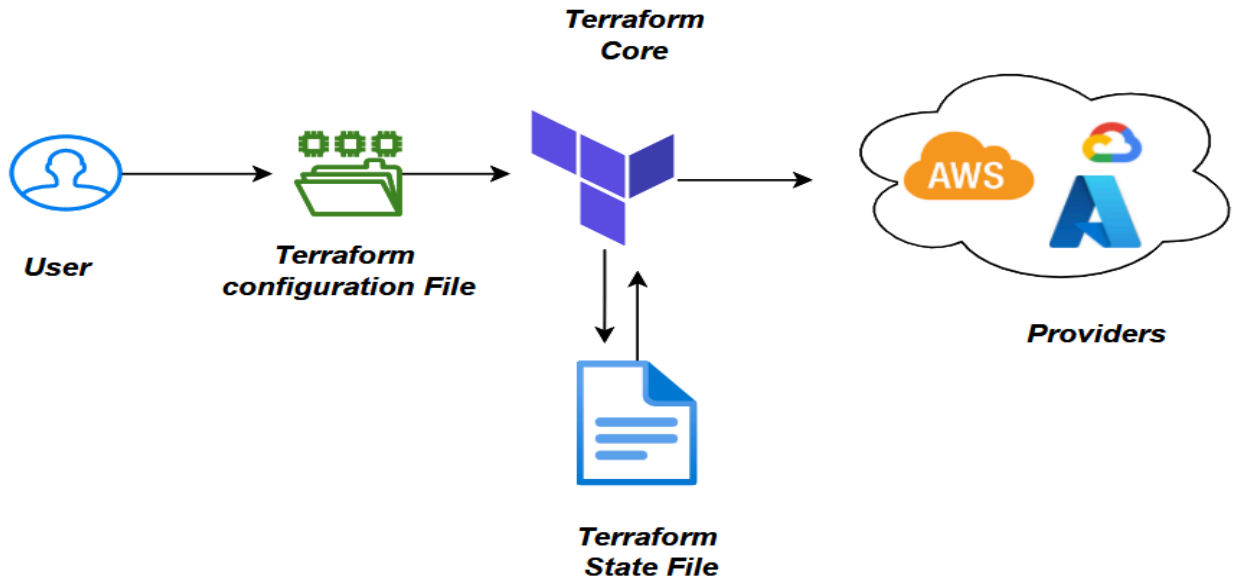
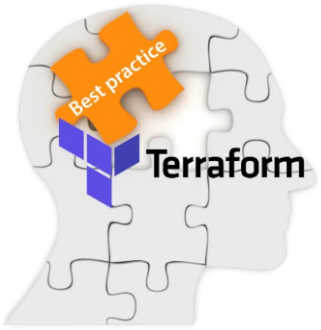
Terraform Best Practices



Terraform is a powerful Infrastructure as Code (IaC) tool. As one of the most popular tools in the IaC space, We've put together a list of best practices that should help you use this tool effectively!

In this doc, we'll help you create clean and reusable terraform code that you or our colleagues can use and improve upon later with relative ease.

IaC Best Practices



Welcome to our guide on Infrastructure as Code (IaC) Best Practices! Here, we'll explore simple yet effective strategies for managing your digital infrastructure. From making setup a breeze to keeping everything running smoothly, this document is for mastering IaC.

Directory structuring

When we are working on a large production infrastructure project using Terraform, we must follow a proper directory structure to take care of the complexities that may occur in the project. It would be best if you had separate directories for different purposes.

Note : In a big Terraform project for handling real-world infrastructure, it's crucial to keep things organized.

Here i am giving the 4 example directory structure for reference.

Client-1

Wrapper Base Repository Structure:

- This repository structure is well-suited for managing Terraform configurations for different environments and leveraging reusable modules with wrappers on top of the module.
- If you need to manage separate configurations for different environments like development (dev), quality assurance (qa) and production, this structure provides a clear organization for each environment.

```
terraform-project/
│
├── environments/
│   ├── dev/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   │   └── terraform.tfvars
│   │
│   ├── qa/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   │   └── terraform.tfvars
│   │
│   └── prod/
│       ├── main.tf
│       ├── variables.tf
│       ├── outputs.tf
│       └── terraform.tfvars
│
├── wrapper/
```

```
│   ├── main.tf
│   ├── variables.tf
│   └── outputs.tf
│
└── modules/
    ├── vpc/
    │   ├── main.tf
    │   ├── variables.tf
    │   └── outputs.tf
    │
    └── ec2/
        ├── main.tf
        ├── variables.tf
        └── outputs.tf
```

- Environments:
    - This directory contains subdirectories for different environments like development (dev), quality assurance (qa) and production (prod).
    - Each environment directory contains Terraform configuration files (main.tf, variables.tf, outputs.tf, terraform.tfvars) specific to that environment.
    - These files define the infrastructure resources, input variables, outputs, and environment-specific configurations.
- Wrapper:
    - The configuration in this directory contains an implementation of a single module wrapper pattern, which allows managing several copies of a module in places where using the native Terraform 0.13+ for_each feature is not feasible.
    - It serves as a centralized layer facilitating modular reusability and maintaining a clear separation of concerns, enhancing project scalability and maintainability.
- Modules:
    - The Modules directory contains reusable Terraform modules that define infrastructure components.
    - Each module is organized into its own directory (vpc, ec2, etc.) and contains Terraform configuration files (main.tf, variables.tf, outputs.tf) specific to that module.

