**Terraform vs. Pulumi Experiences**

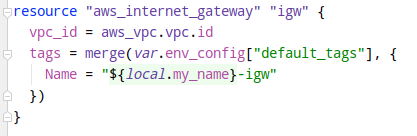
**Introduction**

In my new cloud project our DevOps team decided to create the similar AWS infra POC setup using two infrastructure as code tools: Terraform and Pulumi. The infra was pretty simple: VPC + some subnets + routing + security groups, EKS + node group, and RDS. It was an interesting exercise to implement the exact same infrastructure using two different infrastructure tools.

**Terraform**

[Terraform](https://www.terraform.io/) is an excellent declarative infrastructure as code tool that you can use with all major cloud providers (AWS, Azure and GCP). Terraform uses so called Hashicorp Configuration Language (hcl) to define the cloud resources and their dependencies. The latest 0.12 version brought quite a lot of enhancements to the hcl language.

Example using Terraform:

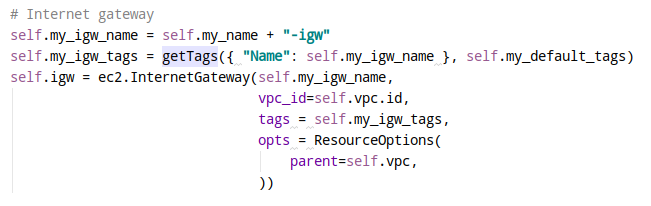


Terraform hcl example.

**Pulumi**

[Pulumi](https://www.pulumi.com/) can also be used with all major cloud providers. The major difference between Terraform and Pulumi is that with Pulumi you can use a real programming language (currently supports Javascript, Typescript and Python).

Example using Pulumi:



Pulumi Python example.

**Comparison**

**Configuration management**

Configuration management can be handled quite easily using both tools:

**Terraform**: Create a file in which you can give default values for each environment (e.g. RDS instance size…) as a map, and then provide other maps for each environment in which you can override the default values. Merge the maps — and you have a simple configuration management using Terraform.

**Pulumi**: You can create a file in which you list the default values for each environment and then a mechanism to read the environment specific YAML file and if the value is not found in the YAML file, use the default value.

**Winner**: tie.

**Modularity**

Modularity is simple using both tools:

**Terraform**: You can easily create re-usable Terraform modules and then create various configuration instances of those modules by injecting different values to the module.

**Pulumi**: You have the power of your favorite programming language to modularize your solution any way you want.

**Winner**: tie.

**Consistent Resource Naming**

I’m pretty stringent to name my cloud resources in a certain way: e.g. providing the prefix + environment in every resource name. This way I can easily search resources using either prefix (e.g. “SystemX”) or environment (e.g. “dev”) or the specific deployment (e.g. “SystemX-dev”).

**Terraform**: It is pretty easy to create a local variable for naming all entities for the current module and then use this local variable to concatenate consistent resource names for each resource entity.

**Pulumi**: Very straightforward since you can use a real programming language.

**Winner**: tie.

**Consistent Resource Tagging**

The same way as I’m pretty stringent with resource naming I want to be stringent with resource tagging as well. I want to add a set of default tags to all infra tool created resources + some extra tags for certain specific resources.

**Terraform**: Quite easy. You provide the default tags e.g. in common environment configuration. Then you merge default and custom tags when creating the actual resource (and provide merged tags).

**Pulumi**: Again very straightforward since you can use a real programming language.

**Winner**: tie.

**Language**

How powerful is the language? How easy it is to provide a declarative configuration of a cloud infrastructure? How easy it is to provide customization and conditional situations (e.g. create this entity only if condition X exists…)?

**Terraform**: Terraform is purely declarative language. That is its power and weakness. Terraform guides you to a certain declarative configuration which makes infrastructure code clean. On the other hand certain conditional situations are a bit clumsy (though usually possible).

**Pulumi**: You can use a real programming language — therefore the solutions often are as different as the developers creating those solutions. On the sunny side you don’t have to make various tricks to provide conditional situations as with Terraform but you can create what ever conditional situations using your favorite programming language.

