enabling the infrastructure as a first-class software deliverable. Section 3.2 focuses on reducing governance friction and enabling self-service patterns at scale.

3.1 Pattern 1: Data-Driven Terraform

Pattern Definition

3.1.1 Intent

This pattern decouples the definition of desired infrastructure primitives from their instantiation by expressing desired state as mutable data structures. This enables infrastructure to be provisioned, amended, and audited solely through the creation, editing, and review of data. In turn, this improves delivery velocity for experienced infrastructure specialists and shifts provisioning and amendment of infrastructure downstream to its primary consumers.

3.1.2 Motivation

Traditional Terraform implementations use imperative logic. Multiplicity is achieved via copy/-paste of existing code, amendments to resources require config alterations, and the desired single source of truth is, in effect, unattainable. Over time, these systems coalesce into the Big Ball of Mud anti-pattern [6], delivery velocity stalls, and infrastructure becomes a blocker rather than an enabler for the business.

The data-driven approach inverts these traditional infrastructure paradigms. By applying software engineering fundamentals of multiplicity, extensibility, and contract-driven development, an infrastructure can be transformed into data structures. Terraform implementations become stateless data transformers, enabling individuals and teams to rapidly iterate and deploy the infrastructure required to fulfil their business needs. Infrastructure specialists become custodians of patterns, architectures, and defined interfaces. Everyone else can self-serve their needs, and as a result, the traditional Ops silo becomes a force multiplier for Dev rather than a blocker.

3.1.3 Applicability

Use data-driven Terraform when:

1. You need to repeatedly deploy many versions of the same primitives.
2. Your development teams need to self-service their infrastructure needs.
3. Imperative approaches to your infrastructure have caused your deployment velocity to stall.
4. Knowledge of your infrastructure codebase is centralised in a select few individuals.
5. Your implementation has coalesced into the Big Ball of Mud anti-pattern [6].

Do not use this pattern when:

1. Your infrastructure requirements are trivial, and thus do not need multiplicity or rapid deployment at scale.
2. Your infrastructure team lacks development discipline; data structures require strict schema validation and documentation for effective usage.

3.1.4 Structure

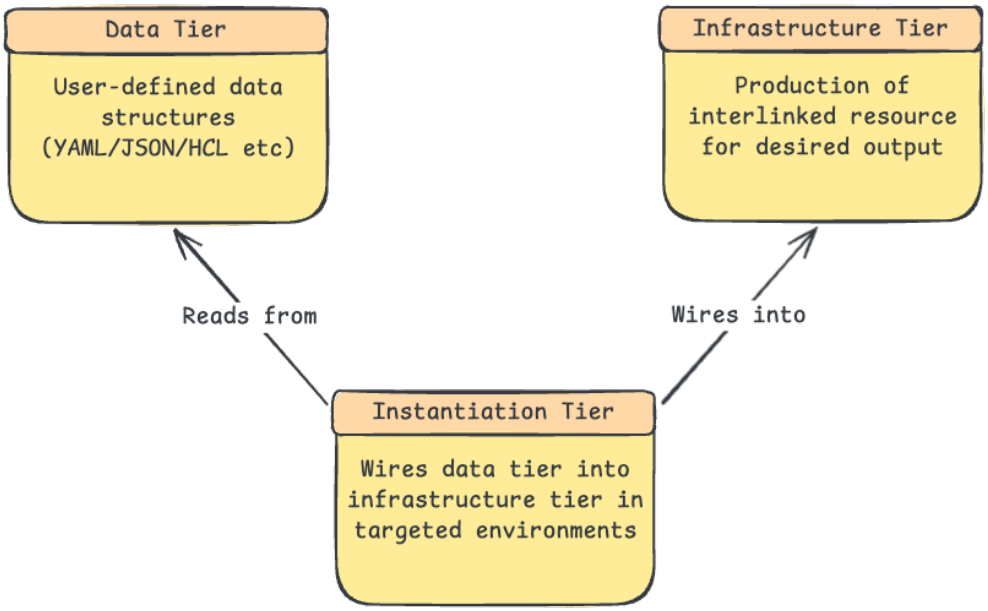


Figure 1: Three-Tier Terraform Pattern

Variant A: Centralised Infrastructure Tier

When an organisation has a large team responsible for Infrastructure, which must be custodians of the platform for all other teams, then a centralised approach (See Section 2.1) works best.

1. The Infrastructure-tier is represented as versioned code artefacts, with each primitive corresponding to one code artefact, which are consumed by multiple other repositories across the business.
2. The Infrastructure-tier interface is usually documented externally, such as on a website or changelog.
3. The Instantiation-tier wires together a local data-tier and a reference to an external, versioned, primitive.

Variant B: Colocated Infrastructure Tier
When an organisation has limited infrastructure resources, uses AI-coding agents heavily, or is relying on non-Infrastructure specialists for Infrastructure creation, then a colocated approach (See Section 2.2) works best.

