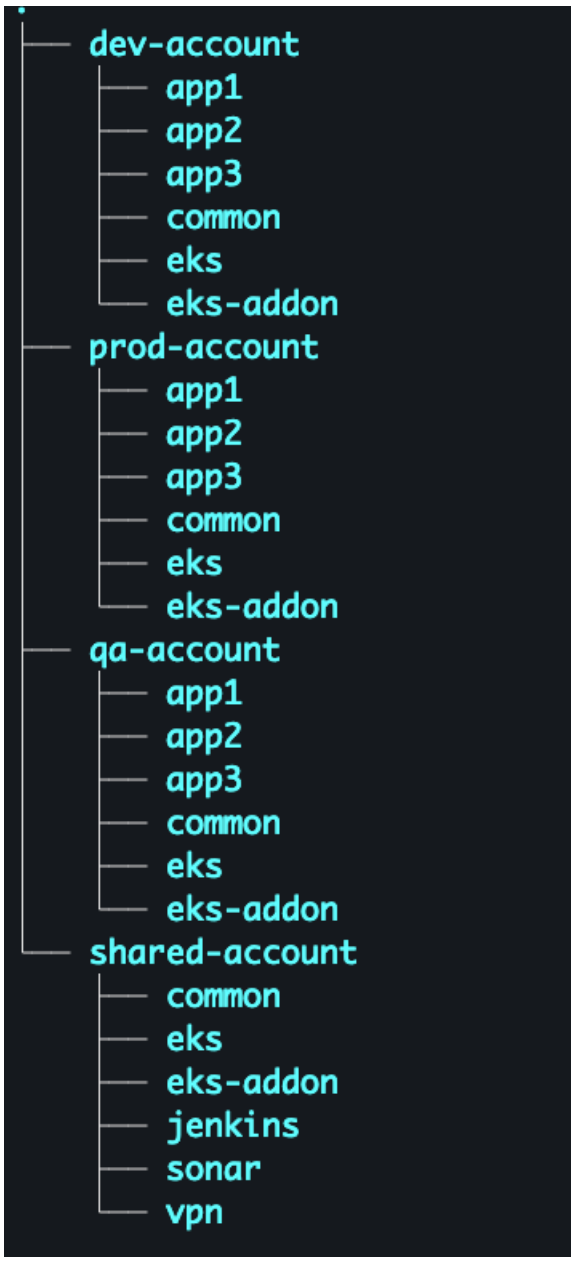
1. Never put all your eggs in one basket

Never put everything in a single place, you should not put your complete terraform code in a single place. Always try to distribute it across different directories. Let’s understand why we should not put everything in a single place.

1. **Increased execution time:** Terraform will take a lot of time to run the plan as every time it has to refresh the state for a lot of resources. And if you will keep fewer resources in your directory it will take less time.
2. **Reduced impact area:** Humans can make mistakes and if your code is split across multiple directories then you are accordingly reducing your impact area.
3. **Code Maintainability:** As your infrastructure expands, a single directory can become overwhelming and challenging to manage. Breaking the code into smaller, logical units, such as modules, makes it easier to understand and update specific components without affecting others.

So always try to break your code as much as possible. Let me tell you how I break my code. See the screenshot below of my sample directory structure.



- I have 3 separate accounts for my application environments like dev, qa, and prod. Along with that, I have one more shared environment on which I host my common like VPN, Jenkins, sonar, etc.
- Now inside every account, I have a directory with a named common in which I have to terraform resources like my VPC, VPC peering, TGW, NACL, Route53, etc. which are common to all applications.
- EKS and EKS-addons are also common but I created a separate directory for EKS and EKS-addons for two reasons:

1. *I don't want to put too much code inside a common directory.*
2. *There can be a few cases where I don't have to create a Kubernetes cluster. Hence I created EKS in a separate folder so that wherever EKS is not needed I will simply remove that folder.*

- Besides this, I have created a separate directory for each application. So that change in code of each app is completely Isolated and one is not impacting the other one

2. Use modules

Now we have already done a lot of segregation based on the folder. Now in order to avoid the code replication I try to use the modules as much as possible.