**Winner**: Pulumi (but not much).

**IDE Support**

IDE support is excellent with both tools. E.g. you can get an excellent Terraform plugin for IntelliJ IDEA. With Pulumi you can use your favorite IDE with that programming language (e.g. Python — PyCharm).

**Winner**: tie.

**Deployment Process**

Can you see a plan phase before actual deployment — i.e. what resources are going to be added/modified/deleted? How complicated is the deployment process? What happens if something goes wrong?

**Terraform**: Terraform supports a plan phase. The actual deployment (apply) is 99% of the deployments pretty straightforward. Sometimes the deployment goes haywire (big time) — just hope it happened in one of the development environments in which the simplest solution is “just destroy it and create the environment from scratch”. Luckily this happens more often in development environment in which you experiment different things (add/remove/modify aggressively).

**Pulumi**: Pulumi also shows the changes and you can then confirm the changes or cancel.

**Conclusions**

Regarding these two short POCs I would say that both Terraform and Pulumi are excellent infrastructure as code tools — it is mostly a personal preference whether you want to stick with a stringent declarative tool (Terraform) or want to use your favorite programming language with more relaxed rules (Pulumi).

o create an EC2 instance using Pulumi, you'll first need to ensure you have the following prerequisites:

1. **Pulumi CLI**: Install the Pulumi CLI from Pulumi's installation guide.
2. **AWS Account**: You need to have an AWS account and set up your AWS credentials (e.g., using aws configure).
3. **Programming Language**: Choose a programming language supported by Pulumi (JavaScript, TypeScript, Python, Go, or C#).

Here's a step-by-step example using TypeScript:

**Step 1: Set Up Your Pulumi Project**

1. **Create a new directory for your project**:

bash

Copy code

mkdir my-ec2-project

cd my-ec2-project

1. **Initialize a new Pulumi project**:

bash

Copy code

pulumi new aws-typescript

Follow the prompts to set up your project.

**Step 2: Install AWS SDK**

Make sure you have the AWS SDK for TypeScript installed:

bash

Copy code

npm install @pulumi/aws

**Step 3: Create an EC2 Instance**

Open the index.ts file in your project and add the following code:

typescript

Copy code

import \* as pulumi from "@pulumi/pulumi";

import \* as aws from "@pulumi/aws";

// Create a new security group

const securityGroup = new aws.ec2.SecurityGroup("web-sec-group", {

ingress: [

{

fromPort: 80,

toPort: 80,

protocol: "tcp",

cidrBlocks: ["0.0.0.0/0"], // Allow HTTP traffic from anywhere

},

{

fromPort: 22,

toPort: 22,

protocol: "tcp",

cidrBlocks: ["0.0.0.0/0"], // Allow SSH traffic from anywhere

},

],

egress: [{

fromPort: 0,

toPort: 0,

protocol: "-1", // Allow all outbound traffic

cidrBlocks: ["0.0.0.0/0"],

}],

});

// Create an EC2 instance

const instance = new aws.ec2.Instance("web-server", {

ami: "ami-0c55b159cbfafe01f", // Replace with the latest Amazon Linux 2 AMI ID for your region

instanceType: "t2.micro",

securityGroups: [securityGroup.name], // Attach the security group

tags: {

Name: "MyWebServer",

},

});

// Export the public IP address of the instance

export const publicIp = instance.publicIp;

export const publicHostName = instance.publicDns;

**Step 4: Deploy the Infrastructure**

Now that you have your configuration set up, run the following command to deploy your infrastructure:

bash

Copy code

pulumi up

You'll see a preview of the changes that will be made. Type yes to confirm and proceed with the deployment.

**Step 5: Access Your EC2 Instance**

After the deployment completes, Pulumi will output the public IP and DNS of your instance. You can SSH into your instance using:

bash

Copy code

ssh -i path/to/your/key.pem ec2-user@<public-ip>

Make sure to replace <public-ip> with the actual IP address and provide the path to your SSH key.

**Cleanup**

When you’re done and want to remove the resources, run:

bash

Copy code

pulumi destroy

Then confirm the action.

**Notes**

* Adjust the AMI ID according to your AWS region and requirements.
* You may want to use a key pair for SSH access. Ensure you create an EC2 key pair in the AWS console or through Pulumi, if needed.