COMMUNITY DAY

PUNE 2022

aws

COMMUNITY DAY

PUNE 2022

Infrastructure as Code Best Practices for AWS With CloudFormation & Terraform

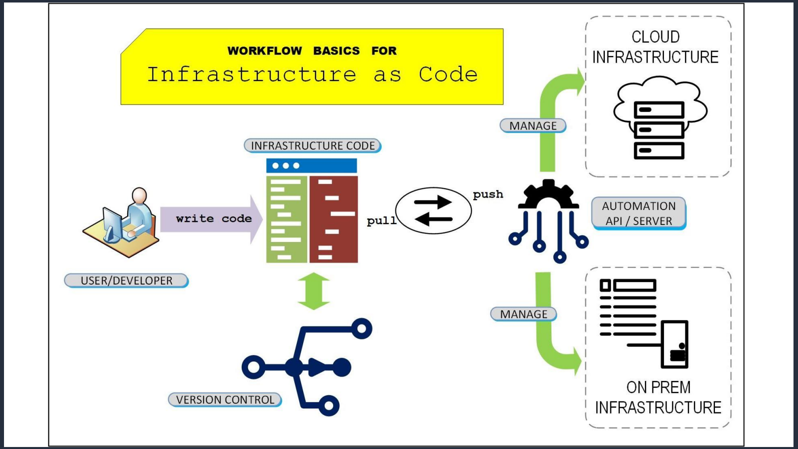
By Sandip Das, Sr. Cloud & DevOps Engineer, AWS Container Hero

Who AM I ?

I am a Senior Cloud & DevOps Engineer, AWS Container Hero, working for multiple tech companies with multiple cutting edge technologies, love writing tech article on Cloud (mainly on AWS) and make video tutorials on my YouTube Channel.

What is Infrastructure as Code?

A fundamental principle of DevOps is to treat infrastructure the same way developers treat code. Application code has a defined format and syntax. If the code is not written according to the rules of the programming language, applications cannot be created. Code is stored in a version management or source control system that logs a history of code development, changes, and bug fixes. When code is compiled or built into applications, we expect a consistent application to be created, and the build is repeatable and reliable. Example: AWS CloudFormation, Terraform, Chef, Puppet, Ansible etc.

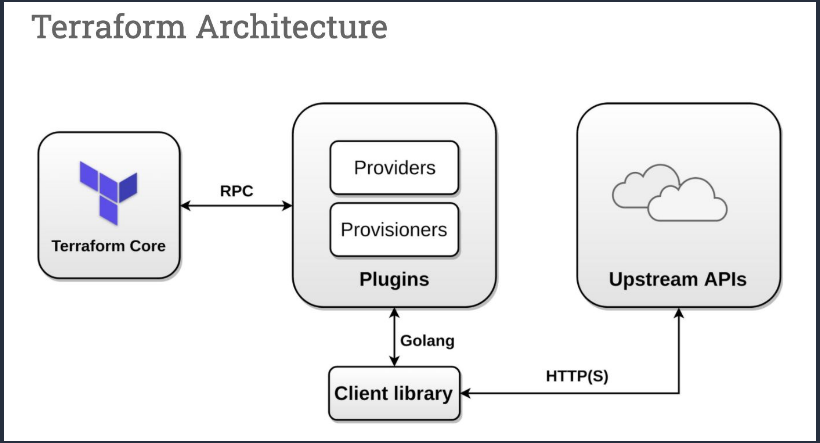


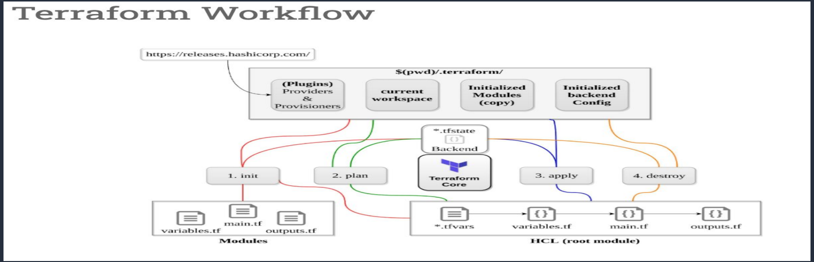
What is Terraform?

