#devops #infrastructure-as-code #aws #pulumi

Structuring your Infrastructure as Code

Published Aug 17, 2023 by Lee Briggs

If you're thinking of migrating to another infrastructure as code tool (and why would you, everything is *great* in the IaC world now, right?!) you might find yourself asking yourself a fundamental question when you get started: how do I structure things in a way that scales well and stands the test of time?

There's no canonical answer. Everyone does things slightly different, and different tools have different ideas on the best way.

In my day to day role as a Solutions Engineer at Pulumi I get to answer this question a *lot*. Customers are migrating from other IaC tools and they want to take this opportunity to think about the way they'd like to structure things.

This blog post is designed to detail my high(ish) level thoughts on the concepts and principles I like to use, and why. As we explore these concepts, I'll talk about some of the lessons I learned from my time in configuration management and the myriad IaC tools I've used before today.

A lot of the concepts in this post are focused on Pulumi, but lots are broadly applicable to other tools.

Layers

I'm sure my system administrator background is showing, but I like to think about infrastructure through the concept of *layers* similar to the OSI Model. Most of the layers I'll outline here closely mirror the OSI model, but what you'll likely want to do before you create your Git repo or write a single line of code, is group your cloud infrastructure into layers. The reason why will become apparent later.

Layer 0: Billing

The billing layer is where you sign up or input your credit card. Each cloud provider does this differently

- AWS: Organization
- Azure: Account
- Google Cloud: Account

In my experience, while there's API for this stuff you likely *don't* want to manage this layer with IaC, so do yourself a favour and do it manually.

Layer 1: Privilege

The privilege layer is how you fundementally separate access in the cloud provider. Again, each provider does this a little differently.

Example Resources

- AWS: Account
- Azure: Subscription
- Google Cloud: Project

You *might* want to manage this layer with IaC, but you need to decide how that'd work. Personally, I find that the API level support for this layer and the rarity of needing to perform this operation means it's often easier to manage this layer manually.

Layer 2: Network

Now we're getting to the layers that'll should *definitely* be managed by IaC. The network layer is foundational to how everything will work in your infrastructure, and includes things like a VPC, subnets, NAT Gateways, VPNs, and anything else that facilitates network communication.

Example Resources

- AWS: VPCs, Subnets, Route Tables, Internet Gateways, NAT Gateways, VPNs
- Azure: Virtual Networks, Subnets, Route Tables, Internet Gateways, NAT Gateways, VPNs
- Google Cloud: VPCs, Subnets, Route Tables, Internet Gateways, Cloud Nat, VPNs

Layer 3: Permissions

Now we've laid down a network layer, we need to allow other people or applications to talk to the cloud provider API. IAM roles, or service principals live in this layer.

Example Resources

- AWS: IAM Roles, IAM Users, IAM Groups
- Azure: Service Principals, Managed Identities
- Google Cloud: Service Accounts

Layer 4: Data

The data layer is where the resources you're managing really start to open up. This is where you'll find things like databases, object stores, message queues, and anything else that's used to store or transfer data.

Example Resources

- AWS: RDS, DynamoDB, S3, SQS, SNS, Kinesis, Redshift, DocumentDB, ElastiCache, DynamoDB
- Azure: SQL, CosmosDB, Blob Storage, Queue Storage, Event Grid, Event Hubs, Service Bus, Redis Cache
- Google Cloud: Cloud SQL, Cloud Spanner, Cloud Storage, Cloud Pub/Sub, Cloud Datastore, Cloud Bigtable, Cloud Memorystore

Layer 5: Compute

The compute layer is where your applications actually run - this is where you'll find things like virtual machines, containers, and serverless functions.

Example Resources

- AWS: EC2, ECS, EKS, Fargate
- Azure: Virtual Machines, Container Instances, AKS
- Google Cloud: Compute Engine, GKE

Layer 6: Ingress

Layer 6 is where you'll find the resources that allow your applications to be accessed by the outside world.

Example Resources

- AWS: Application Load Balancers, Network Load Balancers, Classic Load Balancers, API Gateways
- Azure: Application Gateways, Load Balancers, API Management
- Google Cloud: Load Balancers, API Gateways

Layer 7: Application

Once we've provisioned all the supporting infrastructure, we now need to actually deploy the application itself. This is where things really get a little tricky and depend entirely on your application's deployment model, technology and architecture.

