Table of Contents

1. Introduction

2. Project Organization

   - Directory Structure

   - Module Usage

3. Configuration Management

   - Variables and Outputs

   - Resource Naming Conventions

4. Security Considerations

   - Sensitive Data Handling

   - IAM Roles and Policies

5. State Management

   - Remote State

   - State Locking

6. Infrastructure as Code (IaC) Workflow

   - Version Control

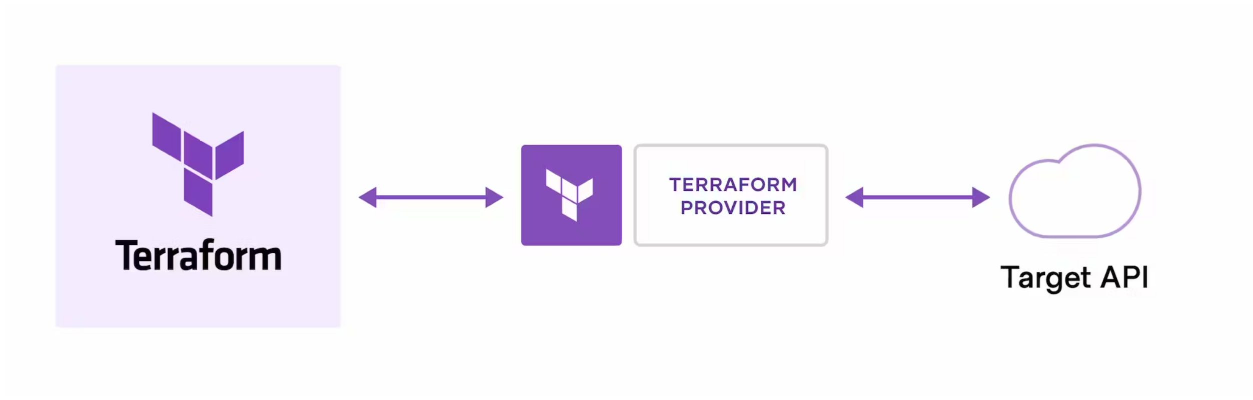
7. General Guidelines

8. Conclusion

9. References

Introduction

Terraform is a powerful Infrastructure as Code (IaC) tool that enables the creation and management of infrastructure across various cloud providers and services. While terraform simplifies infrastructure deployment, adhering to best practices is crucial for maximizing the benefits and minimizing potential challenges associated with managing infrastructure code.



Terraform Key Concepts

In this section, we will describe some key Terraform concepts briefly.

Terraform Configuration Language

Terraform uses its own configuration language to declare infrastructure objects and their associations. The goal of this language is to be declarative and describe the system's state that we want to reach.

Infrastructure as Code (IaC)

Terraform enables you to manage infrastructure using code, allowing for versioning, automation, and collaboration.

Providers

To interact with resources on cloud providers, terraform uses some plugins named providers. This includes cloud providers, SaaS providers, and other APIs. The providers are specified in the Terraform configuration code. They tell Terraform which services it needs to interact with.

Resources

Resources represent infrastructure objects and are one of the basic blocks of the Terraform language.

Data Sources

Data sources feed our Terraform configurations with external data or data defined by separate Terraform projects.

Modules

Modules help us group several resources and are the primary way to package resources in Terraform for reusability purposes.

State

Terraform keeps the information about the state of our infrastructure to store track mappings to our live infrastructure and metadata, create plans and apply new changes.

Execution Plan (Plan)

Before making changes to infrastructure, Terraform generates an execution plan based on the desired state defined in configuration files and the current state stored in the state file.

Apply Changes:

After reviewing the execution plan, you apply the changes to the infrastructure by running the terraform apply command. Terraform compares the desired state with the current state, executes the necessary actions, and updates the state file accordingly.