Terraform is an infrastructure as code (IaC) tool that allows you to build, change, and version infrastructure safely and efficiently. This includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc. Terraform can manage both existing service providers and custom in-house solutions.

Terraform has become one of the most popular **Infrastructure-as-code (IaC) tool,** getting used by DevOps team worldwide to automate infrastructure provisioning and management.

Terraform is an open-source, cloud-agnostic provisioning tool developed by HashiCorp and written in GO language.



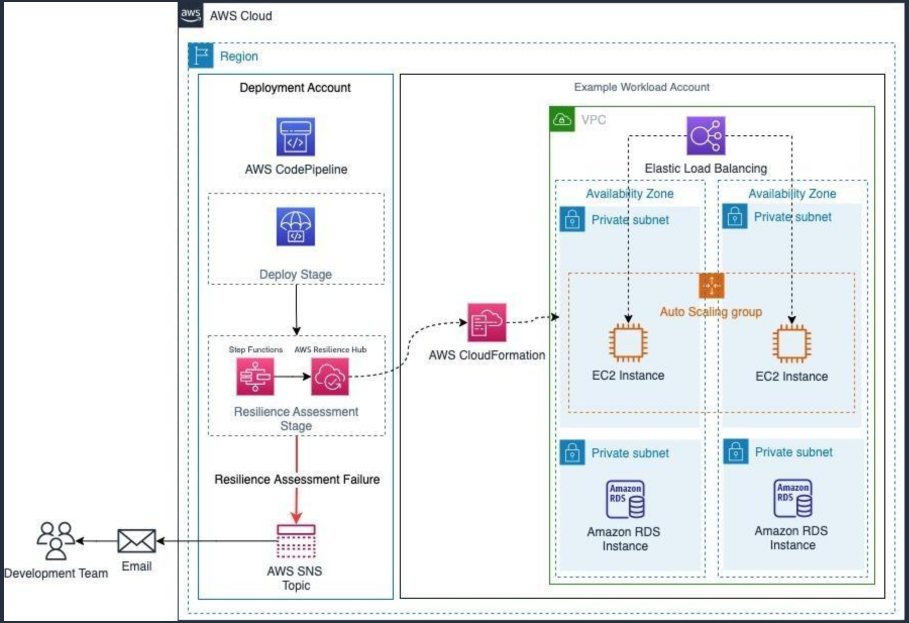


aws
COMMUNITY DAY
PUNE 2022

What is AWS CloudFormation?

AWS CloudFormation is a service that helps us model and set up our AWS resources so that we can spend less time managing those resources and more time focusing on our applications that run in AWS.

We can create a template that describes all the AWS resources that you want (like Amazon EC2 instances or Amazon RDS DB instances), and CloudFormation takes care of provisioning and configuring those resources for you. You don't need to individually create and configure AWS resources and figure out what's dependent on what; CloudFormation handles that. The following scenarios demonstrate how CloudFormation can help.



aws COMMUNITY DAY

PUNE 2022

Best Practices while Working With CloudFormation & Terraform

aws COMMUNITY DAY
PUNE 2022

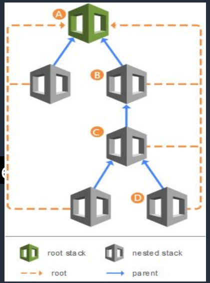
1. Use Templates

AWS Cloudformation

We can have use single CloudFormation template to automate a complex deployment but that would make debugging / reading code hard and complicated, so instead it’s better break into smaller templates.

This would also help in achieve code reusability. We can call it nested stacks as well.
Example.

```
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  myStack:
    Type: AWS::CloudFormation::Stack
    Properties:

TemplateURL:https://s3.amazonaws.com/cloudformation-templat
east-1/S3_Bucket.template
      TimeoutInMinutes: '60'
Outputs:
  StackRef:
    Value: !Ref myStack
  OutputFromNestedStack:
    Value: !GetAtt myStack.Outputs.BucketName
```



Terraform

Terraform too have supports for templates which does the similar job and increase the code reusability . We can write terraform template files with *.tftpl files.