Client-2

Multi-Cloud Base Repository:

If your infrastructure spans across multiple cloud providers (e.g., AWS, Azure, GCP) and you need to manage Terraform configurations for each provider separately, this structure provides a clear organization.

AWS and Azure

- These directories represent the cloud providers AWS and Azure, respectively.
- Each cloud provider directory contains subdirectories for managing infrastructure specific to that provider.

Modules

- Under each cloud provider directory, there's a Modules directory containing reusable Terraform modules.
- Each module encapsulates infrastructure components with its own Terraform configuration files.

services

- This directory is structured to manage services or applications running on the respective cloud providers.
- Under each cloud provider's services directory, there's an Environments directory containing environment-specific Terraform configurations.

- Each environment directory contains Terraform configuration files specific to that environment.

Shared Services

- This directory contains infrastructure components shared across environments or applications.
- The vpc directory contains Terraform configuration files for creating a Virtual Private Cloud (VPC), which would be shared across multiple services or applications within the same cloud provider.

```
multicloud-terraform/
├── aws/
│   ├── Modules/
│   │   ├── vpc/
│   │   │   ├── main.tf
│   │   │   ├── variables.tf
│   │   │   └── outputs.tf
│   │   ├── ec2/
│   │   │   ├── main.tf
│   │   │   ├── variables.tf
│   │   │   └── outputs.tf
│   ├── services/
│   │   ├── Environments
│   │   │   ├── Dev
│   │   │   │   ├── vpc.tf
│   │   │   │   ├── ec2.tf
│   │   │   │   ├── variables.tf
│   │   │   │   ├── provider.tf
│   │   │   │   ├── backend.tf
│   │   │   │   └── terraform.tfvars
│   │   ├── Shared Services
```

```
│   │   │   ├── vpc/
│   │   │   │   ├── main.tf
│   │   │   │   ├── variables.tf
│   │   │   │   └── outputs.tf
├── azure/
│   ├── Modules/
│   │   ├── vnet/
│   │   │   ├── main.tf
│   │   │   ├── variables.tf
│   │   │   └── outputs.tf
│   │   ├── vm/
│   │   │   ├── main.tf
│   │   │   ├── variables.tf
│   │   │   └── outputs.tf
│   ├── services/
│   │   ├── Environments
│   │   │   ├── Dev
│   │   │   │   ├── vpc.tf
│   │   │   │   ├── ec2.tf
│   │   │   │   ├── variables.tf
│   │   │   │   ├── provider.tf
│   │   │   │   ├── backend.tf
│   │   │   │   └── terraform.tfvars
│   │   ├── Shared Services
│   │   │   ├── vpc/
│   │   │   │   ├── main.tf
│   │   │   │   ├── variables.tf
│   │   │   │   └── outputs.tf
```

Client-3

WorkSpace Based Repository Structure:

When we have an exact replica of the infrastructure for all the environments, We can go ahead with this type of repository structure. Basically we will have a base sub-repo where we will create terraform configuration files and then we will provide the input values via terraformtfvars file from the environments.

```
terraform-project/
│
├── modules/
│   ├── vpc/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   │
│   └── ec2/
│       ├── main.tf
│       ├── variables.tf
│       └── outputs.tf
│
└── environments/
    ├── base/
    │   ├── main.tf
    │   ├── variables.tf
    │   └── outputs.tf
    │
    ├── dev/
    │   └── terraform.tfvars
    │
    ├── qa/
    │   └── terraform.tfvars
    │
```

```
└── prod/
    └── terraform.tfvars
```

Modules

- This directory contains reusable Terraform modules. Modules encapsulate infrastructure components with their own Terraform configuration files (main.tf, variables.tf, outputs.tf).
- Organizing modules in this directory promotes code reuse and modularity across different environments.

Environments

- This directory contains environment-specific Terraform configurations.

Base

- This subdirectory contains the base configuration that defines common infrastructure shared across all environments. It includes main.tf, variables.tf, and outputs.tf.
- dev/, qa/, staging/, prod/: These subdirectories represent different environments such as development, quality assurance, staging, and production.
- Each workspace (corresponding to an environment) will have its own state file in this directory, ensuring isolation and proper management of Terraform state.
- Each environment directory contains symlinks (main.tf, variables.tf, outputs.tf) pointing to the corresponding files in the base/ directory.
- Environment-specific configurations are defined in terraform.tfvars files, allowing for overrides and customization.