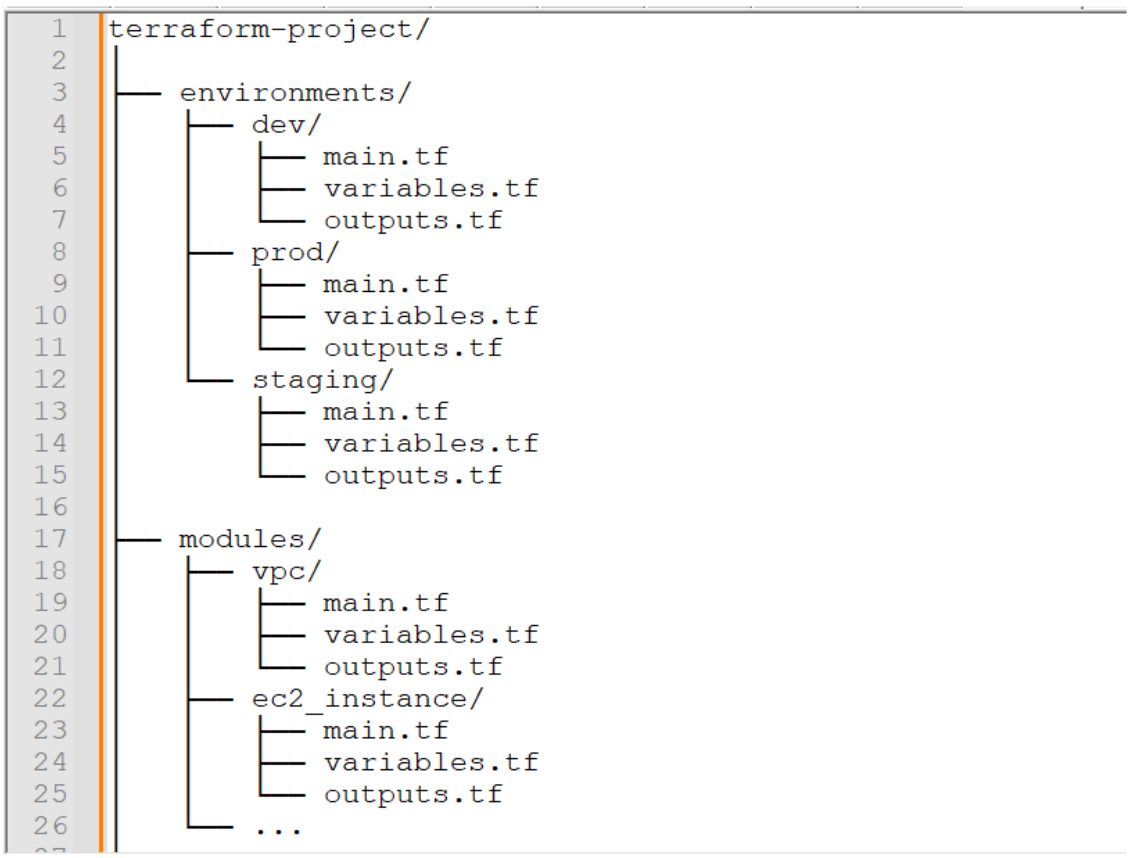
Variables and Outputs

Variables allow you to parameterize your Terraform configurations, making them more flexible and reusable across different environments. Outputs enable you to extract information from Terraform executions, such as resource attributes or computed values, for use in other parts of your infrastructure or external systems.

