# Terraform — Best Practices

Ashish Patel · Follow
Published in DevOps Mojo · 10 min read · May 25, 2022

404 7

Best practices for using Terraform.

DevOps Mojo — Terraform Best Practices

Terraform is is one of the most popular Infrastructure as Code (IaC) tools which allows multi-cloud infrastructure management. It uses a declarative approach, meaning you define how you want infrastructure to look rather than the steps to reach that outcome. One of the other great things about Terraform is that it is modular.

This article provides guidelines and recommendations for effective development with Terraform across multiple team members and work streams.

. . .

## Consistence File Structure

When you are working on a large production infrastructure project, you must follow a proper directory structure to take care of the complexities that may occur in the project. It is recommended to have separate directories for different purposes. Use a consistent format, style & code structure.

```
-- PROJECT-DIRECTORY/
   -- modules/
      -- <service1-name>/
         -- main.tf
         -- variables.tf
         -- outputs.tf
         -- provider.tf
         -- README
      -- <service2-name>/
         -- main.tf
         -- variables.tf
         -- outputs.tf
         -- provider.tf
         -- README
      -- ...other…

   -- environments/
      -- dev/
         -- backend.tf
         -- main.tf
         -- outputs.tf
         -- variables.tf
         -- terraform.tfvars

      -- qa/
         -- backend.tf
         -- main.tf
         -- outputs.tf
         -- variables.tf
         -- terraform.tfvars

```

```
    -- stage/
       -- backend.tf
       -- main.tf
       -- outputs.tf
       -- variables.tf
       -- terraform.tfvars

    -- prod/
       -- backend.tf
       -- main.tf
       -- outputs.tf
       -- variables.tf
       -- terraform.tfvars
```

Terraform-Project-Directory-Structure.tf hosted with ❤️ by GitHub view raw

### Terraform configurations files separation

Putting all code in `main.tf` is not a good idea, better having several files like:

- `main.tf` - call modules, locals, and data sources to create all resources.
- `variables.tf` - contains declarations of variables used in `main.tf`
- `outputs.tf` - contains outputs from the resources created in `main.tf`
- `versions.tf` - contains version requirements for Terraform and providers.
- `terraform.tfvars` - contains variables values and should not be used anywhere.

### Follow a standard module structure

- Terraform modules must follow the standard module structure.
- Group resources by their shared purpose, such as `vpc.tf`, `instances.tf`, or `s3.tf`. Avoid giving every resource its own file.
- In every module, include a `README.md` file in Markdown format which include basic documentation about the module.

### Use separate directories for each application

- To manage applications and projects independently of each other, put resources for each application and project in their own Terraform directories.
- A service might represent a particular application or a common service such as shared networking. Nest all Terraform code for a particular service under *one* directory.

### Use separate directories for each environment

- Use separate directory for each environment (`dev`, `qa`, `stage`, `prod`).
- Each environment directory corresponds to a default Terraform workspace and deploys a version of the service to that environment.
- Use only the default workspace. Workspaces alone are insufficient for modeling different environments.
- Use modules to share code across environments. Typically, this might be a service module that includes base shared configuration for service.
- This environment directory must contain the following files:
  - `backend.tf` file, declaring the Terraform backend state location.
  - `main.tf` file that instantiates the service module.

### Put static files in a separate directory

- Static files that Terraform references but doesn't execute (e.g. startup scripts) must be organized into a `files/` directory.
- Place lengthy HereDocs in external files, separate from their HCL and reference them with the `file()` function.
- For files that are read in by using the Terraform `templatefile` function, use the file extension `.tftpl`.
- Templates must be placed in a `templates/` directory.

### Common recommendations for structuring code

- Place `count`, `for_each`, `tags`, `depends_on` and `lifecycle` blocks of code in consistent locations within resources.

- Include argument `count` / `for_each` inside resource or data source block as the first argument at the top and separate by a newline.
- `tags`, `depends_on` and `lifecycle` blocks if applicable should always be listed as the last arguments, always in the same order. All of these should be separated by a single empty line.
- Keep resource modules as plain as possible.
- Don't hardcode values that can be passed as variables or discovered using data sources.
- Use data sources and `terraform_remote_state` specifically as a glue between infrastructure modules within the composition.

. . .

## Use Consistence Naming & Code Style Conventions and Formatting

Like procedural code, Terraform code should be written for people to read first. Naming conventions are used in Terraform to make things easily understandable.