Client-4

Repository Structure for Reduce the Blast Radius:

This Repository structure will be used when we need to reduce the blast radius by isolating the resource used in the application into different directories.
Deploying plenty of configurations at once is very risky.

```
terraform-project/
│
├── Application/
│   ├── vpc/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── outputs.tf
│   │
│   └── ec2/
│       ├── main.tf
│       ├── variables.tf
│       └── outputs.tf
│
└── modules/
    ├── vpc/
    │   ├── main.tf
    │   ├── variables.tf
    │   └── outputs.tf
    │
    └── ec2/
        ├── main.tf
```

```
├── variables.tf
└── outputs.tf
```

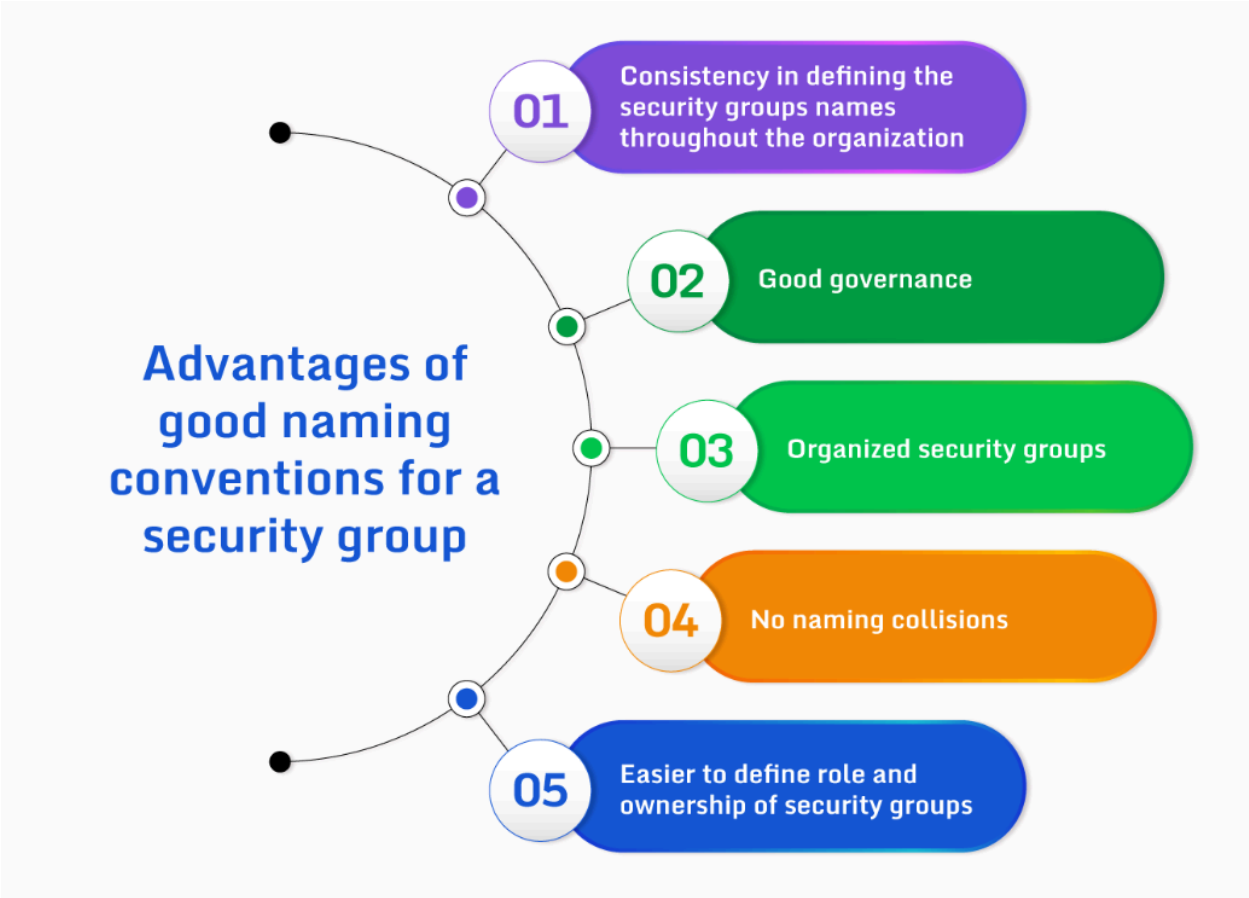
Modules

- Contains reusable Terraform modules for defining infrastructure components.
- Modules encapsulate configuration logic and promote code reuse.

Application

- Contains environment-specific configurations for development and production environments.
- Each environment directory includes subdirectories with Terraform configuration files specific to that environment.
- This structure ensures the isolation of resources between dev and prod environments, as changes made in one environment won't affect the other.

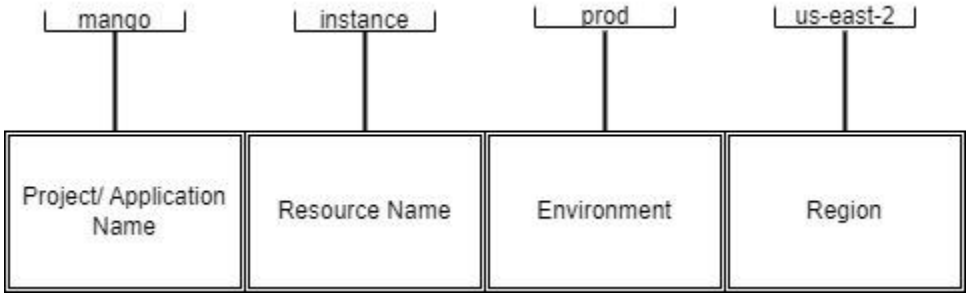
Naming conventions



While writing the aws tf code we have to follow the naming conventions.