1. The Infrastructure-tier is stored locally in the same repository within which it is consumed, with each primitive corresponding to a `modules/infrastructure/<primitive>` folder.
2. The Infrastructure-tier interface is documented near the source code, such as docstrings or Markdown files.
3. The Instantiation-tier wires together local Infrastructure and Data tiers.

3.1.5 Participants

- Desired primitives provider. The destination platform, which will provision the required service.
- Infrastructure specialist. An individual with specialist knowledge of the desired primitives and the skillset to create the Infrastructure and Instantiation tiers. Not mutually exclusive with End User.
- End user. An individual with a desire to create desired primitives. Not mutually exclusive with an Infrastructure specialist.
- Data-tier. Mutable collection of desired Infrastructure state, the definitive single source of truth. The expected interface for End User operations.
- Infrastructure-tier. Consumes outputs from the data-tier to create a single desired primitive. Must define a well-known public interface with expected outputs, and enforce its own invariants.
- Instantiation-tier. Wires the data-tier and infrastructure-tier together, instantiating the infrastructure in desired environments.

3.1.6 Constraints and Tradeoffs

Critical Implementation Constraints:

1. Strict schema discipline is required. Data-driven Terraform without validation tooling in your CI/CD platform results in invalid data propagating to the plan stage and surfacing only at runtime.
2. Mature state management is required. Remote state backends with state locking are essential to prevent concurrent modifications from corrupting the state.
3. Learning curve for teams. Infrastructure specialists must learn to think in terms of data structures and algorithms. Variant A (See Section 3.1.4) requires knowledge of semantic versioning. End users must learn conventions for their required primitives.

Variant A (Centralised) Tradeoffs:

1. The core benefit is strong governance control. The platform team controls patterns, ensuring consistency, quality, and standardisation across all implementations.
2. The core cost is the overhead required for module versioning. Semantic versioning of modules, managing changelogs/release notes, and ensuring ongoing maintenance to prevent software ageing [11] can delay the implementation of new features.

Variant B (Colocated) Tradeoffs:

1. The core benefit is that, without the overhead of module versioning, rapid iteration across the entire stack is possible to respond to changing business needs.
2. The core cost is the lack of re-use across repositories. Two different data and infrastructure tiers for the same primitive may exist in two different repositories.

3.1.7 Implementation

1. Identify the infrastructure primitives by the business, and decide how many pieces of infrastructure each primitive abstraction must contain.
2. Sketch out your data structure. Declare required and optional fields. Declare constraints. Decide on invariants.
3. Decide on if your use-case warrants Variant A (see Section 3.1.4) or Variant B (see Section 3.1.4).

   Both variants may be used in a business

   Your core platform team may utilise variant A, provisioning centralised modules for usage by all teams.

   The same team may also use variant B to manage their Datadog account, as there is no concept of multiplicity when managing a single SaaS service, and the primitives created there will never have a need outside that one repository.

4. Build your infrastructure-tier, with one module created per primitive. Within the module, validate your inputs and invariants.
5. Test the infrastructure-tier with a sample data-tier. If possible, wire it using an instantiation-tier that deploys to a sandbox.
6. Once it works, formally document your data structure, invariants, and expected outputs. Include examples. This is the formal contract between the data and infrastructure tiers.
7. Setup your CI/CD validation for the infrastructure-tier. Add schema validators to ensure you don't break the contract.
8. Expose your infrastructure-tier documentation in a manner suitable to your organisation.

Begin the process with one primitive required for one team. Gather feedback, refine the data and infrastructure tiers, and incrementally expand.

3.2 Pattern 2: CI/CD at Scale

3.2.1 Intent

This pattern defines a scalable, maintainable, continuous integration and deployment system using composable components, naming conventions, and automated safety enforcement. This enables three outcomes: (1) reduced friction for non-functional delivery; (2) reduced lead time for pipeline creation; (3) shift-left of ownership from platform teams to developers.

3.2.2 Motivation

Governance, and deployment, at the scales enabled by data-driven Terraform (see Section 3.1) put a strain on non-functionals that Rugged Computing [4] principles address: performance, security and quality. Without established patterns, CI/CD pipelines proliferate organically in an organisation, leading to systemic failures.

1. *Loss of traceability.* Who deployed what, when, and how is ambiguous.
2. *Loss of flexibility.* Without centralised control, enforcement of new or updated NFRs requires manual propagation across the entire estate.
3. *Ineffective resource usage.* Specialist CI/CD engineers spend their time hand-crafting pipelines, rather than building platform capabilities.

