RCG Global Services
Ideas. Realized.®

WHITEPAPER

A Blueprint for Terraform Management

Adopting a Global Strategy for Governing Infrastructure-as-Code (IaC)

By Kurt Wysock, RCG Cloud Engineering Practice Leader and
Mark Sontz, RCG Principal Solutions Architect

Introduction

Terraform is an open-source infrastructure as code (IaC) tool that enables users to automate the provisioning, deployment, and management of cloud resources. It was first released in 2014 by HashiCorp, and since then, it has become one of the most popular IaC tools in the industry. Terraform provides a declarative language for describing infrastructure and a set of providers to interact with different cloud service providers. This white paper discusses the adoption and maturity of Terraform in the industry.

Adoption of Terraform

Terraform has gained a lot of popularity in recent years, with many companies and organizations adopting it as their primary tool for infrastructure management. The reasons for this adoption are numerous, but some of the most common include:

1. Multi-cloud: Terraform supports multiple cloud providers, including AWS, Azure, GCP, and many others, making it an attractive choice for companies that use multiple cloud providers.

2. Infrastructure as code: Terraform provides a way to define infrastructure as code, enabling teams to manage infrastructure in the same way they manage application code.

3. Declarative: Terraform is a declarative language. You specify what you want provisioned, not how to provision resources. Every time a piece of terraform code is executed to provision a resource, that resource will be the same, exactly as you specified.

4. Version control: Terraform code can be stored in version control systems like Git, allowing teams to track and audit changes, collaborate, and roll back changes as needed.

5. Automation: Terraform automates the provisioning and deployment of infrastructure, reducing the time and effort required to manage resources manually.

6. Modular design: Terraform has a modular design, allowing teams to reuse code and create a library of infrastructure modules that can be easily shared across different projects.

Maturity of Terraform

Terraform has evolved significantly since its initial release, and its maturity can be evaluated from various perspectives.

1. Features: Terraform now has an extensive feature set, including support for multiple cloud service providers, a rich set of resource types, a powerful language for defining infrastructure, and a growing ecosystem of providers and plugins.

2. Community: Terraform has a large active community of users and contributors. The community provides support through forums, GitHub issues, and other channels, and also contributes to the development of Terraform by creating plugins and modules.

3. Integration: Terraform integrates well with other tools and platforms, including popular CI/CD tools like Jenkins and CircleCI, as well as cloud management platforms like CloudHealth and Turbot.

4. Security: Terraform provides several security features, including support for secrets management, role-based access control, and compliance with industry standards like SOC 2 and HIPAA.

5. Best practices: Terraform has a growing set of best practices and guidelines for using the tool effectively, including recommendations for modular design, version control, testing, and more.

A layered Approach to Managing Terraform Modules

As organizations move towards cloud-native architectures, infrastructure management has become increasingly complex. To manage this complexity, many organizations have adopted Infrastructure as Code (IaC) tools like Terraform. However, managing infrastructure with Terraform requires careful planning and coordination across different teams, including infrastructure, network, security, and application teams. We will discuss the best practices for managing Terraform modules in a layered approach between these teams.

Layered Approach

One of the best practices for managing Terraform modules is to use a layered approach that separates the infrastructure, network, security, and application layers. This approach enables teams to work independently while ensuring that infrastructure changes are made in a controlled and coordinated manner.

The infrastructure layer is responsible for managing the foundational resources required for the cloud environment, such as compute instances and storage. The network layer is responsible for managing the network resources required for the cloud environment, such as Virtual Private Clouds (VPCs), load balancers, subnets, and security groups. The security layer is responsible for managing the security resources required for the cloud environment, such as identity and access management (IAM), security policies, and encryption keys. The application layer is responsible for managing the application-specific resources required for the cloud environment, such as databases, messaging services, and containers.
Each layer should be managed by a separate team with specialized knowledge and expertise in that area. This approach ensures that changes are made in a controlled and coordinated manner, reducing the risk of errors and conflicts.

Module Management

In addition to the layered approach, there are several best practices for managing Terraform modules within each layer.