Example:

```
data "template_file" "init" {

template = "${file("${path.module}/init.tpl")}" vars
= { consul_address =
"${aws_instance.consul.private_ip}" }

 }
```

init.tpl

#!/bin/bash echo "CONSUL_ADDRESS = ${consul_address}" > /tmp/iplist

AWS COMMUNITY DAY PUNE 2022

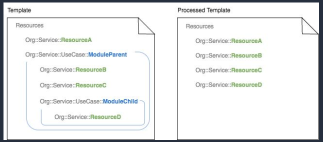
2. Use Modules

AWS Cloudformation

In CloudFormation, Modules are a way for us to package resource configurations for inclusion across stack templates, in a transparent, manageable, and repeatable way. Modules can encapsulate common service configurations and best practices as modular, customizable building blocks for you to include in your stack templates. Example:

```
// Template containing My::S3::SampleBucket::MODULE
{
  "Parameters": {
    "BucketName": {
      "Description": "Name for your sample bucket",
      "Type": "String"
    }
  },
  "Resources": {
    "MyBucket": {
      "Type": "My::S3::SampleBucket::MODULE",
      "Properties": {
        "BucketName": {
          "Ref": "BucketName"
        }
      }
    }
  }
}
```

```
// My::S3::SampleBucketPrivate::MODULE
{
   "AWSTemplateFormatVersion": "2010-09-09",
   "Description": "A sample S3 Bucket with Versioning and DeletionPolicy.",
   "Parameters": {
      "BucketName": {
         "Description": "Name for the bucket",
         "Type": "String"
      },
      "AccessControl": {
         "Description": "AccessControl for the bucket",
         "Type": "String"
      }
   },
   "Resources": {
      "S3Bucket": {
         "Type": "AWS::S3::Bucket",
         "Properties": {
            "BucketName": {
               "Ref": "BucketName"
            },
            "AccessControl": {
               "Ref": "AccessControl"
            },
            "DeletionPolicy": "Retain",
            "VersioningConfiguration": {
               "Status": "Enabled"
            }
         }
      }
   }
}
```



Terraform

A Terraform module is a set of Terraform configuration files in a single directory. Even a simple configuration consisting of a single directory with one or more .tf files is a module. When you run Terraform commands directly from such a directory, it is considered the root module. So in this sense, every Terraform configuration is part of a module. Modules are the main way to package and reuse resource configurations with Terraform.
Example:

Calling a nested module:

```
module "NestedA" {

 source = "../modules/nestedA"

 # (module arguments)

}
```

```
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
├── ...
├── modules/
│   ├── nestedA/
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   ├── nestedB/
│   ├── .../
├── examples/
│   ├── exampleA/
│   │   ├── main.tf
│   ├── exampleB/
│   ├── .../
```

3. Use **integrated development environment with linting**

AWS Cloudformation

It takes lot of time to develop and deploy AWS CloudFormation template varies from a few minutes to several hours. With complex deployments, a developer might make a simple mistake and not realize it until a stack fails late during testing. Discovering mistakes at the last minute can lead to unnecessary frustration and wasted time.

While AWS CloudFormation templates and scripts can be developed with any text editor, working in an integrated development environment (IDE) can improve the process. An IDE can catch formatting mistakes in real time, can display aspects of your code in different colors to highlight them, and can reformat multiple lines of code simultaneously. Most IDEs can incorporate third-party linting tools to ensure that your code is developed correctly for particular applications, including AWS CloudFormation.

AWS CloudFormation Linter (cfn-lint) is an open-source tool maintained by the AWS CloudFormation team. Cfn-lint analyzes AWS CloudFormation templates and checks for syntactic errors. Suggested Editor: VS Code, Atom etc

Terraform

The same applies for Terraform as well, although it's easier to write in Terraform than writing in CloudFormation, but while writing complex configuration as code , if missed even something minor mistake might cause hours of debugging!

It's easy find such early stage error via any integrated development environment (IDE)

Terraform has official support for all major IDEs , just simply search in extension **"Hashicorp Terraform"**, and you will get the extension.
This extension help us via: syntax highlighting , syntax validation , auto completion, auto formatting and much more features. Having right extension would help maintaining consistent code quality by detecting issues early.

4. Naming variable and resources

AWS Cloudformation

When we use a modular approach with complex deployments, you may find yourself working with a large number of AWS CloudFormation templates. So, Keeping parameter names consistent across all AWS CloudFormation templates makes it easier to troubleshoot and iterate.

e.g.