This pattern applies the principles outlined in Section 2.3 to CI/CD. Simple name conventions and structure provide clarity. Composable task libraries provide versioned, documented, and discoverable components that allow both central control and teams to self-service their needs. For functionality not natively available in your CI/CD platform of choice, guidance is provided on how to plug gaps using internal CLI tooling without being tightly coupled to the platform. Pipeline specialists transition from propagating hand-made pipelines to provisioning a platform that enables self-service and enforces governance. Everyone else can self-serve their needs, whilst the business ensures enforcement of key NFRs.

3.2.3 Applicability

Use this pattern when:

1. You want to enforce NFRs, via code quality standards, in a uniform manner across the business.
2. You are managing more than 10 pipelines per specialist.
3. Your teams are spending significant time copying and pasting existing workflows, rather than focusing on platform feature delivery.
4. You wish to have a multi-environment rollout across more than 20 repositories.

Do not use this pattern when:

1. Your CI/CD requirements are trivial and do not require centralised management by a specialist team.
2. Your specialist pipeline team lacks development discipline; composable tasks require strict interfaces, documentation, and requirements for effective use.
3. Your workflows are genuinely unique across teams, with no meaningful overlap in requirements or technology. Premature standardisation creates complexity without benefit.

3.2.4 Structure

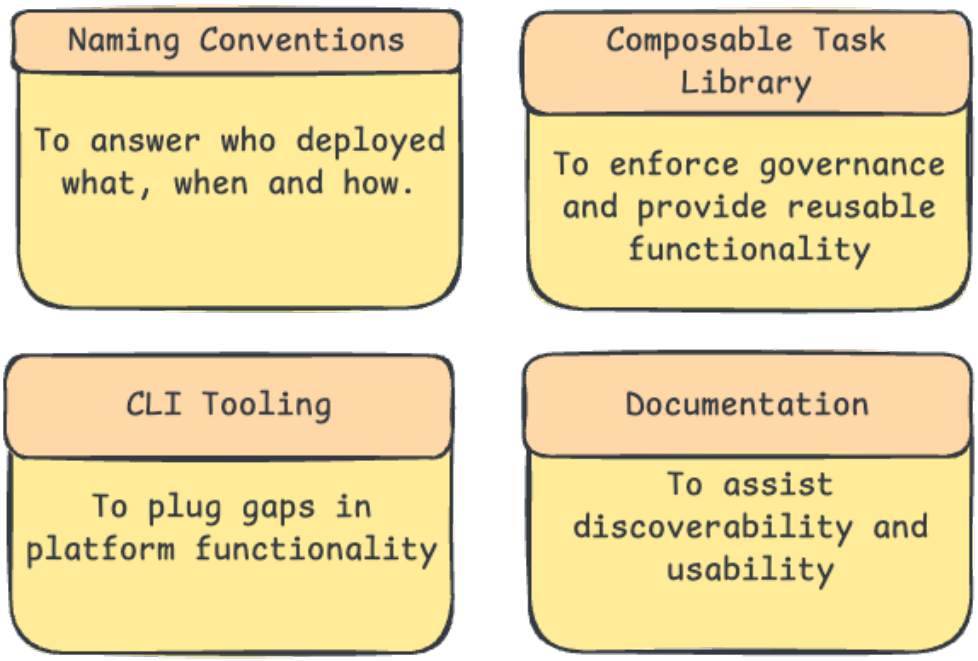


Figure 2: The four components of CI/CD at scale

Naming Conventions

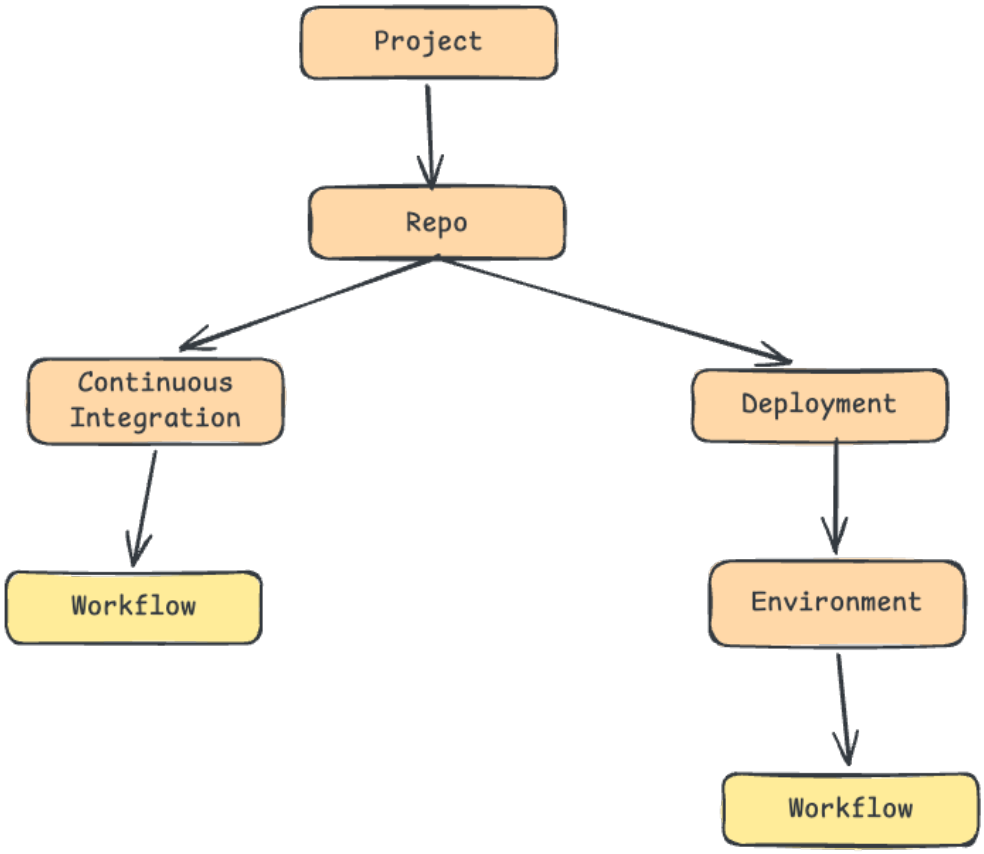


Figure 3: Workflows organised into a project, repository, type, and environment naming hierarchy for repositories with clean separation of concerns

Workflows must be structured such that the following is obvious at all times from naming alone.

1. What project does the workflow belong to
2. Where, in the source code, the workflow lives
3. Whether this is a CI or CD pipeline
4. What environment does the pipeline interact with

Well-structured workflows solve traceability in the short term. In the long term, dashboards and reporting may be built for real-time insights, but these depend on knowing answers to the above, and thus naming conventions are critical for CI/CD at scale.

Monolithic Repositories

The previous structure assumes a clean separation of concerns, where each repository is responsible for a single application in your business. If this is not the case, then you must add the application to the naming hierarchy.

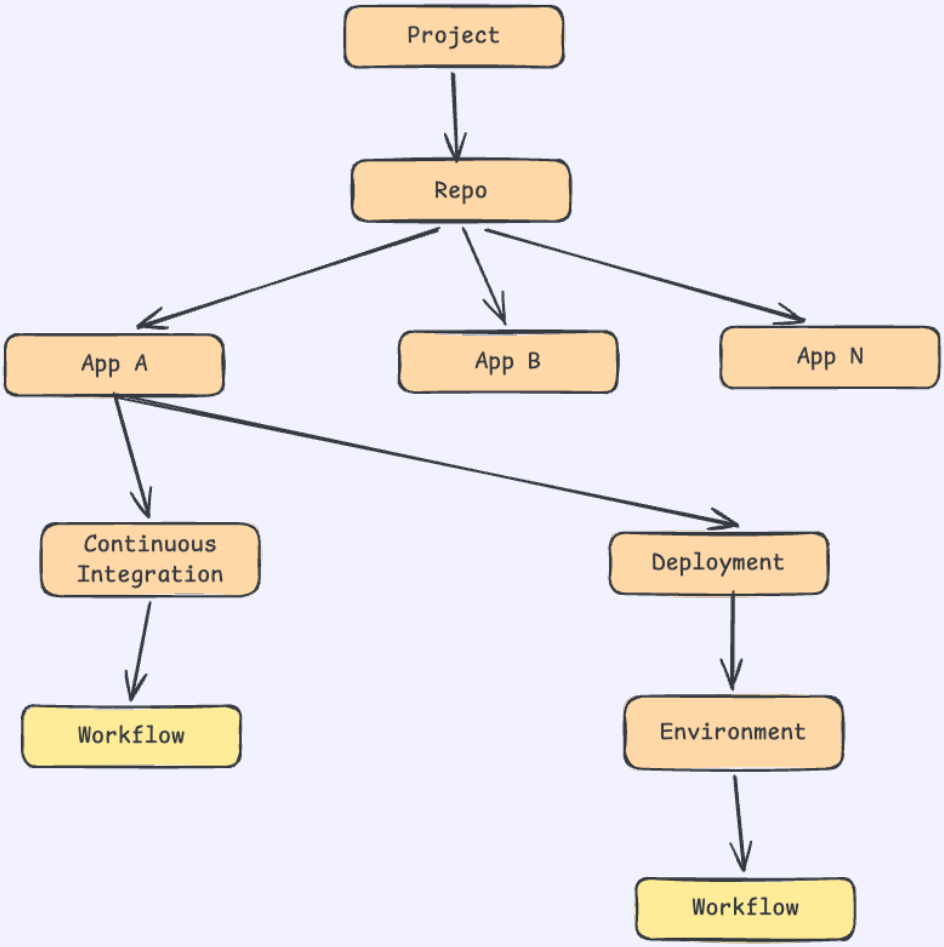


Figure 4: Workflows organised into a project, repository, type, and environment naming hierarchy for repositories with mixed separation of concerns

CLI Tool Usage

Distribution of an internal CLI tool can be achieved by integrating it into your composable task library.