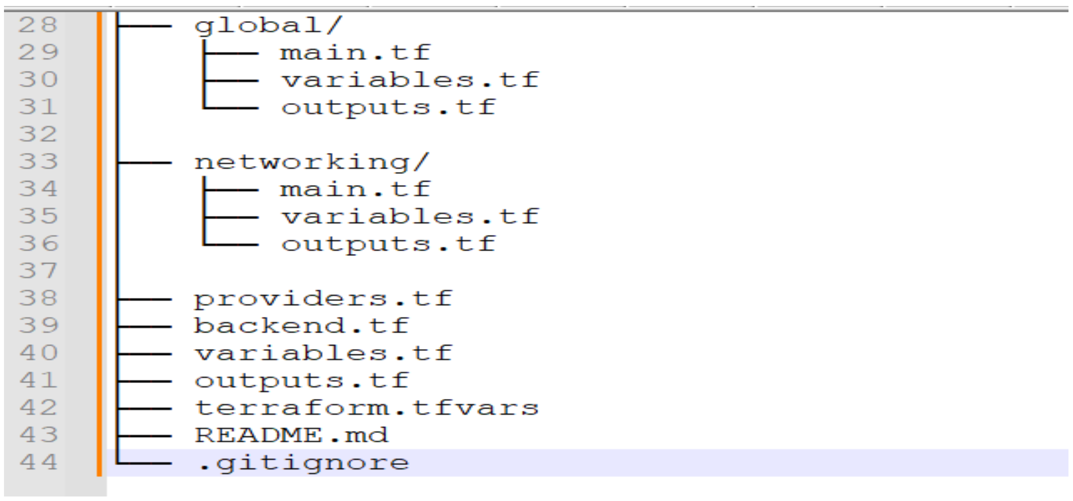
Project Organization

A well-organized directory structure is essential for enhancing the readability and maintainability of Terraform projects. A clear structure helps team members easily locate and understand different components of the project, promotes consistency, and simplifies collaboration.

Here's a recommended directory structure for a typical Terraform project:



```
terraform-project/
│
├── environments/
│   ├── dev/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   ├── prod/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   └── staging/
│       ├── main.tf
│       ├── variables.tf
│       └── outputs.tf
│
├── modules/
│   ├── vpc/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   ├── ec2_instance/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   └── ...
```



```
├── global/
│   ├── main.tf
│   ├── variables.tf
│   └── outputs.tf
│
├── networking/
│   ├── main.tf
│   ├── variables.tf
│   └── outputs.tf
│
├── providers.tf
├── backend.tf
├── variables.tf
├── outputs.tf
├── terraform.tfvars
├── README.md
└── .gitignore
```

Directory Structure

Let's break down the structure

environments/:

Contains directories for different environments (e.g., dev, prod, staging). Each environment has its own Terraform configuration files (**main.tf**, **variables.tf**, **outputs.tf**).

This structure allows you to manage environment-specific configurations independently.

modules/:

Houses reusable Terraform modules, organized by functionality (e.g., VPC, EC2 instance). Each module has its own **main.tf**, **variables.tf**, and **outputs.tf**.

Encourages modular design, making it easier to reuse and share components across different environments or projects.

global/:

Contains Terraform configurations applicable to all environments. This can include configurations for IAM roles, security groups, or any global resources.

Provides a centralized location for settings that are common across environments.

networking/:

Includes configurations related to networking components such as subnets, routes, or DNS settings.

Promotes organization based on resource type.

Providers.tf:

Lists provider configurations (e.g., AWS, Azure) used in the project.

Centralizes provider configurations for consistency.

backend.tf:

Specifies the remote backend configuration for storing the Terraform state remotely.

Helps manage state files securely and ensures collaboration.

variables.tf and outputs.tf:

Centralized files for defining input variables and outputs used across the project.

Enhances consistency and makes it easy to understand the inputs and outputs of each module or environment.

terraform.tfvars:

Contains variable values specific to a particular environment. Should not be committed to version control to avoid exposing sensitive information.

README.md:

Documentation for the project, providing an overview, usage instructions, and any additional information.

.gitignore:

Excludes unnecessary files and directories from version control.

Module Structure

Creating and using Terraform modules is a crucial practice for promoting reusability and modularity in your infrastructure code. Modules allow you to encapsulate and abstract components of your infrastructure, making it easier to maintain, share, and reuse across different projects.

Here are some guidelines for creating and using Terraform modules:

Module Structure:

- **Organization:**
  - Organize your modules into a clear directory structure. Each module should have its own directory containing **main.tf**, **variables.tf**, and **outputs.tf** files.
- **Input Variables:**
  - Define a clear and concise set of input variables for your modules in the **variables.tf** file. Use appropriate types and descriptions to make it easy for users to understand how to use the module.
- **Output Variables:**
  - Define meaningful output variables in the **outputs.tf** file. Outputs should provide essential information about the resources created by the module

Module README:

· Include a README file in each module directory. This file should provide information about the purpose of the module, its usage, and any specific considerations or requirements.