- Use _ (underscore) instead of - (dash) everywhere (in resource names, data source names, variable names, outputs, etc).
- Prefer to use lowercase letters and numbers..

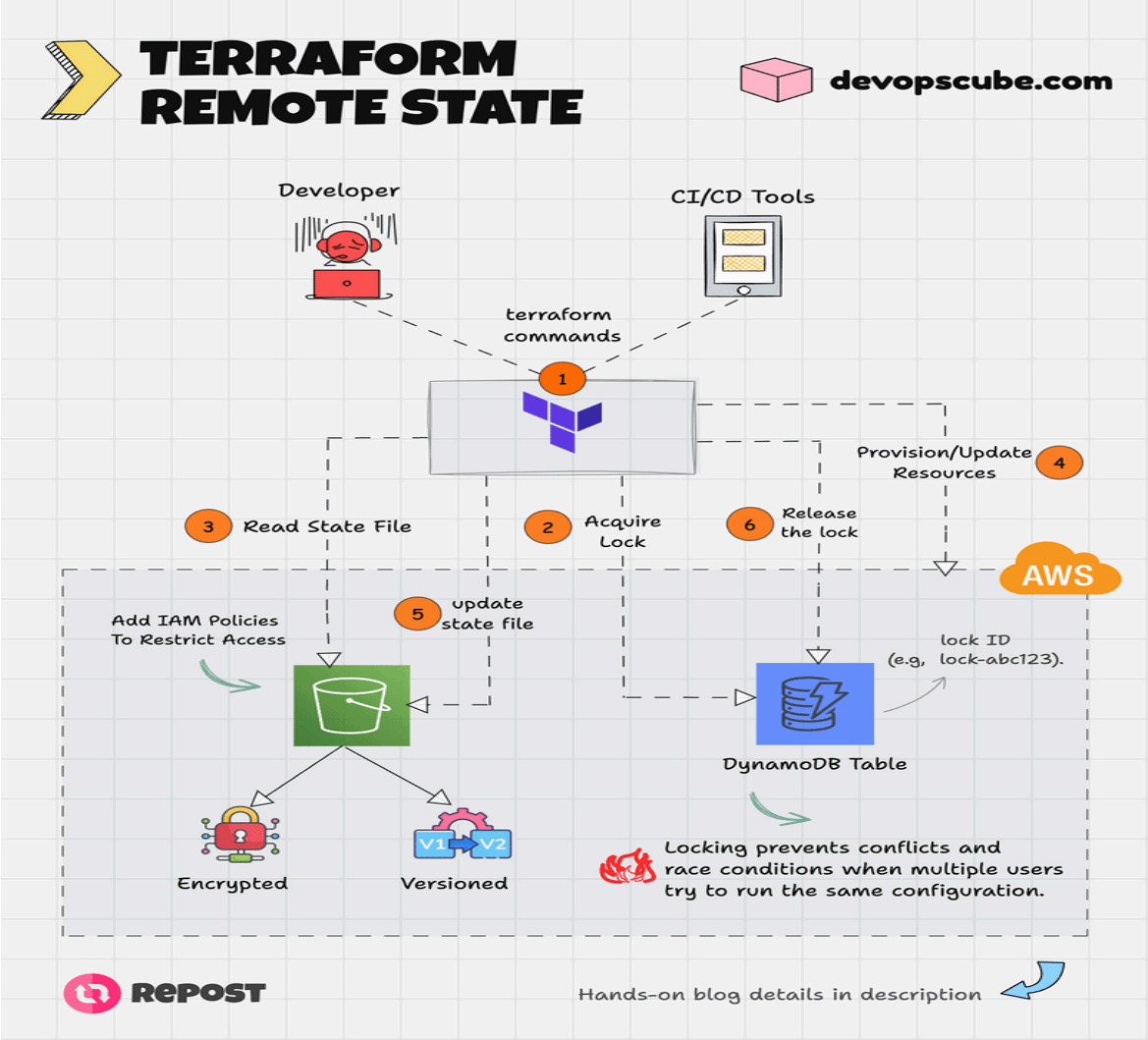
Eg:**- mango-instance-prod-us-east-2-01**



Naming Component	Description
• Project, application, or service name	• Name of a project, application, or service that the resource is a part of
• Resource Type	• An abbreviation that represents the type of AWS/Azure res
• Environment	• The stage of the development lifecycle for the workload that the resource supports.
• Location	• The region or cloud provider where the resource is deployed.

• Count (When it required)	• The instance count for a specific resource, to differentiate it from other resources that have the same naming convention and naming components. Examples, 01, 001

Use remote state



It's ok to use the local state when experimenting, but use a remote shared state location for anything above that point. Having a single remote backend for your state is considered one of the first best practices you should adopt when working in a team.

Pick one that supports state locking to avoid multiple people changing the state simultaneously.

- For setting up the backed configuration for **aws** use this.

```
terraform {
  backend "s3" {
    bucket = "mangoapps-tfbackend-***"
    key    = "backend-app/env/dev/terraform.tfstate"
    region = "us-west-2"
    dynamodb_table = "mangoapps-tfbackend-db"
  }
}
```

- For setting up the backed configuration for **Azure** use this.

```
terraform {
  backend "azurerm" {
    resource_group_name  = "ma-tf-backend-***"
    storage_account_name = "matfbacke**st"
    container_name       = "tfstate"
    key                  = "shared/rg/terraform.tfstate"
  }
}
```

Implement a Secrets Management Strategy

Never store secrets in plaintext and commit them in your version control system.