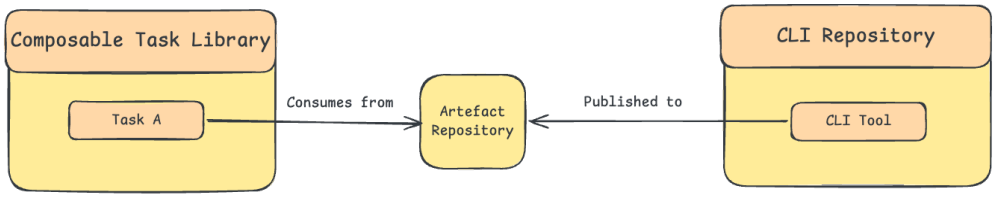


Figure 5: Integration of internal CLI tooling into composable task library

Composable Library Variant A: Workflow Pull Model

1. You maintain a semantically versioned library of composable tasks
2. Downstream repositories orchestrate these tasks into useful workflows
3. In addition, entire workflows may be created as semantically versioned, composable artefacts and utilised by downstream repositories.

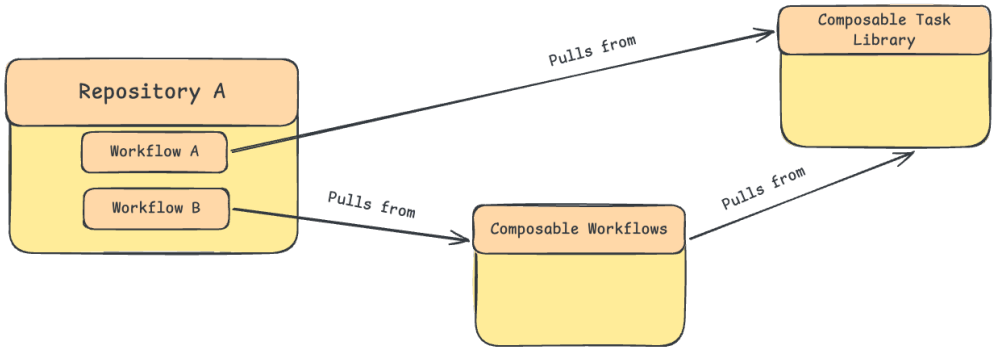


Figure 6: Composable Library Variant A - a pull model for workflows

Composable Library Variant B: Workflow Push Model

1. Workflows are semantically versioned and pushed to downstream repositories.
2. Each workflow reads from a standardised configuration, at runtime, to allow for per-repository customisation.

Interoperability

These variants are not mutually exclusive. Most tasks may use Variant A, but a critical workflow could still orchestrate elements from the Composable task library, which would be managed centrally via Variant B.

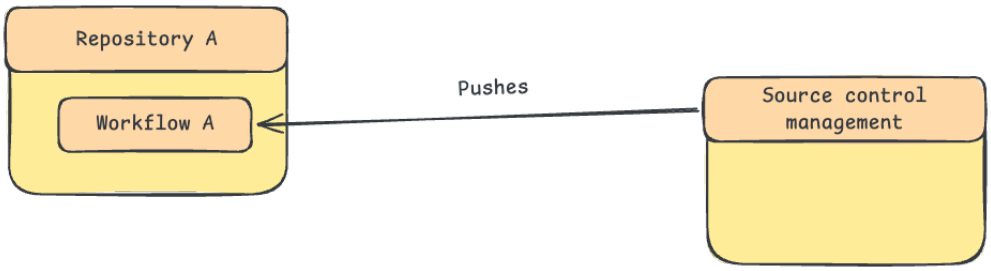


Figure 7: Composable Library Variant B - a push model for workflows

3.2.5 Participants

- CI/CD Platform. Underlying platform for workflow execution.
- Platform primitives. Native functionality composed into tasks can consume a CLI tool.
- CLI tool. Internal tooling to bridge platform gaps.
- Artefact repository. Storage for compiled CLI tooling.
- Composable task library. Semantically versioned library of tasks that fulfil business needs.
- Workflow. Orchestration of tasks and primitives to achieve a means.
- Workflow source. Single source of truth for workflows (Variant B).
- Workflow specialist. Creates tasks and workflows. Not mutually exclusive with Developer or End User.
- Developer. Creates CLI tooling. Not mutually exclusive with Workflow specialist or End User.
- End User. Requires workflows. Not mutually exclusive with a Workflow specialist or Developer.
- Governance/quality standards. Business metrics which require enforcement via the task library.

3.2.6 Constraints and Tradeoffs

Critical Implementation Constraints:

1. Quick turnarounds for new features are required. If components do not fulfil team needs, and the turnaround for new components is too long, standardisation becomes a bottleneck rather than a multiplier.