1. Reusability: Modules should be designed to be reusable across different projects and environments. This approach reduces duplication of effort and enables teams to share best practices and common resources and ensures consistency across the organization.

2. Modularity: Terraform modules should be designed to be modular, with well-defined inputs and outputs. This approach enables teams to build complex infrastructure by combining smaller, reusable modules.

3. Version Control: Modules should be stored in version control systems like Git, allowing teams to track and audit changes, collaborate, and roll back changes as needed.

4. Testing: Modules should be tested thoroughly before being deployed to production environments. This approach reduces the risk of errors and conflicts and ensures that changes are made in a controlled and coordinated manner. Terratest is a great automated testing framework for testing your IaC code.

5. Documentation: Modules should be well documented, with clear descriptions of their inputs, outputs, dependencies and usage examples. This approach reduces the risk of errors and conflicts and enables teams to understand how the infrastructure is designed and implemented.

6. Security: Modules should be designed with security in mind, including support for secrets management, role-based access control, and compliance with industry standards like SOC 2 and HIPAA.

Hardening of Environments using Terraform

As organizations move towards cloud-native architectures, securing their environments has become increasingly important. One way to achieve this is by using Infrastructure as Code (IaC) tools like Terraform to automate the process of hardening an environment. We'll discuss the best practices of hardening an environment using Terraform for use with application development teams.
What is Environment Hardening?

Environment hardening is the process of securing an environment by minimizing its attack surface, reducing vulnerabilities, and ensuring compliance with industry standards and regulations. This process includes securing servers, networks, and applications, as well as ensuring that data is stored securely and access is controlled.

Best Practices for Environment Hardening with Terraform

1. Use Security Best Practices: Terraform modules should incorporate security best practices to ensure that the infrastructure is secure. This includes using secure protocols for communication, setting up firewalls, and ensuring that data is encrypted at rest and in transit.

2. Follow the Principle of Least Privilege: The principle of least privilege is the practice of granting users only the minimum permissions they need to perform their tasks. In Terraform, this can be achieved by using roles and policies to restrict access to resources.

3. Automate Security Configuration: Terraform modules should be designed to automate the configuration of security controls, including user authentication, encryption, and network segmentation. This approach ensures that security controls are consistent across the environment and reduces the risk of human error.

4. Manage Secrets Securely: Terraform should be used to manage secrets securely, such as API keys, passwords, and other sensitive data. Secrets should be stored in secure vaults and accessed using least privilege policies.

5. Conduct Regular Security Audits: Regular security audits should be conducted to ensure that the environment is secure and compliant with industry standards and regulations. Terraform modules can be used to automate security audits and report on vulnerabilities and compliance issues.

6. Use Immutable Infrastructure: Immutable infrastructure is an approach to infrastructure management where infrastructure is treated as disposable and is replaced rather than modified. This approach reduces the risk of configuration drift and ensures that the infrastructure is always in a known, secure state.

Policy as Code

As organizations move towards Infrastructure as Code (IaC) with tools like Terraform, it's essential to ensure that their environment stacks are secure and compliant. One approach to achieving this is by applying policies to code in the Terraform automation process.

What is Policy as Code?

Policy as Code is the practice of defining policies in code, which can be versioned, tested, and enforced automatically. This approach ensures that policies are consistent across the environment and reduces the risk of human error. Policies can include security and compliance requirements, such as ensuring that resources are encrypted, enforcing access controls, and adhering to industry standards and regulations.

Why Apply Policies to Code in the process?

Applying policies to code as a step in the Terraform automation process ensures that the environment stack is complete, secure and in a known state. Policy as Code can be applied at build time (before the Apply step) to ensure what will be provisioned is secure, then at runtime policies can check continuously and check what is provisioned to ensure it remains in compliance.

Organizations can also ensure that their policies are up to date with the latest industry standards and regulations. Policies can be updated and tested alongside the infrastructure code, ensuring that the environment stack remains compliant and secure.