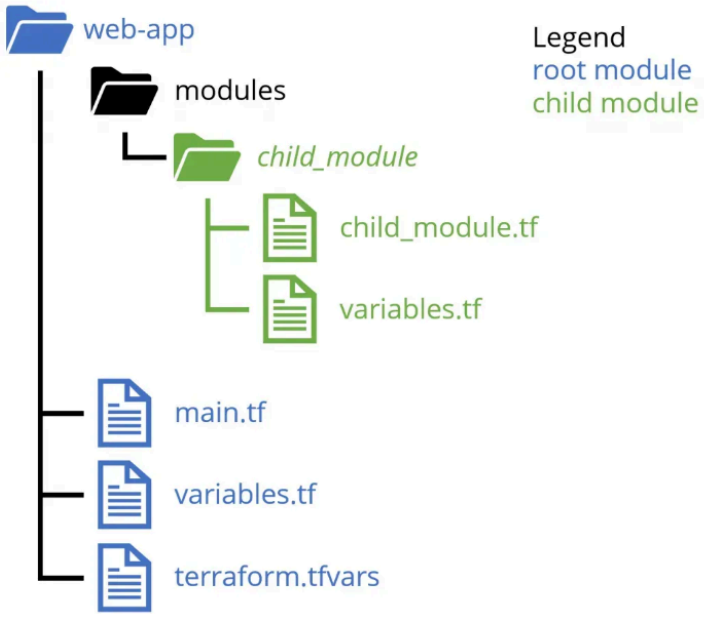
A solution would be to set up a secret store like **Hashicorp Vault** , **AWS Secrets Manager** or store in **key-vault** in azure to handle access to secrets for you. This way, you can protect your secrets at rest and enforce encryption without too much trouble.

Minimize Blast Radius

The blast radius is nothing but the measure of damage that can happen if things do not go as planned.

So, to minimize the blast radius, it is always suggested to push a few configurations on the infrastructure at a time, if something goes wrong, the damage to the infrastructure will be minimal and can be corrected quickly. Deploying plenty of configurations at once is very risky.

Build modules wherever possible



If there is no community module (terraform registry module) available for your use case, it is encouraged to build your own module. Even though sometimes, you will start by building something that seems trivial, as your Infrastructure matures, you will need to come back to your simple module and add more features to it.

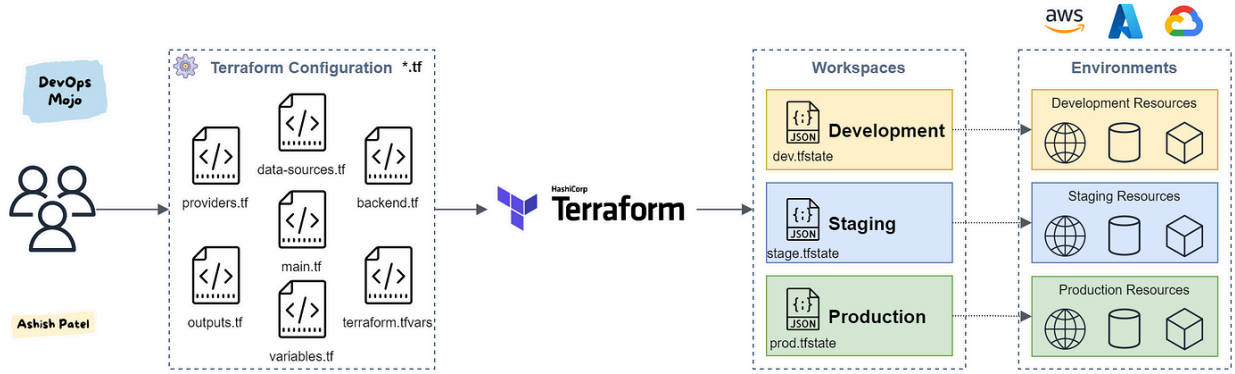
Use loops and conditionals

If any client requirement come like we want to manage several similar objects (like a fixed pool of compute instances) without writing a separate block for each one So a recommendation would be to use **for each** loop concept.

Terraform Workspaces

If any requirement comes in Terraform for managing multiple environments (such as development, staging and production) with the same infrastructure.

For managing the same we can use the workspace concept which is used to manage these environments separately within the same Terraform configuration.



One of the main thing is each workspace has its own state file, allowing you to manage resources independently. This can help prevent accidental changes to production infrastructure when working on development or testing environments.