Inline Comments:

· Add inline comments within your module's main.tf, variables.tf, and outputs.tf files to explain the purpose and functionality of different sections.

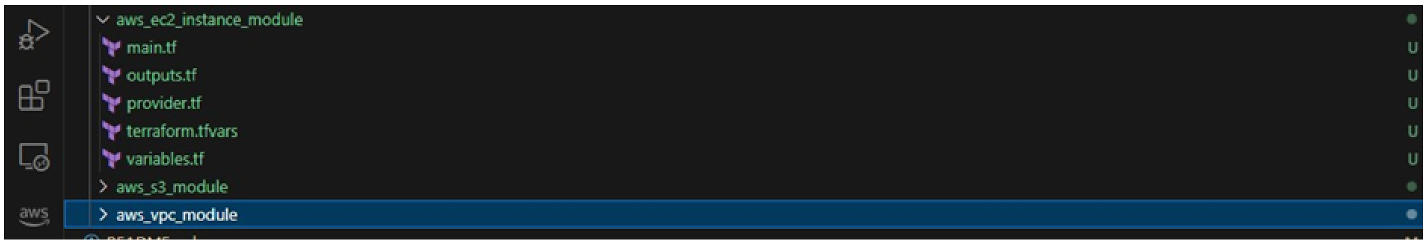
Parameterization:

· Design modules with parameterization in mind. Use input variables to allow users to customize the behavior of the module, making it adaptable to different scenarios.

Separate Module Repositories:

● Consider maintaining each module in its separate Git repository. This allows for independent versioning and testing.

Example Module Structure:



```
v aws_ec2_instance_module
    main.tf             U
    outputs.tf          U
    provider.tf         U
    terraform.tfvars    U
    variables.tf        U
> aws_s3_module
> aws_vpc_module
```

Configuration Management

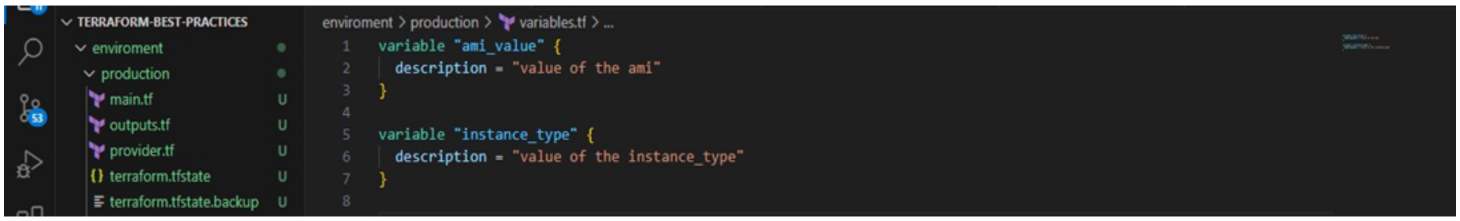
Defining and using variables and outputs in Terraform configurations is a critical aspect of writing modular, reusable, and maintainable infrastructure code.

Here are best practices for working with variables and outputs in Terraform:

Variables:

1. Consistent Naming:

● Use clear and consistent naming conventions for variables. This improves readability and helps maintain a standard across your codebase.



```
v TERRAFORM-BEST-PRACTICES
  v enviroment
    v production
        main.tf                  U
        outputs.tf               U
        provider.tf              U
        terraform.tfstate        U
        terraform.tfstate.backup U
```