I am not a big fan of git submodules. Hence I keep the modules in the same Terraform directory inside a separate directory.

3. Keep your modules versioned

We always try not to change the modules too frequently but in real life, that is not possible. You cannot design a perfect module in one go. There may be scenarios when you will have to change the module.

Now these changes can be very simple and will not impact the already created resources. but sometimes these changes can be contract-breaking as well when already created resources will be impacted and they also need to be updated after updating the module.

So to make this change seamless we should always version our terraform modules.

When to change the module version?

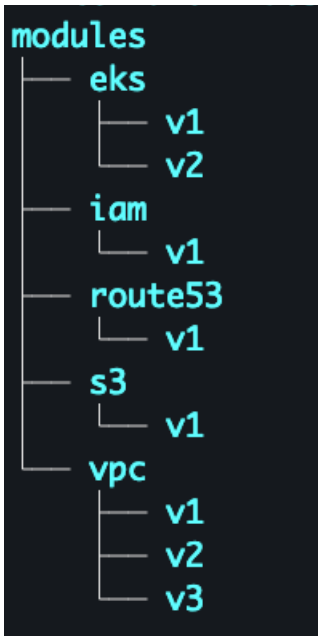
Whenever you are making any contract breaking which requires some code change in already created resources you must update the terraform version.

How to version the terraform modules?

This can be simply done using a sub-directory with the version name.

Example

Let’s take an example of the module directory structure below.



- I am creating the versions using the directories as shown in the above image.
- This way I can still keep using the older version of a module in some places where and can create a new version with some breaking changes. And then can upgrade the version of the older resources one by one.

4. Should we use Terraform workspaces?

I tried using Terraform workspaces a few times and the experience is not good. Below are a few reasons for it.

1. You need a proficient team that can work well with workspaces. If your team is not good enough working with modules then there are very high chances they will break the terraform code.
2. My all environments are not always Identical. In a few cases, I create a different kind of infra in dev and a different kind of infra in prod (due to some organization constraints), and handling these kinds of things via terraform workspaces can be tricky.
3. I still prefer breaking my workspaces based on the directory structure explained above. As I have complete control there.

Now I am leaving this thing open for now. You can let me know your experience working with Terraform workspaces.

5. Naming convention of Terraform resources

Always try to keep the name of resources as generic as possible. Because that gives me the flexibility to copy and paste my resource file anywhere. On the other hand, I always control the environment/application-specific naming convention via variable file. In that way, I know on which particular file I have to make the change. Rest all the files now just act like a template that can be added anywhere depending on the need. Let’s understand that better with an example.

Good example

Actual file which has the resource definition of my VPC

```
module "vpc" {
  source = "../modules/vpc/v1"
  name = local.name
  cidr = var.cidr
  tags = local.tags
  .
  .
  .
}
```

Variable file

```
### I am not defining any default value here as it will come from tfvars file
variable "env" {}
variable "name" {
  default = "myvpc" # I will define name of my vpc here.
}
variable "cidr" {
```

```
  default = "10.10.0.0/16" # I will define my vpc cidr here.
}
locals {
  name = "${var.name}-${var.env}"
  tags = {
    Name           = "${var.name}-${var.env}"
  }
}
```

Bad example

```
module "dev-vpc" {
  source = "../modules/vpc/v1"
  name = "myvpc-dev"
  cidr = "10.10.0.0/16"
  tags = {
    Name           = "myvpc-dev"
  }
  .
  .
  .
}
```

- Now if you see in **bad example** If I have to replicate the vpc for any other environment then after copying the tf file I still have to do a lot of changes to make it compatible with the new environment.
- Whereas In the case of **good example**, you can directly replicate the files to a new workspace and you know you only have to make changes in one place, and that is your variable file.
- These examples are just to demonstrate how you should be careful with terraform naming conventions. In actual scenario, I always try to use my TFvars file as much as possible keeping my variable file also generic and deriving everything from a TFvars file. Share your thoughts in the comment section if you want more details in this regard.