### General Naming Conventions

- Use `_` (underscore) instead of `-` (dash) everywhere (resource names, data source names, variable names, outputs, etc) to delimit multiple words.
- Prefer to use lowercase letters and numbers.
- Always use singular nouns for names.
- Do not repeat resource type in resource name (not partially, nor completely).

```
resource "aws_route_table" "public" {}

// not recommended
resource "aws_route_table" "public_route_table" {}

// not recommended
resource "aws_route_table" "public_aws_route_table" {}
```

- Resource name should be named `this` or `main` if there is no more descriptive and general name available, or if the resource module creates a single resource of this type (e.g. in there is a single resource of type `aws_nat_gateway` and multiple resources of type `aws_route_table`, so `aws_nat_gateway` should be named `this` and `aws_route_table` should have more descriptive names - like `private`, `public`, `database`).
- To differentiate resources of the same type from each other (for example, `primary` and `secondary`, `public` and `private`), provide meaningful resource names.

### Variables Conventions

- Declare all variables in `variables.tf`.
- Give variables descriptive names that are relevant to their usage or purpose.
- Provide meaningful `description` for all variables even if you think it is obvious. Descriptions are automatically included in a published module's auto-generated documentation. Descriptions add additional context for new developers that descriptive names cannot provide.
- Order keys in a variable block like this: `description` , `type`, `default`, `validation`.
- When appropriate, provide default values.
  For variables that have environment-independent values (such as disk size), provide default values.
  For variables that have environment-specific values, don't provide default values.
- Use the plural form in a variable name when type is `list(...)` or `map(...)`.

- Prefer using simple types (`number`, `string`, `list(...)`, `map(...)`, `any`) over specific type like `object()` unless you need to have strict constraints on each key.
- To simplify conditional logic, give boolean variables positive names (for example, `enable_external_access`).
- Inputs, local variables, and outputs representing numeric values — such as disk sizes or RAM size — *must* be named with units (like `ram_size_gb`).
- For units of storage, use binary unit prefixes (`kilo`, `mega`, `giga`).
- In cases where a literal is reused in multiple places, you can use a local value without exposing it as a variable.
- Avoid hardcoding variables.

#### Outputs Conventions

- Organize all outputs in an `outputs.tf` file.
- Output all useful values that root modules might need to refer to or share.
- Make outputs consistent and understandable outside of its scope.
- Provide meaningful `description` for all outputs even if you think it is obvious.
- The name of output should describe the property it contains and be less free-form than you would normally want. Good structure for the name of output looks like `{name}_{type}_{attribute}`.
- If the output is returning a value with interpolation functions and multiple resources, `{name}` and `{type}` there should be as generic as possible (`this` as prefix should be omitted).
- Document output descriptions in the `README.md` file. Auto-generate descriptions on commit with tools like terraform-docs.

#### Use built-in formatting

- `terraform fmt` command is used to rewrite Terraform configuration files to a canonical format and style.
- All Terraform files must conform to the standards of `terraform fmt`.

. . .

## Better Security practices

Terraform requires sensitive access to your cloud infrastructure to operate.

#### Use remote state

- Never to store the state file on your local machine or version control.
- State file may include sensitive values in plain text, representing a security risk, anyone with access to your machine or this file can potentially view it.
- With *remote* state, Terraform writes the state data to a remote data store, which can be shared between all team members. This approach locks the state to allow for collaboration as a team.
- Configure Terraform backend using remote state (shared locations) services such as Amazon S3, Azure Blob Storage, GCP Cloud Storage, Terraform Cloud.
- It also separates the state and all the potentially sensitive information from version control.
- Don’t commit the .tfstate file source control. To prevent accidentally committing development state to source control, use gitignore for Terraform state files.
- Manipulate state only through the commands.
- Encrypt state: Even though no secrets should be in the state file, always encrypt the state as an additional measure of defense.
- Keep your backends small.
- Back up your state files.
- Use one state per environment.

### Setup backend state locking

- There can be multiple scenarios where more than one developer tries to run the terraform configuration at the same time. This can lead to the corruption of the terraform state file or even data loss.
- As multiple users access the same state file, the state file should be locked when it is in use. The locking mechanism helps to prevent such scenarios. It makes sure that at a time, only one person is running the terraform configurations, and there is no conflict.
- Not all backend support locking. e.g. Azure Blob storage natively supports locking, while Amazon S3 supports using DynamoDB in AWS.

```
terraform {
  backend "s3" {
    bucket         = "YOUR_S3_BUCKET_NAME"
    dynamodb_table = "YOUR_DYNAMODB_TABLE_NAME"
    key            = "prod_terraform.tfstate"
    region         = "us-east-1"

    #  Authentication
    profile        = "MY_PROFILE"
  }
}
```

aws-s3-backend.tf hosted with ❤ by GitHub view raw

### Don't store secrets in state

- There are many resources and data providers in Terraform that store secret values in plaintext in the state file. Where possible, avoid storing secrets in state.
- Also, never commit secrets to source control, including in Terraform configuration.
- Instead, upload them to a system like AWS Secret Manager, Azure Key Vault, GCP Secret Manager, HashiCorp Vault, and reference them by using data sources.