- VPCCIDR – Classless Inter-Domain Routing (CIDR) block of the VPC into which the Quick Start is deployed
- PrivateSubnet1ID – ID of the first Availability Zone's private subnet
- PrivateSubnet2ID – ID of the second Availability Zone's private subnet
- PublicSubnet1ID – ID of the first Availability Zone's public subnet
- PublicSubnet2ID – ID of the second Availability Zone's public subnet

Terraform

It's best practice to Name all configuration objects using underscores to delimit multiple words. This practice ensures consistency with the naming convention for resource types, data source types, and other predefined values

e.g.

```
resource "resource_type_name" "resource_name" {
  name = "resource-name"
  }
```

Variables must have descriptions. Descriptions are automatically included in a published module's auto-generated documentation. Descriptions add additional context for new developers that descriptive names cannot provide.

5. Automate testing

AWS Cloudformation

When you build AWS CloudFormation templates, you must test them.

It's time-consuming to manually deploy AWS CloudFormation templates across multiple AWS Regions and then clean up the assets you deployed during testing.

To address these issues, the AWS Quick Start team has developed a process to automate AWS CloudFormation testing using an open-source tool called **TaskCat**.

We enter Regions and parameter values into a YAML-formatted file called `taskcat.yml` in the root of your repository. TaskCat uses these values to automatically deploy your AWS CloudFormation templates.It then logs the results of the deployment—whether it succeeded or failed (and, if it failed, the reason)—and deletes any assets that were deployed.

Terraform

Same goes for Terraform, it's time consuming to test manually, an there are multiple test tools available, out of which the most promising one is **Terratest**

Terratest is a Go library that makes it easier to write automated tests for your infrastructure code. It provides a variety of helper functions and patterns for common infrastructure testing tasks for terraform:

e.g.

```
package test

import (
	"testing"

	"github.com/gruntwork-io/terratest/modules/terraform"
	"github.com/stretchr/testify/assert"
)

func TestTerraformHelloWorldExample(t *testing.T) {
	// retryable errors in terraform testing.
	terraformOptions := terraform.WithDefaultRetryableErrors(t, &terraform.Options{
		TerraformDir: "../examples/terraform-hello-world-example",
	})

	defer terraform.Destroy(t, terraformOptions)

	terraform.InitAndApply(t, terraformOptions)

	output := terraform.Output(t, terraformOptions, "hello_world")
	assert.Equal(t, "Hello, World!", output)
  }
```

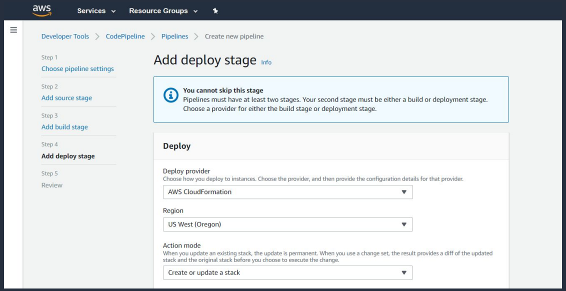
AWS COMMUNITY DAY PUNE 2022

6. Deploy it like code

AWS Cloudformation

Just like how we can have CI/CD for normal applications, for Infrastructure as code also we can go for the same path.

Doing this , it's not just help deployment / infrastructure updates or rollbacks (if any issue) get faster , it also reduce chances of human errors! We can host CloudFormation templates in AWS CodeCommit => then on code push AWS CodePipeline will trigger deployment via AWS CodeBuild (if any test / pre processing required or skip if not required) => AWS CodeDeploy for the deployment



Add deploy stage Info

You cannot skip this stage
Pipelines must have at least two stages. Your second stage must be either a build or deployment stage. Choose a provider for either the build stage or deployment stage.