Best Practices for Applying Policies as Code

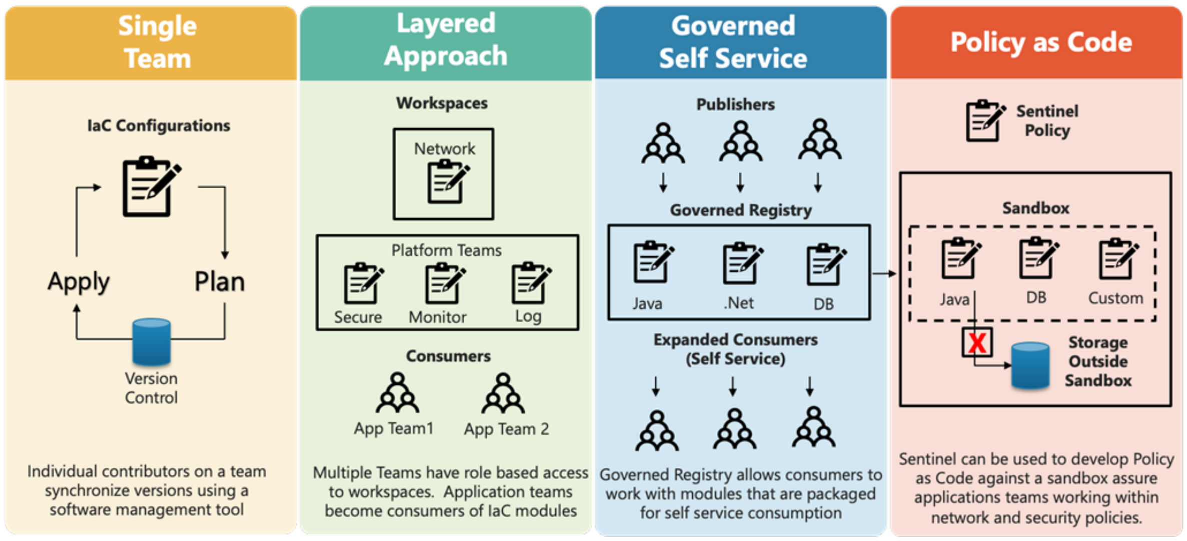
1. Use a Policy Framework: A policy framework can be used to define policies in code and enforce them automatically. Frameworks such as Open Policy Agent (OPA), Bridgecrew and HashiCorp Sentinel can be integrated with Terraform to enforce policies.

2. Define Policies Early: Policies should be defined as early as possible in the Terraform automation process. This approach ensures that policies are considered throughout the infrastructure design process and are tested alongside the infrastructure code.

3. Use Automated Testing: Automated testing can be used to ensure that policies are applied correctly and that the infrastructure stack is compliant. Testing should be automated and integrated into the deployment pipeline to ensure that policies are enforced consistently.

4. Monitor Policy Compliance: Policy compliance should be monitored regularly to ensure that the environment stack remains compliant with industry standards and regulations. Monitoring can be automated, and reports can be generated to highlight any non-compliant resources.

Terraform Adoption, Management, and Governance


Single Team
IaC Configurations
Apply
Plan
Version Control
Individual contributors on a team synchronize versions using a software management tool
Layered Approach
Workspaces
Network
Platform Teams
Secure
Monitor
Log
Consumers
App Team1
App Team 2
Multiple Teams have role based access to workspaces. Application teams become consumers of IaC modules
Governed Self Service
Publishers
Governed Registry
Java
.Net
DB
Expanded Consumers (Self Service)
Governed Registry allows consumers to work with modules that are packaged for self service consumption
Policy as Code
Sentinel Policy
Sandbox
Java
DB
Custom
Storage Outside Sandbox
Sentinel can be used to develop Policy as Code against a sandbox assure applications teams working within network and security policies.

- Terraform deployments usually starts with an individual practitioner who writes a Terraform configuration file (“infrastructure as code”), iterates to make the plan correct, then applies the plan. As needs change, they modify the configuration file and repeat the plan-and-apply process. If there’s a team collaborating to use Terraform, rather than just one individual, the process is basically the same, but they should use some sort of version-control to provide a single source of truth.

- If there are several teams, each responsible for a different part of the infrastructure, Terraform’s config files can be decomposed into separate Workspaces, each of which can have role-based access control.