Default Tags

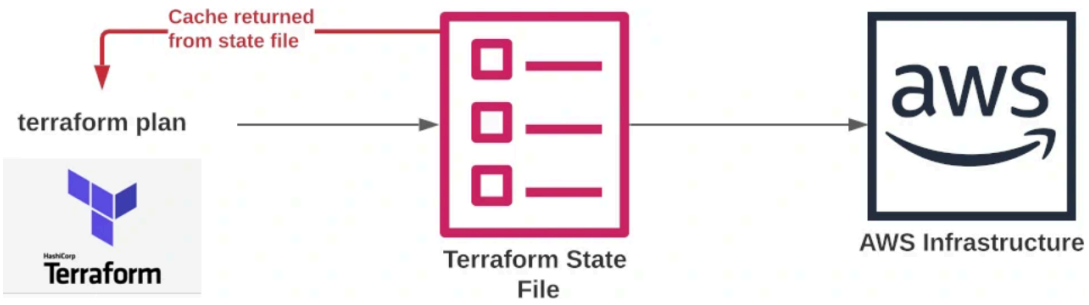
All resources that can accept tags should. The terraform aws provider has a default_tags feature that should be used inside the root module.

Consider adding necessary tags to all resources created by a Terraform module. Here is a list of possible tags to attach:

```
Name:  Human readable resource name.
Owner: The Owner of the application using the resource.
Creation_Date: The creation date of the resources.
Purpose: The resource's business purpose, e.g. "frontend ui",
"payment processor".
Environment: dev, test or prod.
Project_Name: projects that use the resource.
```

```
CostCenter: who to bill for the resource usage.
```

Retrieve state metadata from a remote backend



Terraform state with performance and caching

Normally we have several layers to manage terraform resources, such as network, database, and application layers. After you create the basic network resources, such as vpc, security group, subnets, and nat gateway in vpc stack. Your database layer and applications layer should always refer to the resource from the vpc layer directly via the terraform_remote_state data source.

Note: in Terraform v0.12+, you need to add extra outputs to reference the attributes, otherwise you will get an error message of Unsupported attribute

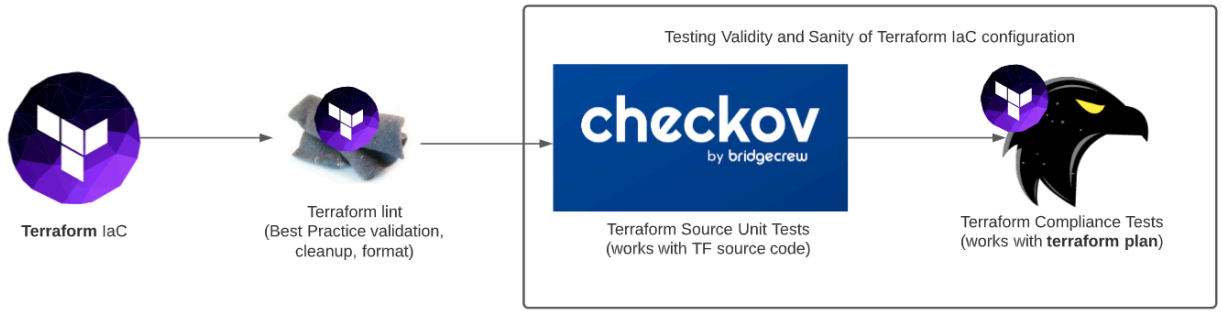
```

```

Retrieves the vpc_id and subnet_ids directly from remote backend state files.

```
resource "aws_ec2" "main" {
  subnet_ids = split(",",
```

```
data.terraform_remote_state.vpc.data_subnets)
  vpc_id     = data.terraform_remote_state.vpc.outputs.vpc_id
}
```

Integrating Security into Terraform



TFlint:

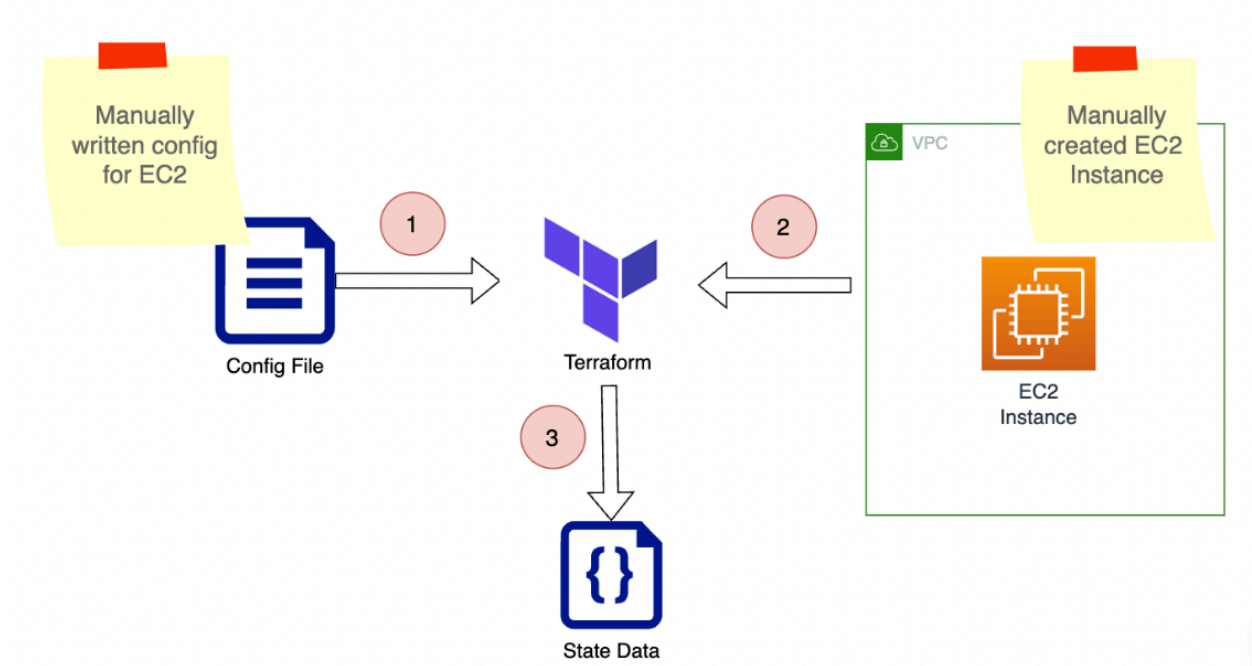
TFLint is a Terraform linter that analyzes Terraform configuration files (.tf files) and provides feedback on potential errors, best practices, and style violations. TFlint helps ensure that the Terraform code adheres to standards, follows best practices, and is free of common mistakes.