### Minimize Blast Radius

- The blast radius is nothing but the measure of damage that can happen if things do not go as planned.
- It is easier and faster to work with a smaller number of resources. A blast radius is smaller with fewer resources.
- For example, if you are deploying some terraform configurations on the infrastructure and the configuration do not get applied correctly, what will be the amount of damage to the infrastructure.
- To minimize the blast radius, it is always suggested to push a few configurations on the infrastructure at a time. So, if something went wrong, the damage to the infrastructure will be minimal and can be corrected quickly.

### Run continuous audits

- After the `terraform apply` command has executed, run automated security checks.
- These checks can help to ensure that infrastructure doesn't drift into an insecure state.
- `InSpec` and `Serverspec` tools are valid choices for this type of check.

### Use `Sensitive` flag variables

- Terraform configuration often includes sensitive inputs, such as passwords, API tokens, or Personally Identifiable Information (PII).
- With `sensitive` flag, Terraform will redact the values of sensitive variables in console and log output, to reduce the risk of accidentally disclosing these values.
- `sensitive` flag helps prevent accidental disclosure of sensitive values, but is not sufficient to fully secure your Terraform configuration.

```
variable "db_password" {
  description = "Database administrator password."
  type        = string
  sensitive   = true
}
```

#### Use variable definitions (`.tfvars`) files

- To set lots of variables, it is more convenient to specify their values in a *variable definitions file* (with a filename ending in either `.tfvars` or `.tfvars.json`).
- Specify that file on the command line with `-var-file`: `terraform apply -var-file=”testing.tfvars”`
- Terraform also automatically loads a number of variable definitions files if they are present.
- It is always suggested to pass variables for a password, secret key, etc. locally through *-var-file* rather than saving it inside terraform configurations or on a remote location version control system.

. . .

## Use modules

Modules are meant for reuse, use modules wherever possible.

#### Use Shared Modules

- It is strongly suggested to use official Terraform modules. No need to reinvent a module that already exists.
- It saves a lot of time and pain. Terraform registry has plenty of modules readily available. Make changes to the existing modules as per the need.
- Each module should concentrate on only one aspect of the infrastructure, such as creating instances, databases, etc.

#### Release tagged versions

- Sometimes modules require breaking changes and you need to communicate the effects to users so that they can pin their configurations to a specific version.

#### Don’t use providers or backends

- Shared modules must not declare providers or backends.
- Instead, declare providers and backends in root modules.

#### Expose outputs for all resources

- Variables and outputs let you infer dependencies between modules and resources. Without any outputs, users cannot properly order your module in relation to their Terraform configurations.
- For every resource defined in a shared module, include at least one output that references the resource.

#### Use inline submodules for complex logic

- Inline modules let you organize complex Terraform modules into smaller units and de-duplicate common resources.
- Place inline modules in `modules/$modulename`.
- Treat inline modules as private, not to be used by outside modules, unless the shared module’s documentation specifically states otherwise.

#### Minimize the number of resources in each root module

- It is important to keep a single root configuration from growing too large, with too many resources stored in the same directory and state.
- Fewer resources in a project are easier and faster to work with.

. . .

#### Version control

- Like application code, store infrastructure code in version control to preserve history and allow easy rollbacks.

- Use a default branching strategy (such as GitFlow, GitHubFlow).
- Encourage infrastructure stakeholders to submit merge requests as part of the change request process.
- Use separate environment branches for root configurations if required.
- Organize repositories based on team boundaries.

### Testing

- A combination of tools can be used to perform different types of testing to provide wider code coverage.
- **Static analysis:** To verify the contents of the configuration as well as testing the syntax and structure of your configuration without deploying any resources, using tools such as compilers, linters, and dry runs. Use `terraform validate` and tools such as `tflint`, `config-lint`, `Checkov`, `Terrascan`, `tfsec`, `Deepsource`.
- **Integration testing:** To ensure that modules work correctly, test individual modules in isolation. Use tools and frameworks such as `Terratest`, `Kitchen-Terraform`, `InSpec`.
- `terraform plan` can be used to verify the config file will work as expected for a particular component.
- Maintain a strict policy of reviewing terraform validate and plan outputs before allowing terraform changes to be applied to an environment.

### Use latest version of Terraform

- 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 `terraform -v` command to check of a new update.

