End to End Automation

Terraform state file

Terraform is a *stateful* application.

- Keeps track of everything you build inside of a state file
    - terraform.tfstate
- State file is another source of truth


```
{
  "version": 3,
  "terraform_version": "0.12.29",
  "serial": 6,
  "lineage": "0a209e29-de63-9e87-2cd2-4f2071717cee",
  "modules": [
      {
          "path": [
             "root"
           ],
           "outputs": {
              "MySQL_Server_FQDN": {
                 "value": "labtest1-mysql-server.azure.com
```

Terraform state file

Each Terraform configuration can specify a backend, which defines where and how operations are performed, where state snapshots are stored, etc.

Terraform state file

When starting out with Terraform the best approach is to stick with a local backend. It also makes sense to use a local backend if you are the only one managing the infrastructure.

Terraform state file

Backend configuration is only used by Terraform CLI.

Terraform Cloud and Terraform Enterprise always use their own state storage when performing Terraform runs, so they ignore any backend block in the configuration.

It's common to use Terraform CLI with Terraform Cloud, so best practice is to include a backend block in the configuration with a remote backend pointing to the relevant Terraform Cloud workspace(s)

Terraform state file

Terraform supports local and remote state file storage.

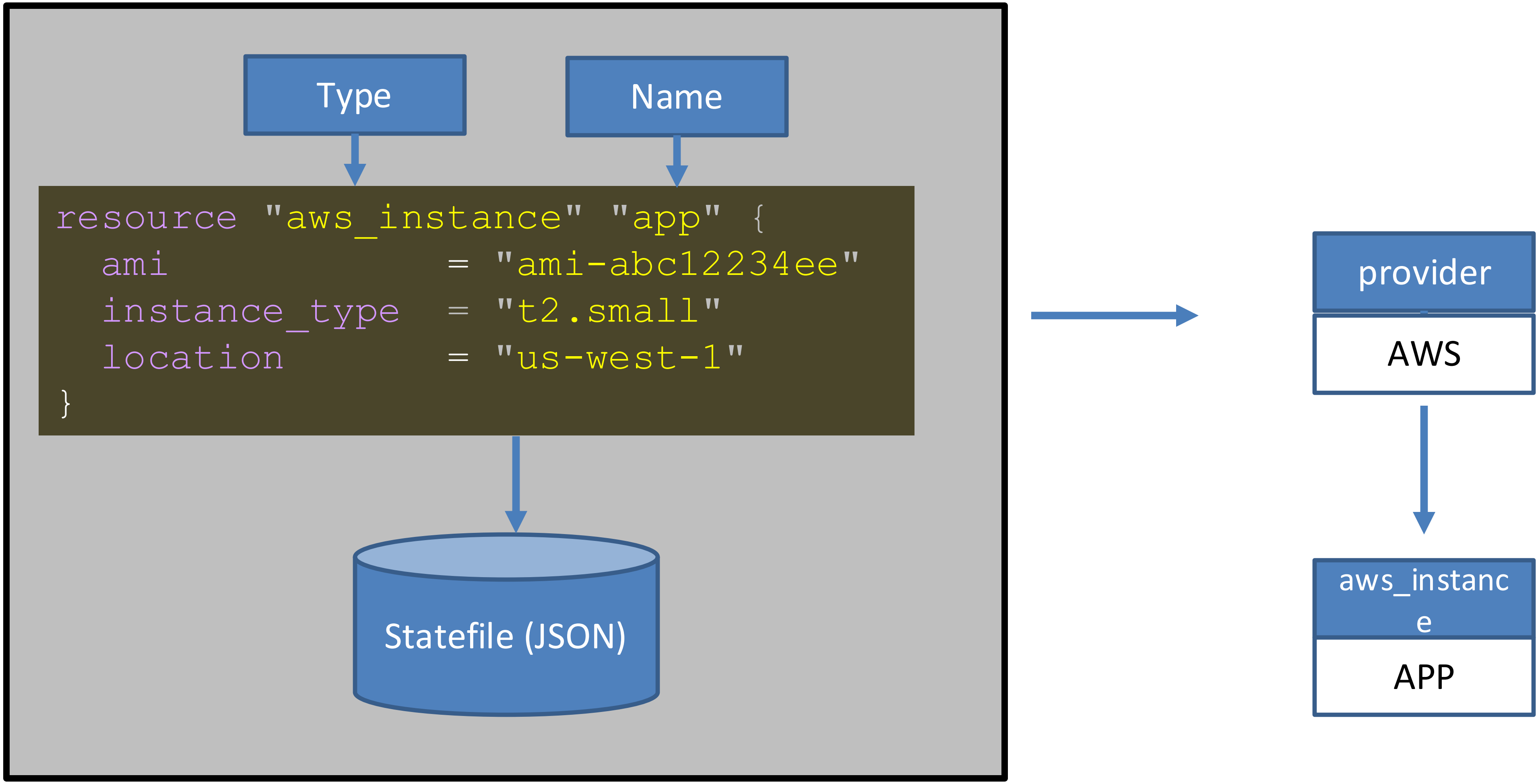
- Default is local storage
- Remote backends:
  - S3, Azure Storage, Google Cloud Storage