2. Naming conventions may be constrained by the platform. A majority of platforms allow for pipelines to be organised into hierarchical folders. Of the major providers, GitHub Actions is the only one which requires a flat file structure, thus limiting what may be achieved with naming conventions.

3. Internal tooling creates overhead. The design, implementation and distribution of internal CLI tooling is only worth it if there are significant gaps in off-the-shelf functionality (such as integration with your own APIs).

4. Documentation discipline is required. If your team cannot keep documentation in sync with component changes, the platform's developer experience suffers.

Variant A (pull model) Tradeoffs:

1. The core benefit is standardisation and reduced lead-time for new pipelines. For example, your organisation can enforce required code coverage and scanning in one place. Teams compose pre-made components faster than building from scratch.

2. As with data-driven Terraform, the core cost is the overhead required for module versioning.

3. A secondary core cost is the complexity of organisation-wide changes. If new quality checks need to be enforced, this will require a version bump across all repositories, which in turn may require interface changes. Such changes may be automated, but the implementation is non-trivial.

Variant B (push model) Tradeoffs:

1. The core benefit is lead time for organisation-wide changes. With workflows managed in one place, a new security policy or quality check can be deployed to all repositories at once without much overhead.

2. The core cost is inflexibility. One team's edge case impacts another team's workflow, making the component less reusable across teams. For 50+ repositories, the conditional logic required to handle all edge cases may necessitate migrating to the pull model.

3.2.7 Implementation

This pattern benefits from Kent Beck's extreme programming methodology [12] of:

1. Make it work (get the CI/CD component made locally to unblock teams)

2. Make it right (generalise the component and onboard into the component library)

3. Make it fast (optimise the component once you have real data usage to inform your decisions)

A worked example from Case Study A (see Section 2.1) may help illustrate the point. A team we were working with required containerised Docker builds. First, we built them a basic task in their own repository to build an image (make it work). Second, we generalised the task and added it to the composable library (make it right). Third, we kept the interface but swapped out how images were built to reduce build time by 95% (make it fast).

3.3 Safe Adoption and Governance

Both patterns, combined, prioritise safe adoption and enforcement of internal governance requirements. Data-driven Terraform enforces invariants and constraints, ensuring developers cannot provision Infrastructure outside approved permutations. CI/CD at scale allows enforcement of quality gates - such as static code analysis, linting and code coverage - across the entire business. When combined, the same level of quality assurance normally expected of application code can be applied to infrastructure code, providing an additional layer of safety beyond the invariants enforced in data-driven Terraform.

Migration from existing platforms should be incremental, following Kent Beck's aforementioned extreme programming methodology [12]. Migrate one primitive at a time, rather than all at once, aiming for a piecemeal rollover rather than a big-bang replacement.

4 Conclusion

Over the last 40 years, software engineering has synthesised the ideas of the Agile Manifesto [1], Lean [2], Flickr's continuous deployment practices [3], and Rugged Computing [4] into the DevOps movement. Yet despite the rich tooling around these innovations, most organisations are exponentially outperformed by their peers on key delivery metrics [10] due to a failure to adopt idealised cultural changes. Focus has now shifted from attempting cultural changes to provisioning self-service platforms to improve developer autonomy [9].

The twin principles of treating Infrastructure as a first-class Software deliverable and making governance and self-service as frictionless as possible (see Section 2.3) were used to derive two transformation patterns.

1. Data-driven Terraform (see Section 3.1)
2. CI/CD at scale (see Section 3.2)

The transformative effects of these patterns were explored via two case studies (see Section 2), both of which resulted in exponential increases in key delivery metrics (see Tables 3 and 4) despite differing client constraints and circumstances.

A concrete implementation of data-driven Terraform (see Section 3.1) is provided in Appendix A for reference, showing the application of data-driven Terraform to the problem of managing users and teams in a GitHub Organisation. As noted in Section 1, the idioms they represent are more grounded in Software Engineering than traditional Operations. Most organisations struggle with their implementation; those that succeed create competitive moats allowing them to outperform their peers.

References

[1] Beck, K., et al. (2001). *Manifesto for Agile Software Development*. https://agilemanifesto.org/

[2] Poppendieck, M., & Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*. Addison-Wesley Professional.

[3] Allspaw, J., & Hammond, P. (2009). 10+ Deploys Per Day: Dev and Ops Cooperation at Flickr. O'Reilly Velocity Conference.

[4] McGraw, G., et al. (2010). *Rugged Software Manifesto*. https://ruggedsoftware.org/

[5] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[6] Foote, B., & Yoder, J. (1997). Big Ball of Mud. In *Pattern Languages of Program Design* (Vol. 4).

[7] Kim, G., Behr, K., & Spafford, G. (2013). *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*. IT Revolution Press.