You might choose not to use IaC for application at all, but if you do..

Example Resources

- AWS: Lambda, ECS Tasks, Kubernetes Manifests, EC2 User Data
- Azure: Azure Functions, Kubernetes Manifests
- Google Cloud: Cloud Functions, Kubernetes Manifests

Visualization

If you're a visual learner like me, you might find this visualization helpful:

Layer	Name	Example Resources
0	Billing	AWS Organization/Azure Account/Google Cloud Account
1	Privilege	AWS Account/Azure Subscription/Google Cloud Project
2	Network	AWS VPC/Google Cloud VPC/Azure Virtual Network
3	Permissions	AWS IAM/Azure Managed Identity/Google Cloud Service Account
4	Data	AWS RDS/Azure Cosmos DB/Google Cloud SQL
5	Compute	AWS EC2/Azure Container Instances/GKE
6	Ingress	AWS ELB/Azure Load Balancer/Google Cloud Load Balancer
7	Application	Kubernetes Manifests/Azure Functions/ECS Taks/Google Cloud Functions

Principle 1: The Rate of Change

One of the difficult concepts to quantify when thinking of these layers is the *rate of change* that these resources undergo when you're managing them. The lower layers generally will change *less frequently* than your higher layers, and in addition to this these layers are generally most fraught with risk when changing them.

You might be wondering why this matter - the answer to this is because when structuring your IaC, you'll want to consider how you group resources together in your chosen IaC's encapsulation mechanism. For example, Pulumi uses the concept of *projects* to group resources together.

When creating and defining resources in a Pulumi project, the fundamental consideration you need think of when adding a resource is "which layer does this resource live in?". You generally shouldn't have resources from different layers in the same project, because the rate of change of those resources will be different and the risk of changing them is different.

Principle 2: Resource Lifecycle

> **Note:** This principle comes to you thanks to my wonderful colleague Ringo De Smet, who reminded me of the importance of breaking the rules when reviewing this post

As with all Principle in life, there are situations where principle 1 doesn't broadly apply.

There are resources within the above layers where you might think "ah! this is a network resources so I'll put it in my network project" but the *lifecycle* of the resource doesn't necessarily fit as a shared resource. A great example of this is an AWS security group.

Security groups are generally specific to another resource - perhaps an application you're deploying, a loadbalancer that's shared or maybe a database in the data layer. With these resources, it's generally best to consider the overall lifecycle of the dependent resources when deciding where to put it.

My rule of thumb here is this - if I wanted to provision this resource in a different environment, or better yet, destroy it - what other resources do I want to destroy at the same time?

Another great considering for this is the permissions layer. I already mentioned when discussing permissions that you'll need to think about that layer as *shared* permissions, application specific permissions are entirely different - they really want to go directly with your application deployment code.

The summary here is: don't be afraid to break the first principle, but make sure when you're doing it you're thinking about the resource lifecycle.

Principle 3: Repositories

The mono-repo vs multi-repo debate is one that's will rage long after we're all done with cloud computing and have migrated back to physical infrastructure, and I'm not going to try and solve it here. What I *will* say is that I've seen both work well, and both work poorly.

When it comes to IaC repositories, I again come back to our layering system and make differing decisions for where the code for deploying the repo should live is based on the layer.

The Control Repo

For the foundational, shared aspects of the infrastructure, I generally like to include those projects in a single mono-repo which back in my days of using Puppet we called a control repo. I still like to use this nomenclature.

If we refer back to our layers, I generally like to ensure layers 1 and 2 in this shared repo. Things get a *little* trickier once we get to layer 3, the permissions layer. At this stage, we need to decide if the resource itself is shared or not. A good example of this is an IAM role that might be used for *human* users instead of application user. This is generally going to be shared across multiple humans and teams, so it's a good candidate for the control repo.

Layer 4 really depends on your application architecture. If you have a message bus spanning multiple applications, putting it in your control repo probably makes sense, but if you have a database that is only used by a single application, you likely don't want it in the control repo for a variety of reasons.

Layer 5 again depends on your organisation's permission model and cloud architecture. It's not uncommon to share shared compute like an ECS cluster or Kubernetes cluster which spans many applications, so including it in an application repo probably isn't going to make much sense. However if you're isolating compute on a per-application basis, you're almost certainly going to want to make this application specific.