- In some organizations, there are many users, most of whom are not trained on Terraform, and it wouldn’t be practical to train them all. One common pattern is to have a few publishers and many consumers all working against a central governed registry.

- Organizations can also use Sentinel to define and maintain a sandbox, which polices what consumers can and cannot do (“policy as code”).

Single Team – Individual Contributors

Managing Terraform with a single team using version control involves creating a streamlined process for developing and deploying infrastructure as code. Here are the steps to follow:

1. Choose a version control system: The first step is to choose a version control system (VCS) to manage your Terraform code. Git is a popular choice, but other VCS such as Mercurial, Subversion, or Perforce can also be used. It’s essential to ensure that all team members are familiar with the chosen VCS and can work with it efficiently.

2. Create a Terraform module: Terraform code can quickly become complex, and to ensure consistency, it’s best to create a Terraform module. A module is a reusable component that can be shared across multiple Terraform configurations. The module should be designed to be flexible and customizable, allowing the team to configure the infrastructure according to their needs.

3. Choose a version control system: The first step is to choose a version control system (VCS) to manage your Terraform code. Git is a popular choice, but other VCS such as Mercurial, Subversion, or Perforce can also be used. It's essential to ensure that all team members are familiar with the chosen VCS and can work with it efficiently.

4. Create a Terraform module: Terraform code can quickly become complex, and to ensure consistency, it's best to create a Terraform module. A module is a reusable component that can be shared across multiple Terraform configurations. The module should be designed to be flexible and customizable, allowing the team to configure the infrastructure according to their needs.

5. Use branches: It's essential to use branches to manage changes to the Terraform code. The master branch should contain the stable, production-ready code, while feature branches should be used to develop new functionality or fix bugs. When a feature is complete, it should be merged into development branch, qa branch and eventually the master branch after being reviewed by other team members.

6. Use pull requests: Pull requests are a useful feature of VCS that allows team members to review code changes before they are merged into another branch. Pull requests can be used to discuss changes, request modifications, or approve changes.

7. Use tags: Tags are a useful feature of VCS that allows team members to mark specific versions of the Terraform code. Tags can be used to indicate significant releases or milestones, making it easy to track changes over time.

8. Automate infrastructure deployment: Once the Terraform code is ready to be deployed, it's best to automate the deployment process. Tools like Jenkins, CircleCI, or GitLab CI/CD can be used to automate the deployment process, ensuring that the infrastructure is deployed consistently across environments.

9. Audit and Monitor changes: Continuously audit and monitor changes in your infrastructure, since changes in infrastructure can lead to new security vulnerabilities or service disruptions.

By following these steps, teams can effectively manage Terraform code using version control, ensuring that changes are tracked, reviewed, and deployed consistently across environments.

Layered approach –

Multiple teams manage workspaces based on role

Using Terraform workspaces to manage a layered approach with policies is a best practice for managing complex infrastructure configurations that require multiple environments and policies. Here's how it works:

Layered Approach: The first step is to define a layered approach to your Terraform configuration. This involves breaking down your infrastructure configuration into separate layers based on functionality or environment. For example, you may have a layer for networking, another for compute, and another for application deployment. Each layer should be defined in its own Terraform configuration file. Examples would be a team that manages the network and landing zones in the cloud. Platform teams would segregate infrastructure buildouts and apply security, monitoring, and logging features.

Workspaces: Next, use Terraform workspaces to manage each layer for each environment. A workspace is a named state in Terraform that allows you to manage multiple environments or configurations from a single Terraform configuration file. For example, you might create separate workspaces for development, staging, and production environments.

By using a layered approach with Terraform workspaces, you can ensure that your infrastructure configurations are managed consistently across multiple environments and with proper governance. This approach can also help to reduce the risk of configuration drift and improve the overall maintainability of your infrastructure.