[8] Spotify Engineering. (2014). *Spotify Engineering Culture - Part 1*. https://engineering.atspotify.com/2014/03/spotify-engineering-culture-part-1

[9] Skelton, M., & Pais, M. (2019). *Team Topologies: Organising Business and Technology Teams for Fast Flow*. IT Revolution Press.

[10] Forsgren, N., Humble, J., Kim, G., & Adkins, J. (2023). *Accelerate: State of DevOps Report 2023*. Google Cloud & IT Revolution Press.

[11] Parnas, D. L. (1994). *Software Aging*. In *Proceedings of the 16th International Conference on Software Engineering*.

[12] Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional.

[13] Hashicorp. (2024). *GitHub Terraform Provider Documentation*. https://registry.terraform.io/providers/integrations/github/latest/docs

[14] Forsgren, N., Storey, M.-A., Maddila, C., Zimmermann, T., Houck, B., & Butler, J. (2021). *The SPACE of Developer Productivity*. ACM Queue, Vol. 19, No. 1.

A Appendix A: Data-Driven Terraform — Variant B (GitHub Management)

This appendix demonstrates Data-Driven Variant B (colocated) of the Data-Driven Terraform applied to the GitHub management problem domain. It demonstrates how management of a singleton SaaS product can adopt the pattern to enable administrators to easily manage teams and team membership.

The appendix will describe the implementation following the first five implementation steps outlined for the pattern in 3.1.4.

A.1 Step 1: Identify Primitives

The GitHub Terraform provider [13] offers a wide range of primitives. However, from an operator perspective, there are two primary requirements we need to fulfil.

1. Allow for easy management of the GitHub organisation as a whole
2. Allow for users to self-service creation of repositories

Reviewing the primitives on offer, we can see that each requirement will require multiple provider primitives to fulfil. In practice, the above requirements fall under either the `Administrators` or `End Users` use cases.

Role	Use Case	ID
Administrators	Manage Teams and their members to enforce team-based permissions	A-UC-01
End Users	Create new Repositories to write code for the business	EU-AC-01

Table 5: Differing use-cases for GitHub Management

Conceptually, the path of least resistance for each role is to edit conceptually related resources close together. For example, in A-UC-01, an administrator probably wants to manage what teams a user is in rather than what users are in a team. Therefore, the next step is to map out the primitives that we expect consumers of our module to interact with.

Data Structure	Intended Audience	Purpose
Teams	Administrators	Manage all teams in the Organisation
Users	Administrators	Manage users and their team assignments
Repositories	End Users	Define new repositories for code work

Table 6: Intended data-tier primitives

We'll focus solely on end-to-end implementation for Administrators' use cases in order to keep the appendix focused.

A.2 Step 2: Sketch out the data structure

Having identified the primitives, we can then sketch out a simple data structure for individuals to interact with at the data tier. As we're using variant B, the documentation will live close to the code, so docstrings are acceptable.

Code block 1: Users data structure

```
/*
  Users' data structure definition:

  Each user object requires:
  - username (string): The GitHub username
  - membership (string): The organisation role ("admin" or "member")
  - teams (optional list): Team memberships for the user
    - team (string): The team slug/name (must match a team defined in teams.tf
       )
    - role (string): The role within the team ("maintainer" or "member")

  Constraints:
  - Users with membership = "admin" must have role = "maintainer" in all teams
  - Team values must reference teams defined in the teams local in teams.tf

  Example:
  {
    username   = "github-username"
    membership = "admin"
    teams = [
      {
        team = "team-slug"
        role = "maintainer"
      }
    ]
  }
*/
```

Code block 2: Teams data structure

```
/*
  Teams data structure definition:

  Each team object requires:
  - name (string): The GitHub team name/slug (must be unique, lowercase with
     hyphens)
  - description (string): A description of the team's purpose

  Constraints:
  - Team names must be unique and serve as the reference key for users.tf
     teams list
  - Team names should use lowercase with hyphens (e.g., "team-name")

  Example:
  {
    name        = "team-slug"
    description = "Team description"
  }
*/
```

A.3 Step 3: Decide on the variant to use

As a SaaS product, GitHub inherently has only one version to manage. Therefore, we decided to use a simple `github.organisation` repository - with colocated modules - to manage the organisation as opposed to a centrally managed, semantically versioned, module instantiated once.

GitHub Enterprise

GitHub Enterprise allows for multiple organisations under a single Enterprise. If utilising such a pattern, then a singular `github.organisation` per Organisation is a sensible approach; if discrete sections of your business are unique enough to warrant their own organisations, their needs are probably unique enough that colocated modules provided the required flexibility.