### Protect stateful resources

- For stateful resources, such as databases, ensure that deletion protection is enabled.

### Use self variable

- `self` variable is a special kind of variable that is used when you don't know the value of the variable before deploying an infrastructure.
- For example, you want to use the IP address of an instance which will be deployed only after terraform apply command, so you don't know the IP address until it is up and running.

### Limit the complexity of expressions

- Limit the complexity of any individual interpolated expressions. If many functions are needed in a single expression, consider splitting it out into multiple expressions by using local values.
- Never have more than one ternary operation in a single line. Instead, use multiple local values to build up the logic.

### Use Docker

- Execute Terraform in an automated build.
- Terraform provides official Docker containers that can be used.
- When you are running a CI/CD pipeline build job, it is suggested to use docker containers. In case you are changing the CI/CD server, you can easily pass the infrastructure inside a container.

. . .

**View more from *DevOps Mojo***

- Top useful and most popular DevOps Tools
- Kubernetes Ingress Overview
- Kubernetes Architecture Overview

- Kubernetes Service Types Overview
- Kubernetes Storage Overview — PV, PVC and Storage Class

*Happy Learning!!!*

Terraform Infrastructure As Code DevOps Best Practices Cloud Computing

404 7

## Published in DevOps Mojo

Follow

1.2K Followers · Last published Dec 29, 2023

Your place to learn more about DevOps. Kubernetes, Docker, Terraform, Helm, ArgoCD, Prometheus, Grafana, Loki, Istio, Ansible, Jenkins, Fluentd, FluentBit, IaC, GitOps, CI/CD, Git, etc.

## Written by Ashish Patel

Follow

20K Followers · 18 Following

Cloud Architect • 4x AWS Certified • 6x Azure Certified • 1x Kubernetes Certified • MCP • .NET • Terraform • DevOps • Blogger [https://bit.ly/iamashishpatel]

## Responses (7)

What are your thoughts? Respond

Progress Chef
Jun 7, 2022

Thanks for sharing -- great article Ashish!

6 Reply

bloomr r
May 17, 2023

great guide ty. hashicorp needs more official guidance like this, esp about folder structure

5 Reply

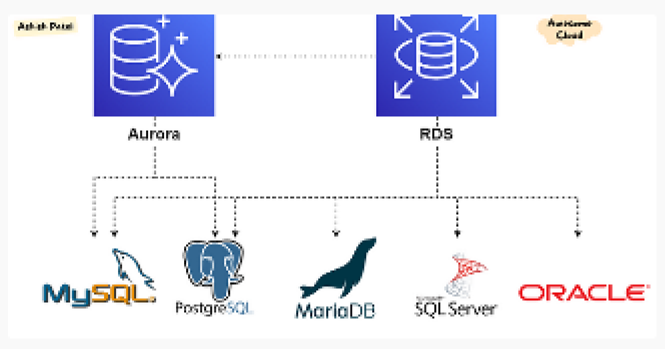
devopsforlife
Nov 18, 2022

That's a great post for DevOps guys. Thanks :3

5 Reply

See all responses

## More from Ashish Patel and DevOps Mojo

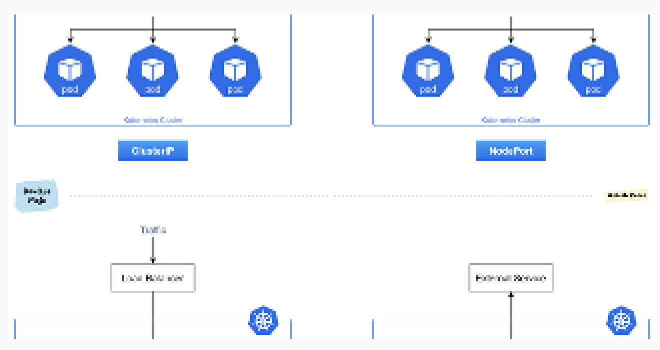


In Awesome Cloud by Ashish Patel

**AWS—Difference between Amazon Aurora and Amazon RDS**

Comparison: Amazon Aurora vs Amazon RDS.

Feb 7, 2022 1.1K 5

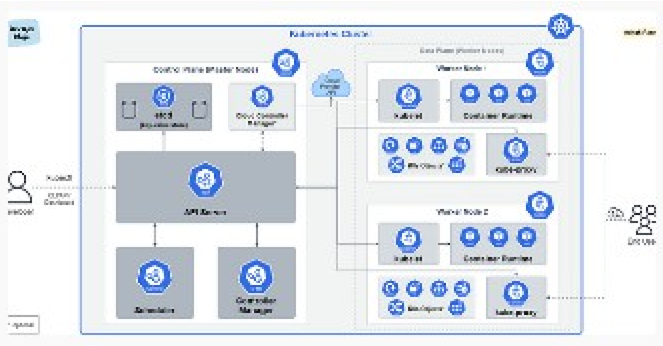


In DevOps Mojo by Ashish Patel

**Kubernetes—Service Types Overview**

Introduction to Service types in K8s — Types of Kubernetes Services.