Terraform state file

Terraform supports many backends:

- local (default)
- remote (Terraform Enterprise/Terraform Cloud)
- azurerm
- consul
- s3
- gcs
- more...

Terraform state (local)

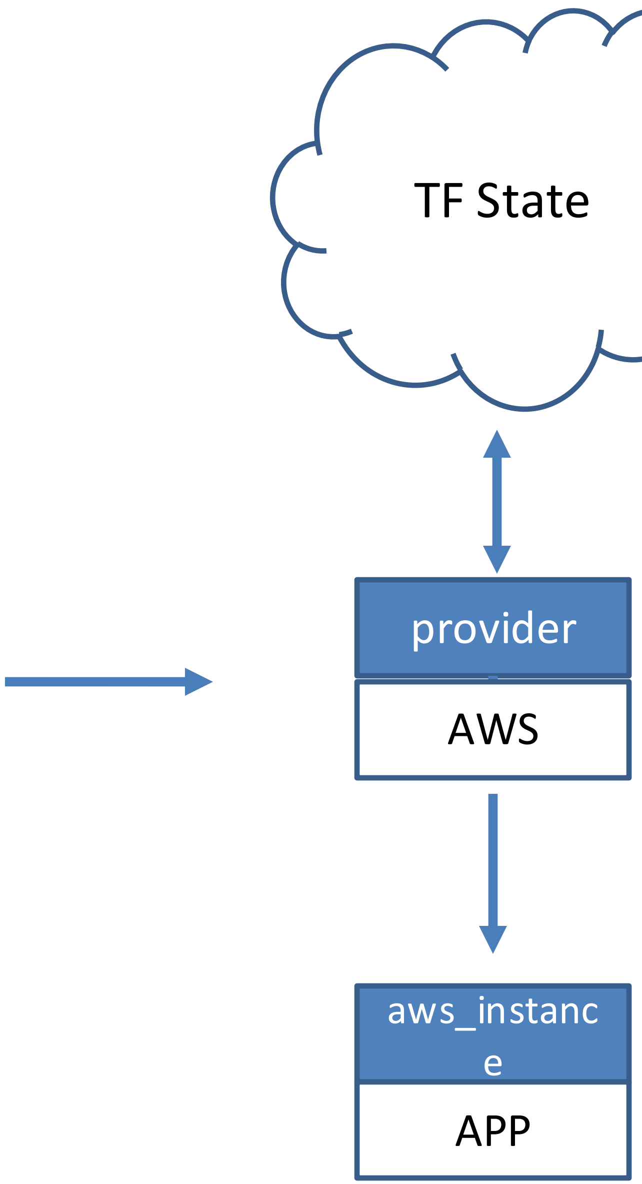


Terraform state (remote)

Type

Name

```
resource "aws_instance" "app" {
  ami             = "ami-abc12234ee"
  instance_type   = "t2.small"
  location        = "us-west-1"
}
```



Terraform state file locking

Terraform will lock your state for all operations that could write state if your backend supports it. This prevents others from acquiring the lock and possibly corrupting your state.

State locking automatically happens on all operations that write state.