Deploy

Deploy provider
Choose how you deploy to instances. Choose the provider, and then provide the configuration details for that provider.
AWS CloudFormation

Region
US West (Oregon)

Action mode
When you update an existing stack, the update is permanent. When you use a change set, the result provides a diff of the updated stack and the original stack before you choose to execute the change.
Create or update a stack

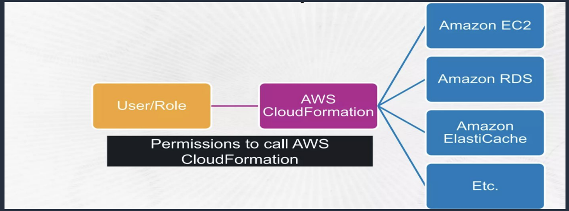
Terraform

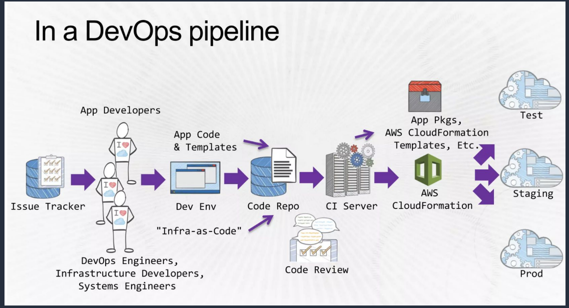
To have a CI/CD for Terraform, we can use AWS CodeCommit to version control .tf codes => on code push have AWS CodePipeline trigger AWS CodeBuild to deploy terraform changes to the target environment.

```
version: 0.2


phases:
 install:
   runtime-versions:
     python: 3.7
   commands:
     - "cd /usr/bin"
     - "curl -s -qL -o terraform.zip
https://releases.hashicorp.com/terraform/${TF_VERSION}/terraform_${TF_VERSION}_linux_amd64.z
ip"
     - "unzip -o terraform.zip"


 build:
   commands:
     - cd "$CODEBUILD_SRC_DIR"
     - terraform init -input=false --backend-config=./env_vars/${TF_ENV}.conf
     - terraform ${TF_ACTION} -input=false -var-file=./env_vars/${TF_ENV}.tfvars -auto-approve


 post_build:
   commands:
     - echo "Terraform completed on `date`"


artifacts:
 files:
   - '**/*'
```

AWS CloudFormation Bonus Tips:

1) Ensure sufficient permission via IAM to call Cloudformation and other resources
2) Utilize CloudWatch Logs for debugging
3) Check periodically for drift detection
4) Make sure to have full scooped DevOps pipeline i.e. it's smart choice to have Test , Staging & Production separate environments.
5) Do not embed credentials in template, instead use Input parameters or AWS Parameter Store / Secret Manager for the same with encryption enabled.
6) Use cross stack references to export shared resources. Stacks can use the exported resources by calling them using the Fn::ImportValue
7) Ensure that stack can create all the required resources without hitting the AWS account limits.
8) Validate templates before creating or updating a stack, it helps catch syntax error such as circular dependencies, before AWS CloudFormation creates any resources, also it check JSON/YAML valid or not.
9) Use Change Set to preview of how the proposed changes to a stack might impact the running resources before you implement them
10) Stack policies help protect critical stack resources from unintentional updates that could cause resources to be interrupted or even replaced
11) Make sure to use tags for each and every resources





Terraform Bonus Tips:

1) Ensure sufficient permission via IAM to make sure terraform has required permission to do plans, apply / destroy resources as required.
2) Run terraform with var file, so that we can easily manage manage environment (dev/stag/uat/prod) via variables e.g.

```
terraform plan -var-file=config/dev.tfvars
```

1) Manage S3 Back-end to store terraform state files, as saving in local is dangerous and if working in team it will create inconsistent / conflicting resources.
2) Manage multiple Terraform modules and environments easily with Terragrunt, Terragrunt is a thin wrapper for Terraform that provides extra tools for working with multiple Terraform modules
3) Turn-on Debug mode when troubleshooting and need more details on errors using:
   TF_LOG=DEBUG terraform <command>

1) Use terraform version manager to use different terraform version for different projects.
2) Use moved statement for terraform migrations , e.g. normal terraform code to terraform module migration, to make sure existing resources don't get removed
3) Use "count" to dynamically manage resources

```
# Terraform s3 Back-end
terraform {

 backend "s3" {

   bucket = "mybucket"

   key    = "path/to/my/key"

   region = "us-east-1"

 }

}
```

```
# install several terraform binary with different versions
$ tfenv install 1.1.9
$ tfenv install 1.2.1
$ tfenv install 0.12.11

# list terraform versions managed by tfenv
$ terraform list

# set the default terraform version
$ terraform use 1.2.1
$ terraform version
```

COMMUNITY DAY

PUNE 2022

Thank you!

See you at the AWS Community Day Pune 2023