Layer 6: As it likely becoming an obvious trend, you'll need to take your application architecture and permission model into account. If you're using a shared load balancer and routing traffic that way, you'll likely want to include it in the control repo, but if you're using a per-application load balancer you'll want to include it in the application repo.

Application Repositories

At the very least, if you're using IaC to deploy your application, having a `deploy/` directory in your application repo is a great starting point. If you use Pulumi and want to use the same language as your application to do your deployments, you might consider having all of your dependencies in a single `package.json` or `requirements.txt` depending on your chosen language.

You'll need to think about the rate of change here when you're defining projects to group resources together. Do you perhaps need to separate your database layer and your application layer resources? I'd argue that you do, because the rate of change of your application layer is likely going to be much higher than your database layer, but you'll need to make a decision that makes sense for your organisation and project.

Why do this?

The primary reason for making the decision to use both mono-repos and keeping deployment code with applications is built from a perspective of *ownership* and *orchestration*.

Foundational infrastructure at layers 1, 2 and possibly up to layer 5 is an order of operations problem and a workflow orchestration problem. In most circumstances, you'll be creating resources that depend on *other* resources while building the IaC graph.

By deciding to break the resources into different projects, you can create a workflow that allows you to deploy the resources in the correct order. You'll be able to utilize Pulumi stack references to share resources between stacks and projects, but you'll need to ensure that a resource in a project in layer 2 that depends on a project in layer 1 has been created and resolved first.

In a mono-repo, this is as simple as ensuring that the workflow or CI/CD tool runs the projects in the correct order, but in a multi-repo implementation, it becomes a complex orchestration problem that likely involves multi repo webhooks and a lot of duct tape.

Application repos are far enough down the layering system that all of the infrastructure required to run your application will be in place. Placing application deployment infrastructure code in the application repo allows you to give the application developers full ownership of their code from writing and features to getting them into production.

Principle 3: Encapsulation

Once you've made the foundational decisions above, you'll be well on the way to structuring a well defined set of infrastructure as code patterns, but the final thing you'll need to consider is how you'll share resource patterns across your control repo and application repos.

Every IaC tool has a different way of managing this. In Pulumi you can create a Component Resource for a single language or if you want to support multiple language, you might want to create a Pulumi Package, but the reason for doing this is the same: you want to encapsulate a set of best practices that you can share across multiple projects.

A good consideration for for when to start encapsulating resources is to think about your organisational structure and application architecture. If you're only one team deploying a single application, you might not need to go down the path of encapsulating anything, but if you're a platform team that's likely to support dozens of teams to deploy to a shared layer 5 compute resource, creating a Pulumi package that encapsulates the best practices for deploying your application or creating a package for a best practice object storage bucket which has the required permissions is going to save the teams you're supporting a *lot* of time.

These encapsulations should be in their own, *distinct* repository. You'll want to version these encapsulations in the same way you version and release your applications - follow semver and make sure you create an API that your downstream users can use.

As your downstream users start to depend on these encapsulations, you can introduce concepts like unit testing to make sure you don't break userspace with your infrastructure.

Pitfalls

A common mistake I see at the encapsulation layer when adopting Pulumi is trying to avoid object orientated principles and using a what I like to call the "function based approach".

As an example of this, you might try and encapsulate some resources into a function. In TypeScript it'd look like this:

```
export function createBucket(name: string) {
    return new aws.s3.Bucket(name);
}
```

and in Python like so:

```
def create_bucket(name: str) -> aws.s3.Bucket:
    return aws.s3.Bucket(name)
```

The problem with this implementation of an abstraction is that it creates a nested mechanism that is difficult to manage successfully.

If you use a component, you get you get an abstraction mechanism that is much more native to the way the language works. In TypeScript, it looks like this:

```
export class Bucket extends pulumi.ComponentResource {
    public readonly bucket: aws.s3.Bucket;

    constructor(name: string, args: BucketArgs, opts?: pulumi.ComponentResourceOptions) {
        super("lbrlabs:index:Bucket", name, {}, opts);

        this.bucket = new aws.s3.Bucket(name, args, { parent: this });
    }
}
```

and in Python:

```
class Bucket(pulumi.ComponentResource):
    def __init__(self, name: str, args: BucketArgs, opts: Optional[pulumi.ResourceOptions]
= None):
        super().__init__("lbrlabs:index:Bucket", name, {}, opts)

        self.bucket = aws.s3.Bucket(name, args, parent=self)
```

While the instantiation of the resource is more complex, the manageability of this over time is exponentially easier. Trust me, I've untangled this mess before.