Governed Self Service -

A governed registry allows an expanded consumption model

Using a Governed Registry for Terraform modules allows consumers to work with pre-packaged modules in a self-service manner, while still enforcing governance and control over the modules being used. Here's how it works:

1. Package Modules: The first step is to package the Terraform modules in a way that makes them easy to consume. This involves creating a module package that includes all the necessary code, documentation, examples, and metadata to make it easy for users to understand how to use the module.

2. Store the modules: The modules are then stored in a governed registry, which is a centralized location where approved modules are stored. This can be an internal registry hosted on-premise or cloud-based solutions like Terraform Registry or Artifactory. The governed registry acts as a single source of truth for approved Terraform modules, making it easy for users to find and use the modules they need.

3. Access Control: Access control is implemented on the governed registry to ensure that only approved modules are available for self-service consumption. Access can be granted based on role or group, allowing teams to have granular control over who can access and use the modules.

4. Versioning and Auditing: The governed registry also allows for versioning and auditing of the modules. This means that users can see the history of changes made to a module and track any potential changes that could impact their infrastructure.

5. Continuous Integration and Delivery (CI/CD): The governed registry can be integrated with a CI/CD pipeline to automatically build, test, and deploy Terraform modules. This ensures that modules are always up to date and in sync with the latest changes in the infrastructure environment.

By using a governed registry for Terraform modules, teams can ensure that only approved modules are used, while still allowing for self-service consumption. This approach can help to improve the speed of infrastructure delivery, while reducing the risk of configuration errors and inconsistencies.

Policy as Code – Using Sentinel

Sentinel is a policy as code framework that can be used with Terraform to define and enforce policies for infrastructure configurations. Here's how Sentinel can be used to apply policies as code against a sandbox to ensure application teams are working within network and security policies:

1. Define Policies: The first step is to define the policies that should be enforced. This can be done using Sentinel's policy language, which allows you to write policies in code. Policies can be defined for a wide range of infrastructure configurations, including network and security policies.

2. Create a Sandbox: Once the policies are defined, create a sandbox environment where application teams can work. The sandbox environment should be separate from production environments and configured with network and security policies that align with the policies defined in Sentinel.

3. Integrate Sentinel: Integrate Sentinel with Terraform to enforce policies in the sandbox environment. Sentinel policies can be applied during the Terraform plan and apply process, allowing for real-time enforcement of policies. When a policy violation occurs, Sentinel will prevent the change from being applied and provide feedback to the user on the violation.

4. Test and Refine Policies: Test the policies in the sandbox environment to ensure they are working as expected. Refine the policies as necessary to ensure they are effectively enforcing network and security policies.

5. Rollout Policies to Production: Once the policies have been tested and refined in the sandbox environment, they can be rolled out to production environments. By using Sentinel to enforce policies as code, application teams can be confident that they are working within the network and security policies defined by the organization.

By using Sentinel in Terraform to apply policy as code against a sandbox, application teams can work in a controlled environment that aligns with network and security policies. This approach can help to reduce the risk of configuration errors and security breaches, while also improving the overall efficiency of infrastructure delivery.

Further reading

For a more in-depth look at the strategies discussed in this paper, see the following resources: https://developer.hashicorp.com/terraform/tutorials

About the Authors

Kurt Wysock

Cloud Engineering Practice Leader

Kurt is an accomplished enterprise architect with experience delivering solutions in a broad range of industry verticals. Kurt has led enterprise-wide, multi-year application rationalization and modernization programs that have delivered on both business and IT strategy and goals. As RCG's Cloud Engineering Practice Leader, Kurt evaluates technologies, vendors, and market trends and works with RCG's customers to help deliver application modernization and cloud migration strategies and initiatives.

Mark Sontz

Enterprise Solution Architect

Mark is a seasoned Principal Solutions Architect with over 36 years of experience in architecture, operations, security, and site reliability engineering. He has a proven track record of designing and implementing complex cloud-based solutions for clients across various industries. He is adept at developing cloud migration strategies and has a deep understanding in multi-cloud architectures using AWS, Azure, and Google Cloud. He has experience in implementing security, automation, and observability tools to improve the reliability and performance of cloud-based systems leveraging Infrastructure as Code (IaC). Mark is a contributing member of the Cloud Security Alliance (CSA).