A.4 Step 4: Build your Infrastructure tier

Within the `github.organisation` repository we create the following structure:

Code block 3: GitHub Organisation repository structure at rest

```
github.organisation/
 infrastructure/
    settings/
        data.tf                    # Consumes data tier outputs
        main.tf                    # Provider and backend config
        variables.tf               # Input validation and constraints
        team_membership.tf         # github_team_membership resources
        teams.tf                   # github_team resources
        users.tf                   # github_organization_member resources
```

Infrastructure and Instantiation tiers

With variant B, provided there is only one environment with no scope to change in the future, the infrastructure and instantiation tiers may be colocated. The problem domain we have chosen fits this category (there is only ever one GitHub organisation to manage at a time). In most cases, you'll want to separate out the tiers using the folder structure shown below.

Code block 4: Typical separated tier structure

```
repository/
 modules/
    data/                 # Data tier
    infrastructure/       # Infrastructure tier
        primitive_a/      # Primitive A implementation
            variables.tf
            main.tf
            outputs.tf
 infrastructure/           # Instantiation tier
    environment-a/        # Instantiation for environment-a
    environment-b/        # Instantiation for environment-b
    environment-n/        # Instantiation for environment-n
```

With `primitive-a` instantiated at the appropriate tier as follows.

Code block 5: `infrastructure/environment-a/data.tf` - Consume data-tier

```
module "data" {
  source = "../../modules/data"
}
```

Code block 6: `infrastructure/environment-a/primitive-a.tf` - Wire data and infrastructure tiers together

```
module "primitive_a" {
  source = "../../modules/infrastructure/primitive_a"
  primitives = module.data.primitive_a
}
```

As the primitives we are building are quite simple, we can use a `for_each` loop in our infrastructure tier.

Code block 7: `infrastructure/settings/teams.tf` - Create `github_team` resources

```
resource "github_team" "teams" {
  for_each = { for team in module.data.teams : team.name => team }
  name        = each.value.name
  description = each.value.description
  privacy     = "closed"
}
```

Code block 8: `infrastructure/settings/team_membership.tf` - Flatten and manage team memberships

```
locals {
  team_members = merge([
    for user in module.data.users : {
      for team_config in try(user.teams, []) :
      "${team_config.team}/${user.username}" => {
        team_slug = team_config.team
        username  = user.username
        role      = team_config.role
      }
    }
  ]...)
}

resource "github_team_members" "team_members" {
  for_each = {
    for team in module.data.teams : team.name => team
    if length([
      for member in local.team_members :
      member if member.team_slug == team.name
    ]) > 0
  }

  team_id = github_team.teams[each.key].id

  dynamic "members" {
    for_each = [
      for member in local.team_members :
      member if member.team_slug == each.value.name
    ]
    content {
      username = members.value.username
      role     = members.value.role
    }
  }
}
```

As we are expected to provide a clean interface for the user, we sometimes need to mutate the inputs before they are usable. In the example above, we flatten the `user->teams` object into a single map of `team_members`. Note that user-supplied information - `team` and `user` - is used as a composite primary key. Using user-supplied information in primary keys keeps the abstraction stable and ensures keys are known at compile time, preventing Terraform's dependency graph from breaking.

A.5 Step 5: Test the Infrastructure-tier

We create the data-tier as follows.

Code block 9: Data tier folder structure

```
infrastructure/data/
 teams.tf                  # Team definitions
 users.tf                  # User definitions and team assignments
 output.tf                 # Outputs for consumption by infrastructure tier
```

Code block 10: `infrastructure/data/teams.tf`

```
locals {
  teams = [
    {
      name        = "team-1"
      description = "Team foo"
    },
    {
      name        = "team-2"
      description = "Team bar"
    }
  ]
}
```

Code block 11: `infrastructure/data/users.tf`

```
locals {
  users = [
    {
      username   = "julius"
      membership = "admin"
      teams = [
        { team = "team-1", role = "maintainer" },
        { team = "team-2", role = "maintainer" }
      ]
    },
    {
      username   = "anthony"
      membership = "member"
      teams = [
        { team = "team-1", role = "member" },
      ]
    },
    {
      username   = "longinus"
      membership = "member"
      teams = [
        { team = "team-2", role = "member" },
      ]
    }
  ]
}
```

Code block 12: `infrastructure/data/output.tf`

```
output "users" {
  value = local.users
}

output "teams" {
  value = local.teams
}
```

A simple Terraform plan would show that.

1. `team-1` and `team-2` will be created in the organisation
2. `julius anthony` and `longinus` will be invited to the organisation
3. `julius` will be made an admin of `team-1` and `team-2`
4. anthony will be made a member of `team-1`
5. longinus will be made a member of `team-2`

A.6 Conclusion

This example has been deliberately kept simple, but will work end-to-end for the one use case shown. It shows how the principles of data-driven Terraform can be applied to managing teams and users in a GitHub organisation. Such a pattern compounds. For example, now that teams are in Terraform, it would be relatively simple to swap out `team_membership` with `github_team_sync_group_mapping` and thus shift team membership entirely to your organisation's authentication provider of choice.