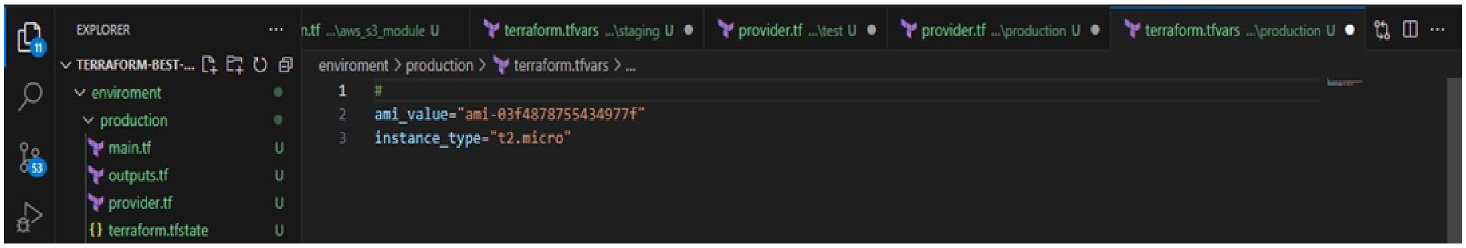
enviroment > production > variables.tf > ...

```
variable "ami_value" {
  description = "value of the ami"
}

variable "instance_type" {
  description = "value of the instance_type"
}
```

· Clearly specify the type and provide meaningful descriptions for variables. This information is valuable for users of your module or configuration.

· Consider using variable files (e.g., **terraform.tfvars**) for environment-specific variable values. This separates variable values from the configuration code.



```
#
ami_value="ami-03f4878755434977f"
instance_type="t2.micro"
```

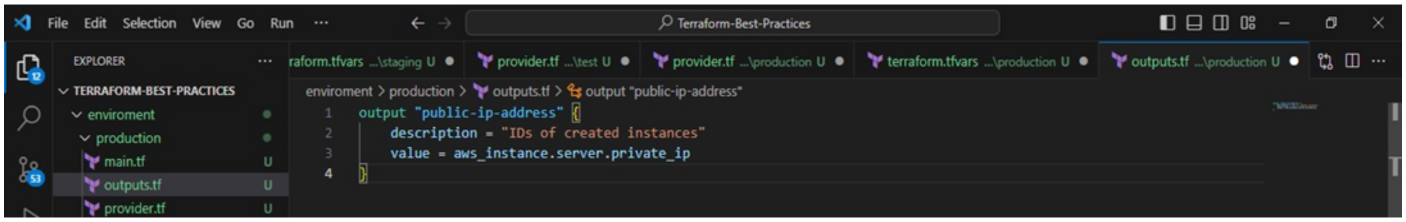
Outputs:

1. Descriptive Output Names:

● Use descriptive names for outputs to convey the purpose of each output variable.

· Include detailed documentation for each output variable in your module or configuration. Explain the purpose and usage of the output.

· Provide output values that are useful to consumers of the module or configuration. Aim to provide information that is commonly needed.



```
output "public-ip-address" {
    description = "IDs of created instances"
    value = aws_instance.server.private_ip
}
```

Design your variables and outputs with reusability in mind. Ensure that they can be easily reused in different environments or by different modules.

Resource Naming Conventions

Establishing clear and consistent naming conventions for resources in your Terraform configurations is crucial for readability, maintainability, and collaboration. A well-defined naming scheme helps everyone involved in the project quickly understand the purpose and function of each resource.

Here are some guidelines for establishing naming conventions for Terraform resources:

1. Use descriptive names to convey the purpose of each resource.

2. Employ consistent prefixes for resource types (e.g., "aws_") to categorize resources.

3. Separate words with underscores for readability.

4. Use singular nouns and keep names short yet meaningful.

5. Include environment information in resource names (e.g., "dev_web_server").

6. Avoid excessive abbreviations and adopt a consistent capitalization style.

7. Optionally include versioning in resource names (e.g., "web_server_v1").

8. Document naming conventions for team awareness and adherence.

Security Considerations

Handling sensitive data, such as passwords and API keys, in Terraform configurations requires careful consideration to ensure security and compliance.

Here are guidelines for managing sensitive information in Terraform:

1. Use Terraform Variables:

● Leverage Terraform variables to store sensitive information. Define variables for items like passwords or API keys and provide them as input during configuration.