Jul 1, 2021 427 6

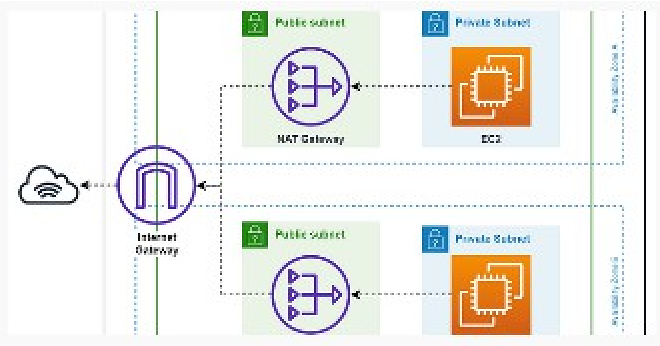


In DevOps Mojo by Ashish Patel

**Kubernetes—Architecture Overview**

Introduction to Kubernetes Architecture and Understanding K8s Cluster Components.

Aug 13, 2021 239 2



In Awesome Cloud by Ashish Patel

**AWS—Difference between Internet gateway and NAT gateway**

Internet gateway vs NAT gateway in AWS

May 25, 2019 1.2K 12

See all from Ashish Patel | See all from DevOps Mojo

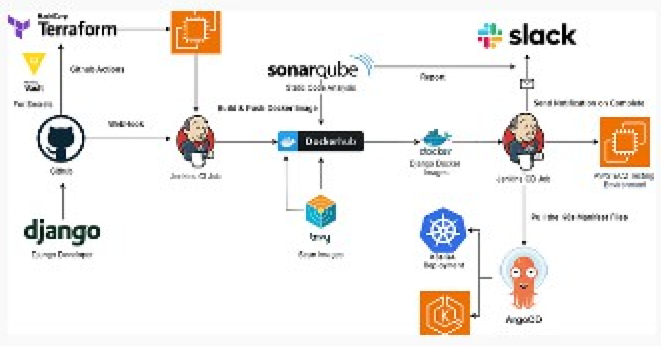
## Recommended from Medium

wqwq

**Simplify Terraform with Terragrunt**

Introduction

Nov 24, 2024 6



In Django Unleashed by Joel Wembo

**Technical Guide: End-to-End CI/CD DevOps with Jenkins, Docker,...**

Building an end-to-end CI/CD pipeline for Django applications using Jenkins, Docker,...

Apr 12, 2024 1K 21

Lists

General Coding Knowledge
20 stories · 1910 saves

Productivity
244 stories · 682 saves

Natural Language Processing
1932 stories · 1586 saves

Workable Tech Blog

**From manual to declarative: Terraform and IaC in a fast growin...**

Our journey from manually provisioning and maintaining infrastructure resources, to fully...

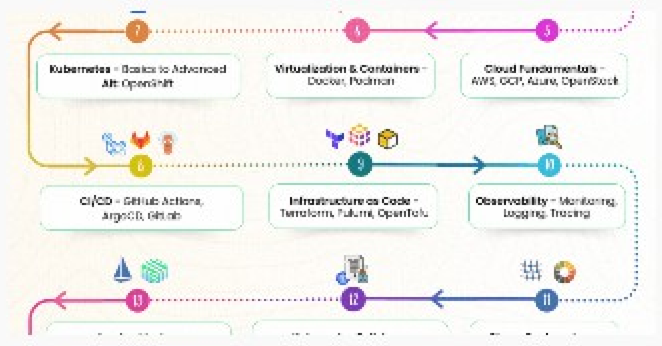
Nov 24, 2023 147 2

In AWS Tip by Krishna Perugupalli

**Top 18 Terraform CLI Commands You Need to Know**

Terrafrom CLI cheetsheet

Aug 16, 2024 13



Rohit Ghumare

In Terraform & Beyond by Shipa S Behani

**DevOps Roadmap 2025**

In today's rapidly evolving tech landscape, DevOps has become more than just a...

Jan 15 338 8

**Terraform Workspaces: Managing Multiple Environments**

When managing infrastructure across multiple environments like development,...

Sep 28, 2024 10

See more recommendations

Help Status About Careers Press Blog Privacy Terms Text to speech Teams