6. Where to define tags

I have divided the tags majorly into two categories mandatory and not mandatory. They can then further be divided into a few categories.

Mandatory Tags

- **Resource Specific Tags:** These are the tags that are specific to a resource and will differ resource by resource but are mandatory and must be added while resource creation. for eg.
- Let’s say I want to add a tag named **Consumer** in all my sqs queues so that I can know who is reading the messages from the queue. So this tag is only needed in case of any sqs queue other resources do not need to have it, And I want to make it mandatory as well.
- **How to define this kind of tag:** This tag should be defined inside the module and its value should be taken from a mandatory variable, Such that no one can avoid it. Let’s see the below example.

The file from where I am calling my module.

```
module "sqs" {
  source = "../modules/sqs/v1"
  name = local.name
  consumer = "some aplication name here"
  .
  .
  .
}
```

The variable file inside the module

```
.
.
variable "consumer" {} # this variable does not have any default value and is
mandatory.
.
.
```

Resource file inside the module

```
resource aws_sqs_queue "sqs" {
  .
  .
  tags = {
    Consumer = var.consumer
    .
    .
  }
}
```

If you look at the code. In the first place, I made a mandatory variable(by not specifying the default value) in the module named consumer. Now I am passing the same variable as a tag value inside the module which will make sure all my sqs have that tag.

- **Global Tags:** These are the tags that we want to be applied across all resources globally. and there are mandatory as well, We don’t want to miss them at any cost. for example Env tag. I want this tag to be added across all resources and value should be based on the respective environment. This can be added in a similar way as we added resource-specific mandatory tags. The only difference would be these have to be added across all resources.
- **Tags with auto-generated value:** As the says I want these tags to be added on each resource and its value will be auto-generated. for example, I want to add tags on each resource with the name **LaunchMonth**. I want to know in which month these resources are created and I can directly use it on the costing explorer. Let’s understand it with an example.

Resource file inside the module

```
resource aws_sqs_queue "sqs" {
  .
  .
  tags = {
    LaunchMonthYear = formatdate("MMM-YYYY", timestamp())
    .
    .
  }
  lifecycle {
    ignore_changes = [
      # Ignore changes to tags
      tags["LaunchMonthYear"]
    ]
```

```
  }
}
```

f you see the module code above. I have specified the value as well in it which will take a dynamic value every time I will apply terraform. I have also added a **lifecycle ignore_changes** block which will make sure the tags are only added at the time of resource creation and their value is not updated every time I am running Terraform.

Optional Tags

Optional tags can be passed at both the module level as well as resource level. Also, let's see after combining everything how it looks like.

The file from where I am calling my module.

```
module "sqs" {
  source = "../modules/sqs/v1"
  name = local.name
  consumer = "some aplication name here"
  env = var.env
  tags = {
    SomeOptionalTag = "SomeValue"
  }
  .
  .
  .
}
```

The variable file inside the module

```
.
.
variable "consumer" {} # this variable does not have any default value and is
mandatory.
variable "env" {} # this variable does not have any default value and is
mandatory.
variable "tags" {
  description = "A mapping of tags to assign to all resources"
  type        = map(string)
  default     = {} # this has a default value as empty hence not mandatoy to
provide a value
}
.
.
```

Resource file inside the module

```
resource aws_sqs_queue "sqs" {
  .
  .
  tags = merge (
    var.tags,
    { Consumer = var.consumer },
    { LaunchMonthYear = formatdate("MMM-YYYY", timestamp()) },
  )
  lifecycle {
    ignore_changes = [
      # Ignore changes to tags
      tags["LaunchMonthYear"]
    ]
```

```
  }
}
```

The above config has everything.

1. resource-specific mandatory tag `Consumer`
2. global tag with the autogenerated value `LaunchMonthYear`
3. optional tags `var.tags`

Now before implementing tags first, you should decide which tag falls in which category and then add it to the terraform code.