2. Avoid Hardcoding Sensitive Values:

● Never hardcode sensitive values directly in the configuration files. Always use variables to abstract and parameterize sensitive data.

3. Input Variable Files:

● Use separate variable files (e.g., **terraform.tfvars**) to store sensitive variable values. These files can be kept out of version control and managed separately.

4. Remote Backend with Encryption:

● If using a remote backend for state storage, ensure that it supports encryption at rest. Popular remote backends like AWS S3 or Azure Storage typically provide encryption features.

5. HashiCorp Vault Integration:

● For more advanced use cases and when dealing with a larger number of secrets, consider integrating Terraform with HashiCorp Vault. Vault provides a secure way to manage and retrieve sensitive data.

6. Version Control Ignore:

● Add sensitive variable files and any other files containing sensitive information to your **.gitignore** or equivalent version control exclusion file to avoid accidental exposure.

State Management

Using remote backends for Terraform state storage offers several benefits, including collaboration, security, and scalability.

State Remote

Using remote backends in Terraform brings numerous benefits to infrastructure as code (IaC) workflows. It facilitates collaboration by providing a centralized location for storing and sharing Terraform state, preventing conflicts among team members and ensuring consistency. They also offer better concurrency control, allowing multiple users or automation processes to apply changes simultaneously without the risk of conflicts. This scalability, coupled with improved auditability, makes remote backends a key component for managing IaC in a collaborative and secure manner.

Configuring remote backends involves choosing an appropriate backend (e.g., AWS S3), updating Terraform configurations, creating the required resources, and initializing Terraform to transition from local to remote state. It's crucial to leverage locking mechanisms provided by remote backends to prevent concurrent modifications and ensure a smooth and secure IaC workflow. An example configuration for AWS S3 backend includes specifying the bucket, key, region, encryption settings, and a DynamoDB table for locking, promoting best practices for collaborative and secure infrastructure management.

State Locking

Enabling state locking is a critical best practice in Terraform to prevent concurrent modifications and maintain the integrity of infrastructure state.

When configuring the backend, choose a provider that supports state locking mechanisms, such as AWS S3 with DynamoDB. Explicitly set up the backend configuration to include locking details. During operations like `terraform apply`, use the `-lock=true` option to acquire locks explicitly, adding an extra layer of control. Terraform automatically handles locking during standard apply or destroy operations, and it releases locks after the completion of each operation.

This best practice ensures a secure and collaborative Terraform workflow, preventing unintended conflicts and disruptions when managing infrastructure as code.

Infrastructure as Code (IaC) Workflow

Effectively managing Terraform configurations in version control systems (VCS) like Git is crucial for collaboration, traceability, and maintaining a reliable infrastructure as code (IaC) workflow. Here are best practices for using Git to manage Terraform configurations:

1. **Git Versioning:**
   ○ Use Git for version control to track changes, collaborate, and rollback to previous states.
2. **Branching Strategy:**
   ○ Adopt a branching strategy (e.g., Git Flow) to organize development, testing, and production changes.
3. **Commit Messages:**
   ○ Write descriptive commit messages explaining the purpose, impact, and context of changes.
4. **Ignore Sensitive Information:**
   ○ Exclude sensitive data (e.g., Terraform state files) using **.gitignore** to prevent security risks.
5. **Modularization:**
   ○ Organize Terraform configurations into reusable modules for clarity and maintainability.
6. **Terraform State Separation:**
   ○ Store state files remotely using a backend (e.g., AWS S3) to avoid conflicts and ensure consistency.
7. **Documentation:**
   ○ Maintain a comprehensive README file with project structure, module usage, and prerequisites.
8. **Monitoring Changes:**
   ○ Utilize tools like Terraform Cloud or Atlantis for remote execution and visibility into changes.

General Guidelines

Set up your IDE for streamlined development

Time spent streamlining your IDE upfront can save developers weeks in the long run.If you regularly work with Terraform and HCL, set up syntax highlighting, auto-completion, documentation tooltips, and automatic formatting and validation of your templates.