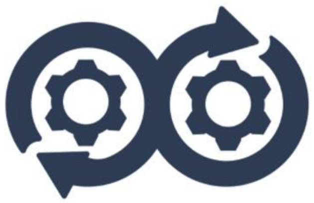
- azurerm
- consul
- etcdv3
- gcs
- s3 (locking via DynamoDB)

Terraform state file

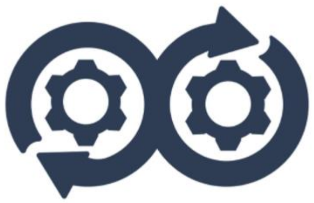
Backends are configured with a nested "backend" block within the top-level terraform block.

```
terraform {
  backend "remote" {
    organization            = "foo"
}
```

Lab: Implement bool, strings and numbers



Lab: Migrate to remote state backend

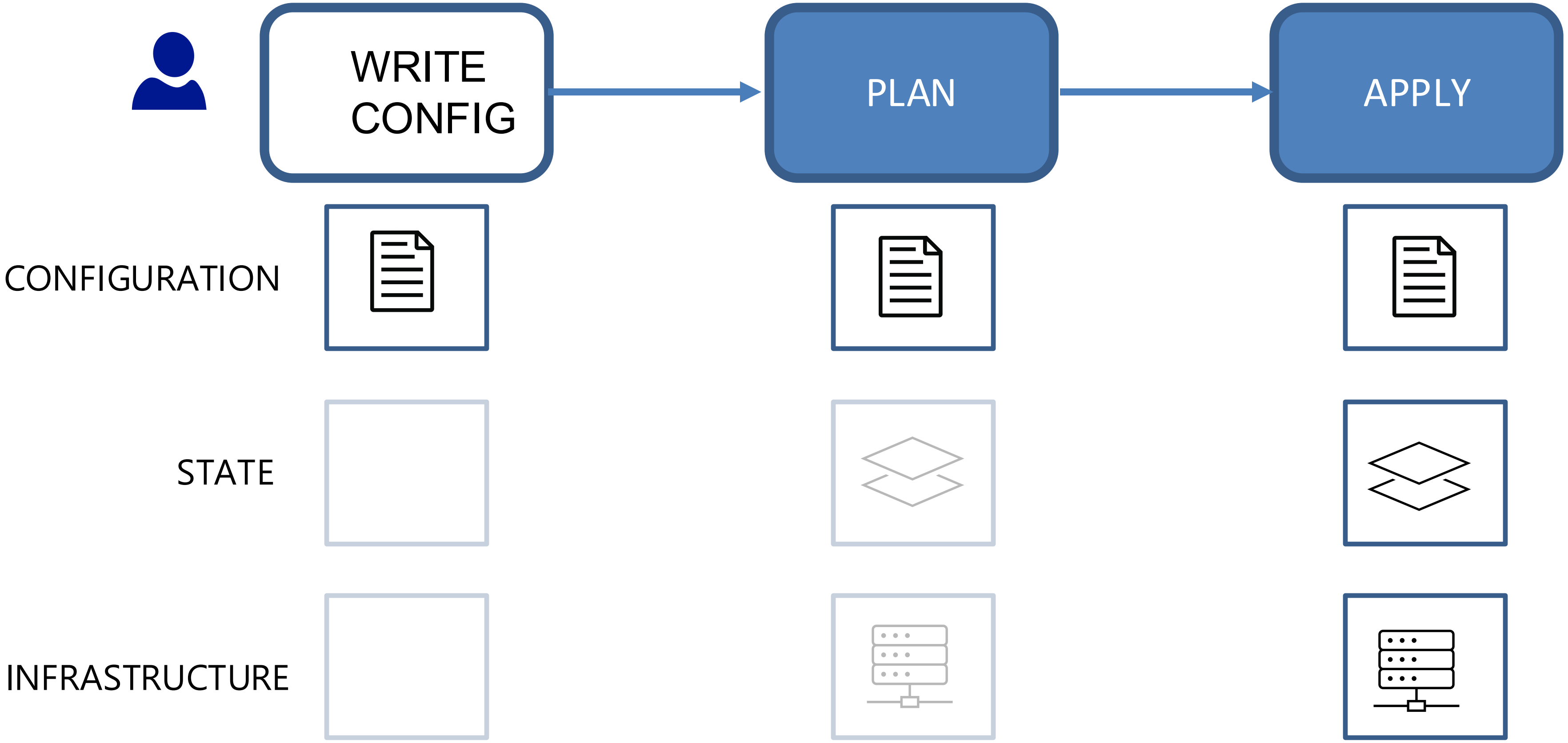


Terraform workflow

The default Terraform workflow involves creating and managing infrastructure entirely with Terraform.