By using TFLint to analyze your Terraform configuration files, you can catch errors early in the development process, ensure consistency and adherence to best practices, and ultimately produce more reliable and maintainable infrastructure code.

Checkov:

Checkov is an open-source static code analysis tool for infrastructure as code (IaC), including Terraform configurations. It helps identify misconfigurations, security issues, compliance violations, and best practices violations in your Terraform code.

Use terraform import to include existing resources



Sometimes developers manually create resources. You need to mark these resources and use Terraform import to include them in codes.

The typical terraform workflow is straightforward when you provision resources from scratch.

However, if there are already created resources outside Terraform, you can use the "Terraform Import" command to track and manage them under Terraform.

For example, imagine you have an EC2 instance created manually using the AWS console user interface. Let's use Terraform Import to track that resource in Terraform so that the subsequent updates to the EC2 can be issued via Terraform. terraform import link

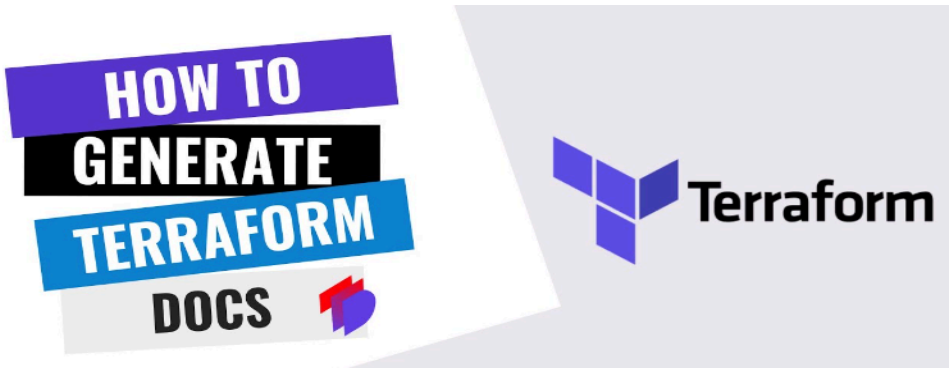
Avoid hard coding the resources

Try to avoid hardcoding the values in tf configuration. We can utilize Terraform variables, data sources, and dynamic configuration techniques. Here's how you can achieve this for an AWS Load Balancer:

- Variables: Define variables in your Terraform configuration to parameterize values that may change across different environments or deployments. For example, you can define variables for the load balancer name, listener configurations, target groups, or security groups.
- Data Sources: Use Terraform data sources to dynamically fetch information about existing resources. For instance, you can use the aws_lb data source to fetch attributes of an imported load balancer, such as its ARN or DNS name, and then reference these attributes in other parts of your configuration.

```
A sample:
account_number="123456789012"
account_alias="mycompany"
region="us-east-2"
The current aws account id, account alias and current region can be input
directly via data sources.
# The attribute `${data.aws_caller_identity.current.account_id}` will be
the current account number.
data "aws_caller_identity" "current" {}
```

Generate README for each module with input and output



Creating README files for each Terraform module is a great practice for improving documentation and facilitating collaboration within your team.

This simplified readme provides a quick reference for users on how to get started with the Terraform project, including cloning the repository, initializing Terraform, and executing common Terraform commands. It also includes a brief example demonstrating how to use the Terraform module in a configuration file.

Terraform-doc is a command-line utility that generates documentation from Terraform modules in various formats, including Markdown and reStructuredText.

Use functions & Dynamic Blocks in TF

Using functions and dynamic blocks in Terraform (TF) allows for more flexible and dynamic infrastructure configurations. They make your code more dynamic and ensure your configuration is DRY. Here's a brief description with an example for each.