Putting it together

An example is worth a thousand words, so let's take a look at a hypothetical control repo and application repo.

Control Repo

Let's say we're going to be super original and call our repo `infrastructure`. Here's how that might look:

```
├── certs
│   ├── Pulumi.development.yaml
│   ├── Pulumi.production.yaml
│   ├── Pulumi.yaml
│   ├── __main__.py
│   ├── requirements.txt
│   └── venv
├── cluster
│   ├── Pulumi.development.yaml
│   ├── Pulumi.production.yaml
│   ├── Pulumi.yaml
│   ├── README.md
│   ├── __main__.py
│   ├── requirements.txt
│   └── venv
├── shared_database
│   ├── Pulumi.development.yaml
│   ├── Pulumi.production.yaml
│   ├── Pulumi.yaml
│   ├── __main__.py
│   ├── components
│   ├── requirements.txt
│   └── venv
├── domains
│   ├── Pulumi.development.yaml
│   ├── Pulumi.production.yaml
│   ├── Pulumi.yaml
│   ├── __main__.py
│   ├── requirements.txt
│   └── venv
├── cache
│   ├── Pulumi.development.yaml
│   ├── Pulumi.production.yaml
│   ├── Pulumi.yaml
│   ├── __main__.py
│   ├── requirements.txt
│   └── venv
├── shared_example_app
│   ├── Pulumi.development.yaml
│   ├── Pulumi.production.yaml
│   ├── Pulumi.yaml
│   ├── README.md
│   ├── __main__.py
│   ├── productionapp.py
│   ├── requirements.txt
│   └── venv
├── shared_bucket
│   ├── Pulumi.development.yaml
│   ├── Pulumi.production.yaml
│   ├── Pulumi.yaml
│   ├── __main__.py
│   ├── requirements.txt
│   └── venv
├── vpc
│   ├── Pulumi.development.yaml
│   ├── Pulumi.production.yaml
│   ├── Pulumi.yaml
│   ├── __main__.py
│   ├── requirements.txt
│   └── venv
└── vpn
    ├── Pulumi.development.yaml
    ├── Pulumi.production.yaml
    ├── Pulumi.yaml
    ├── __main__.py
    ├── requirements.txt
    └── venv
```

You can see here that we're using Pulumi stacks to target differing environments (in this case, development and production), and creating a new project for different layers and resources.

You'll likely also notice that I've been quite liberal with my use of directories for each set of services. I'm not grouping all of the network/layer 2 resources into a single project, however I'm following the layering principle by not grouping any resources from different layers into the same project.

You can definitely reduce the number of projects here (for example, you might choose to groups the VPC and VPN projects together in a `network` project) but I generally find that projects/directories are "free" and reducing the blast radius of changes makes people feel comfortable about contributing to these shared elements.

Application Repo

Once we get to our application repo, it's a lot harder to be prescriptive, but let's say we have a simple Go application called `example-app`. Here's how that might look:

```
.
├── Dockerfile
├── Makefile
├── docker-compose.yml
├── deploy
│   ├── Pulumi.development.yaml
│   ├── Pulumi.production.yaml
│   ├── Pulumi.yaml
│   └── main.go
├── go.mod
├── go.sum
├── main.go
├── readme.md
└── README.md
```

Hopefully this is fairly self explanatory, you've got your application and mechanisms for local development with a `Dockerfile` and `Makefile`, and we can put our Pulumi code in a `deploy/` directory.

Encapsulation Repo

Finally, let's take a look at an example encapsulation repo. These repos can be quite complex, so as an example, take a look at this Pulumi package which encapsulates some level compute here.

Conclusion

We've covered a lot of ground here, but hopefully this has given you some ideas on how you might want to structure your infrastructure as code. If you're using Pulumi, as with all my content, always open to hearing better ideas!

* * * * *

© 2021, Ritij Jain | Pudhina Fresh theme for Jekyll.