1. Write Terraform configuration that defines the infrastructure you want to create.
2. Review the Terraform plan to ensure it will provision expected infrastructure.
3. Apply the config to create your state and infrastructure.

Terraform workflow



Terraform import existing infrastructure

Terraform is designed to manage new infrastructure, but it also supports importing existing infrastructure. This allows you to take resources that have been created by other means and bring it under Terraform management.

This is a first step in transitioning to Infrastructure-as-Code.

Terraform import existing infrastructure

Currently, Terraform can only import resources into the state. It does not generate a configuration. In the future, Terraform will support config generation.

It is necessary to manually write a resource configuration block for the imported resources.

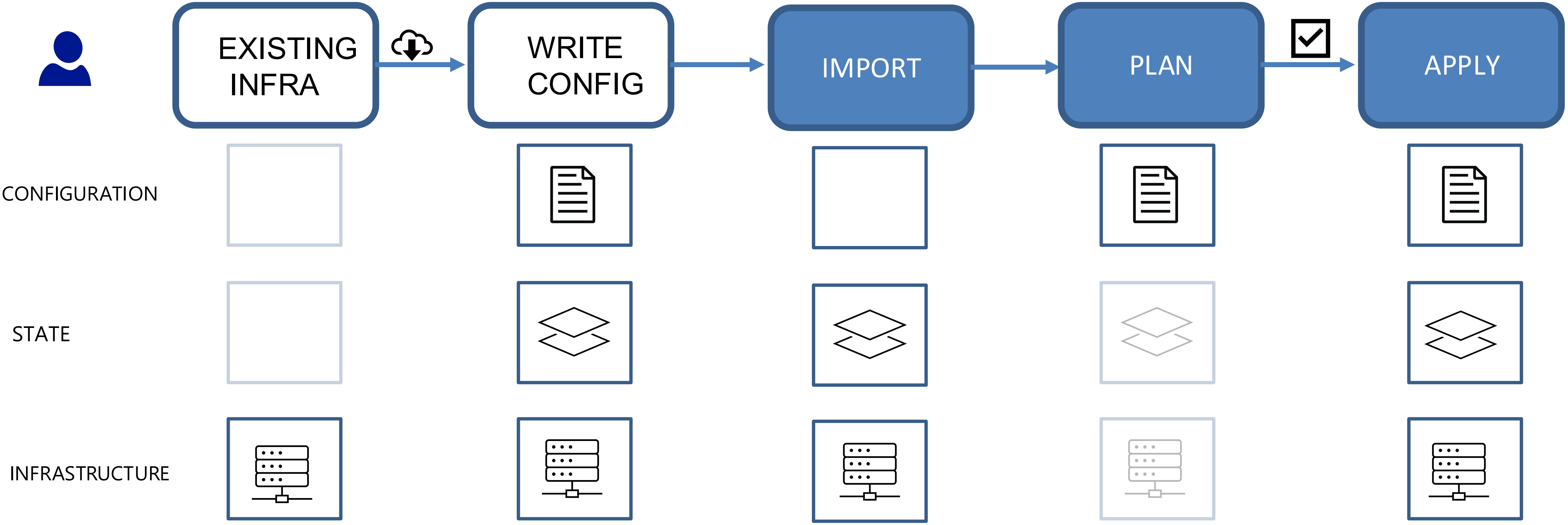
While this may seem tedious, it still gives Terraform users an avenue for importing existing resources.

Terraform workflow

Importing existing infrastructure for management with Terraform.

1. Identify the existing infrastructure to be imported.
2. Write Terraform configuration that matches infrastructure.
3. Import infrastructure into your Terraform state.
4. Review the Terraform plan to ensure the config matches.
5. Apply the configuration to update your Terraform state.

Terraform import workflow



Lab: Import existing resources

Terraform provisioners

Terraform is designed to programmatically create and manage infrastructure. It is not intended to replace Ansible, Chef or any other configuration management tool.

It includes provisioners to execute tasks.

