Terraform Architecture

Terraform State

Terraform must store state and configuration information about the infrastructure it manages. This state is used by Terraform to map real world resources to your configuration and keep track of metadata about the infrastructure.

This state information is then used by Terraform to:

- Detect configuration drift of real world resources
- Plan creation, modification and destruction of resources
- Rollout changes to resources

More information on Terraform state is here.

Remote State

Since multiple people will be working in the same Terraform environments, it's important to have the state information centralized. This allows parallel development to happen while ensuring everyone stays in sync with changes to real world resources.

We will be using Amazon S3 for the storage of remote state since it's highly available and can natively version our state information in case of corruption or accidental deletion.

We will be using a state bucket per account. Current buckets are:

- Sandbox: `ds-tfstate-sandbox`
- Testing: `ds-tfstate-testing`
- Production: `ds-tfstate`

DynamoDB will be used to lock state modifications so multiple processes or people can't update the same state at the same time.

- Table: `terraform-lock`
- Region: us-east-1

Inside the bucket, state will be grouped by application, environment, and when necessary; by service: `/<application>/<environment>/<service>`

Examples:

- `/ogm/global/iam`
- `/ogm/global/s3`
- `/ogm/dev/web`
- `/ogm/prod/cache`

When creating state files, be cognisant of how often the state may change and separate large state files into smaller ones to limit the potential *blast radius* of a change. For example, we would not want our ecommerce platform and VPC in the same state file since the VPC rarely changes, but is put at risk every time an ecommerce deployment is done.

A similar pattern will be used for AWS core services that the Cloud Core Team maintains, but since many of our environments are either global or by region, the names of the environments are slightly different: `/<application>/<region>/<service>`

Examples:

- `/network/us-east-1/management-vpc`
- `/network/us-east-1/production-vpc`
- `/network/us-east-1/testing-vpc`

AWS resources that are cross cutting across all applications or are defined at the AWS account level will be created as if they were applications. Examples are SSO roles, DNS for VPCs and the centralized logging S3 bucket.

Examples:

- `/iam/sso`
- `/dns/vpc-zones`

Note

- If you tinker with your Terraformed infrastructure through the AWS Console or AWS APIs, your Terraform state will become out of sync. This usually causes confusing problems that are difficult to resolve.
- If you modify the `key` path in your state backend configuration after deployment, the new path will not have any state information in it and Terraform will try to recreate

resources. This will fail and you will likely have to move state information to the new location.

Example Remote State Configuration

```
# project-root/accounts/sandbox/{region}/config.tf
provider "aws" {
  region = "us-east-1"
}

terraform {
  backend "s3" {
    bucket         = "ds-tfstate-sandbox"
    key            = "myapp/development/web"
    region         = "us-east-1"
    dynamodb_table = "terraform-lock"
  }

  required_version = ""
}
```

Project Structure

Refer to the Structure Your Terraform Project guide.

Module Versioning

Terraform shared modules will live in the terraform-modules group in GitLab.

Modules will use semantic versioning. Each shared module will maintain major revision branches (v1,v2,...). Breaking changes will be merged to a new major revision branch. Non-breaking feature additions or bug-fixes may be pushed to the existing version branch.

Consumers of the shared modules will reference the module by the major-revision branch like this:

```
module "my_standalone_server" {
  source = "git::ssh://git@gitlab.directsupply.cloud/terraform-modules/standalone-ec2.git?ref=v1"
  ...
}
```