Dynamic Blocks in TF:

- Description: Dynamic blocks in Terraform allow you to generate multiple blocks of configuration dynamically based on lists or maps.

```
variable "ports" {
  type = list(number)
  default = [80, 443, 22]
}

resource "aws_security_group" "example" {
  name = "example-sg"
  vpc_id = "vpc-12345678"

  dynamic "ingress" {
    for_each = var.ports
    content {
      from_port   = ingress.value
      to_port     = ingress.value
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}
```

Functions in TF:

Terraform functions can be used to manipulate strings, perform calculations, and retrieve information dynamically. In this example, we'll use the join function to concatenate strings and create a unique resource name.

```
variable "environment" {
  default = "dev"
}

resource "aws_s3_bucket" "example" {
```

```
  bucket = "${lower(join("-", ["my-bucket", var.environment]))}"
  acl    = "private"
}
```

In this example, ${lower(join("-", ["my-bucket", var.environment]))} demonstrates the use of the join() function to concatenate the string "my-bucket" with the value of the environment variable, separated by a hyphen ("-"). The lower() function is used to convert the resulting string to lowercase, ensuring consistency in the bucket name.

Use the Lifecycle Block

Sometimes, you may have some sophisticated conditions inside your code. Maybe you have a script that has to change something outside Terraform on your resources tags (this is not recommended, by the way, but it is just an example). You can use the Terraform lifecycle block to ignore changes on the tags, ensuring that you are not rolling back to the previous version.

The lifecycle block can also help if you have a pesky resource that, for some reason, seems like it is working properly, but you have to recreate it without downtime. You can use the **“create_before_destroy=true”** option for this.

```
lifecycle {
    # Prevent this resource from being destroyed
    prevent_destroy = true

    # Configure create-before-destroy behavior
    create_before_destroy = true
  }
```

Version Control

Maintaining Terraform config in VCS brings discipline, collaboration benefits common for software code. So as a best practice, We have to push all the code changes in respective VCS.

- Enforce code reviews for all changes to keep quality high and minimize configuration drift across environments.

Branching strategies commonly used with Terraform code

Choose a branching strategy that aligns with your team's development workflow.

- **Environment-Based Branching:** This strategy involves creating branches for each environment (e.g., development, staging, production). Changes are merged

into environment-specific branches for testing and validation before promotion to higher environments. This strategy ensures that changes are thoroughly tested in each environment before reaching production.

So This is not a recommended way of approach because its more difficult than all other branching strategies.

- **GitHub Flow:** GitHub Flow simplifies the GitFlow model by having only two long-lived branches: main (or master) and feature. Developers create feature branches off main, work on their changes, and then open pull requests back into main. This strategy is lightweight and suitable for teams practicing continuous delivery.
  This is the recommended way of approach, So try to implement this. because it's an easier method to work within a smaller team.

Use the latest version of Terraform

- The Terraform development community is very active, and the release of new functionalities happens frequently.
- It is recommended to stay on the latest version of Terraform as in when a new major release happens. You can easily upgrade to the latest version.
- Run the terraform -v command to check for a new update.

Use Locals Correctly

Using Terraform locals effectively can help simplify configurations, improve readability, and reduce redundancy. Here's an example demonstrating how to use locals correctly
I've seen locals in situations where variables would be better suited. I find locals to be very useful, especially in the following situations:

- Use of functions on your outputs and/or inputs
- Concatenate variables to form names of resources
- Use conditional expressions
- You can keep locals in their own file but I generally recommend keeping them in the same file and close to the code they are used for.

```
locals {
  aws_region = "us-west-2"
  subnet_cidr_block = "10.0.1.0/24"
  availability_zones = ["a", "b", "c"]
}
resource "aws_subnet" "example_subnet" {
  cidr_block = local.subnet_cidr_block
  for_each = { for idx, zone in local.availability_zones : idx => zone }
  availability_zone = each.value
}
```

Use Data Sources for Information Retrieval

Utilize Terraform data sources to fetch information from existing resources in your infrastructure.

Example (AWS VPC data source)

```
# main.tf

data "aws_vpc" "all_vpcs" {
vpc_id = "vpc628782"
}
# printing the output
output "vpc_ids" {
  value = data.aws_vpcs.all_vpcs.ids
}
```

Set Resource Dependencies Explicitly

Clearly define dependencies between resources using the depends_on attribute to ensure proper resource creation order.

In this example, the `aws_security_group` resource depends on the aws_vpc and aws_subnet resources. Terraform will ensure that the VPC and subnet are created before attempting to create the `aws_security_group`.

It's important to note that depends_on does not create a direct dependency between resources for data flow or parameter passing; it only enforces an order of operations during resource creation or destruction.

Example:

```
resource "aws_security_group" "example" {
  depends_on = [
    aws_vpc.example,
    aws_subnet.example,
  ]
}
```

Conclusion

Adopting Terraform's best practices across critical areas such as testing, collaboration, tool integration, etc. improves the efficiency, scaling, and reliability of infrastructure automation significantly while lowering risks.

I’ve covered a variety of Terraform best practices spanning organizational, operational, and delivery aspects providing ample ideas to incorporate in our Terraform guidelines and workflows today!

Reference

- https://developer.hashicorp.com/terraform/cloud-docs/recommended-practices
- https://github.com/ozbillwang/terraform-best-practices
- https://medium.com/devops-mojo/terraform-best-practices-top-best-practices-for-terraform-configuration-style-formatting-structure-66b8d938f00c
- https://www.clickittech.com/devops/terraform-best-practices/
- https://blog.gitguardian.com/9-extraordinary-terraform-best-practices/
- https://spacelift.io/blog/terraform-best-practices
- https://www.contino.io/insights/terraform-best-practices