Auto Format Terraform Files

The HashiCorp Terraform language follows the same style guidelines as most other computer languages. A single missing bracket or excessive indentation can make your Terraform configuration files difficult to read, maintain, and understand, resulting in a negative experience.

Always use ‘terraform fmt -diff’ to check and format your Terraform configuration files before you commit and push them.

Validate your Terraform Code

Validating Terraform code ensures that your configurations adhere to syntax rules and constraints before applying changes to infrastructure. The terraform validate command is used for this purpose. It checks the syntax of your configuration files and reports any errors or warnings encountered.

Always run the ‘terraform validate’ command while you are working on writing Terraform configuration files and make it a habit to identify and address problems as early as possible.

Clean up all resources

Testing infrastructure code means that you are deploying actual resources. To avoid incurring charges, consider implementing a clean-up step.

To destroy all remote objects managed by a particular configuration, use the terraform destroy command. After you run the terraform destroy command, also run additional clean-up procedures to remove any resources that Terraform failed to destroy. Do this by deleting any projects used for test execution.

Backup State Files

Backing up Terraform state files is crucial for ensuring data integrity and operational continuity. It involves creating copies of the state file, which tracks the current state of managed infrastructure.

If you’re using a remote backend like AWS S3 Bucket, versioning on the S3 bucket is highly encouraged. This way, your state file looks like it’s corrupted or in an incorrect state, and the bucket supports bucket versioning, you may be able to recover by restoring a previous version of the state file.

Leverage Helper tools

Terraform offers several helper tools and integrations that enhance its functionality and streamline workflows.Some popular Terraform helper tools include:

- tflint – Terraform linter for errors that the plan can’t catch.
- tfenv – Terraform version manager (Read more about it in the How to Use tfenv to Manage Multiple Terraform Versions article)
- checkov – Terraform static analysis tool
- terratest – Go library that helps you with automated tests for Terraform

- pre-commit-terraform – Pre-commit git hooks for automation
- terraform-docs – Quickly generate docs from modules
- atlantis – Workflow for collaborating on Terraform projects
- terraform-cost-estimation – Free cost estimation service for your plans.
- Tfsec - Security scanner for your Terraform code
- Pike - Pike is a tool for determining the permissions or policy required for IAC code

Auto-Generated Documentation

Use of tools like Terraform-docs for automatically generating documentation from code.

Publish modules to a registry

- **Reusable modules**: Publish reusable modules to a module registry.
- **Open source modules**: Publish open source modules to the Terraform Registry.
- **Private modules**: Publish private modules to a private registry.

Conclusion

Following best practices in Terraform is essential for several reasons, as it contributes to the overall efficiency, maintainability, and security of infrastructure as code (IaC) workflows:

Key Takeaways:

1. **Modularization Matters:** Break down infrastructure configurations into modular components to enhance reusability, maintainability, and scalability.
2. **Remote Backends Enhance Collaboration:** Utilize remote backend like AWS S3 for storing Terraform state centrally, fostering collaboration, and ensuring consistency.
3. **State Locking is Crucial:** Enable state locking to prevent concurrent modifications, maintain state integrity, and avoid conflicts during Terraform operations.
4. **Git for Version Control:** Leverage Git for version control, employing commits, clear branching strategies, and meaningful commit messages for effective collaboration.
5. **Documentation is Key:** Maintain comprehensive documentation with README files outlining project structure, module usage, and best practices to enhance onboarding and collaboration.
6. **Security First:** Avoid storing sensitive information in version control, use secure remote backends, and follow best practices for handling credentials to ensure the security of your infrastructure.

References

GitHub Repository - https://github.com/balcha95/Terraform-Best-Practices.git

Creating Modules - https://developer.hashicorp.com/terraform/language/modules/develop

Practices - https://cloud.google.com/docs/terraform/best-practices-for-terraform#custom-scripts

Terraform Registry - https://registry.terraform.io/?product_intent=terraform

About the Docs - https://developer.hashicorp.com/terraform/docs

AWS Provider Documentation - https://registry.terraform.io/providers/hashicorp/aws/latest/docs