Generic provisioners:

- file
- local-exec
- remote-exec

Terraform provisioners

Generic Provisioners:

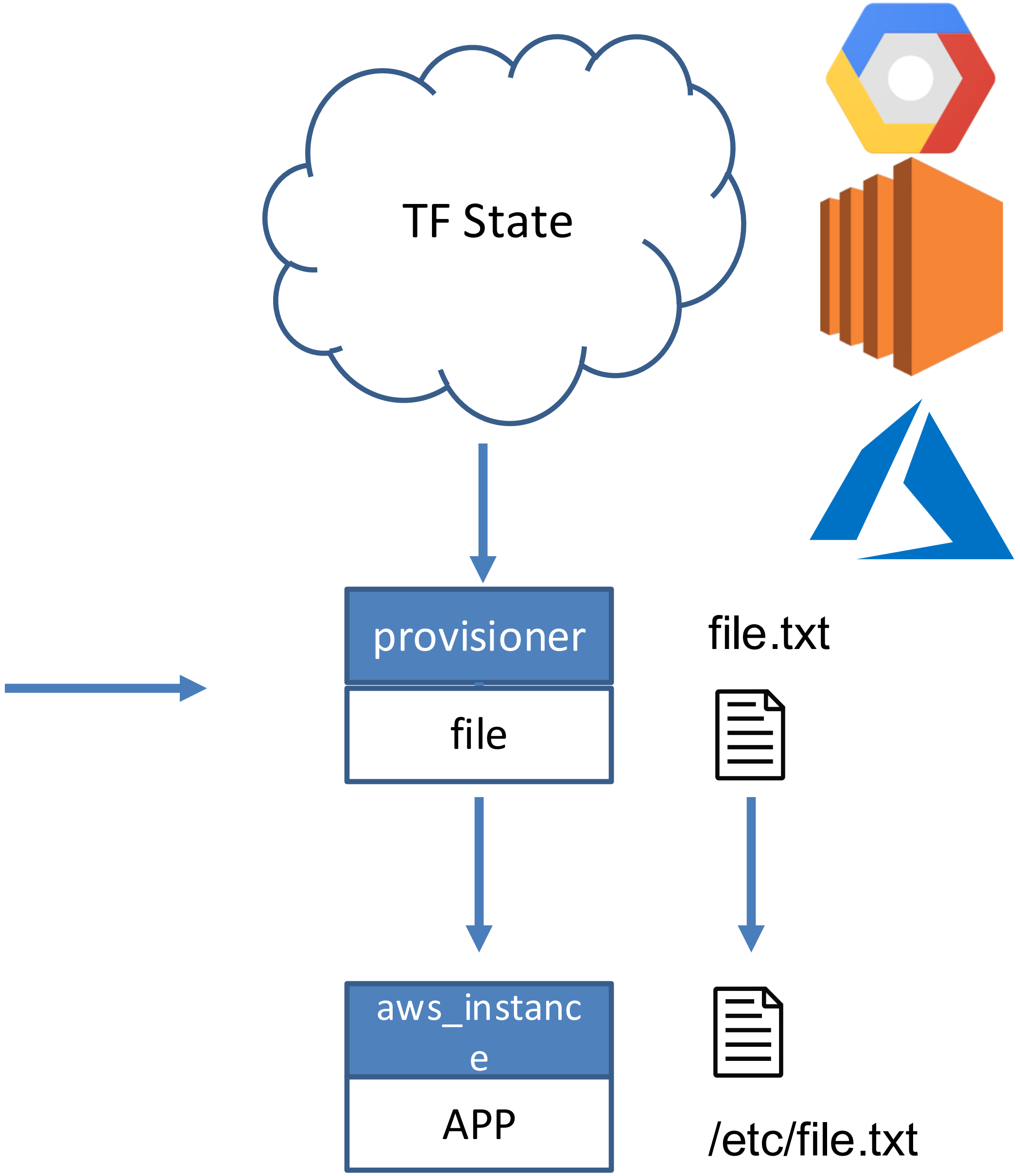
- file
  - The file provisioner is used to copy files or directories from the machine executing Terraform to the newly created resource. The file provisioner supports both SSH and WinRM type connections.
- local-exec
  - The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource.
- remote-exec
  - The remote-exec provisioner invokes a script on a remote resource after it is created. This can be used to run a configuration management tool, bootstrap into a cluster, etc. The remote-exec provisioner supports both SSH and WinRM type connections.

Terraform provisioners

Type

Name

```
resource "aws_instance" "app" {
  #...
# copies the file.txt to /etc/file.txt
  provisioner "file" {
    source      = "file.txt"
    destination = "/etc/file.txt"
  }
}
```

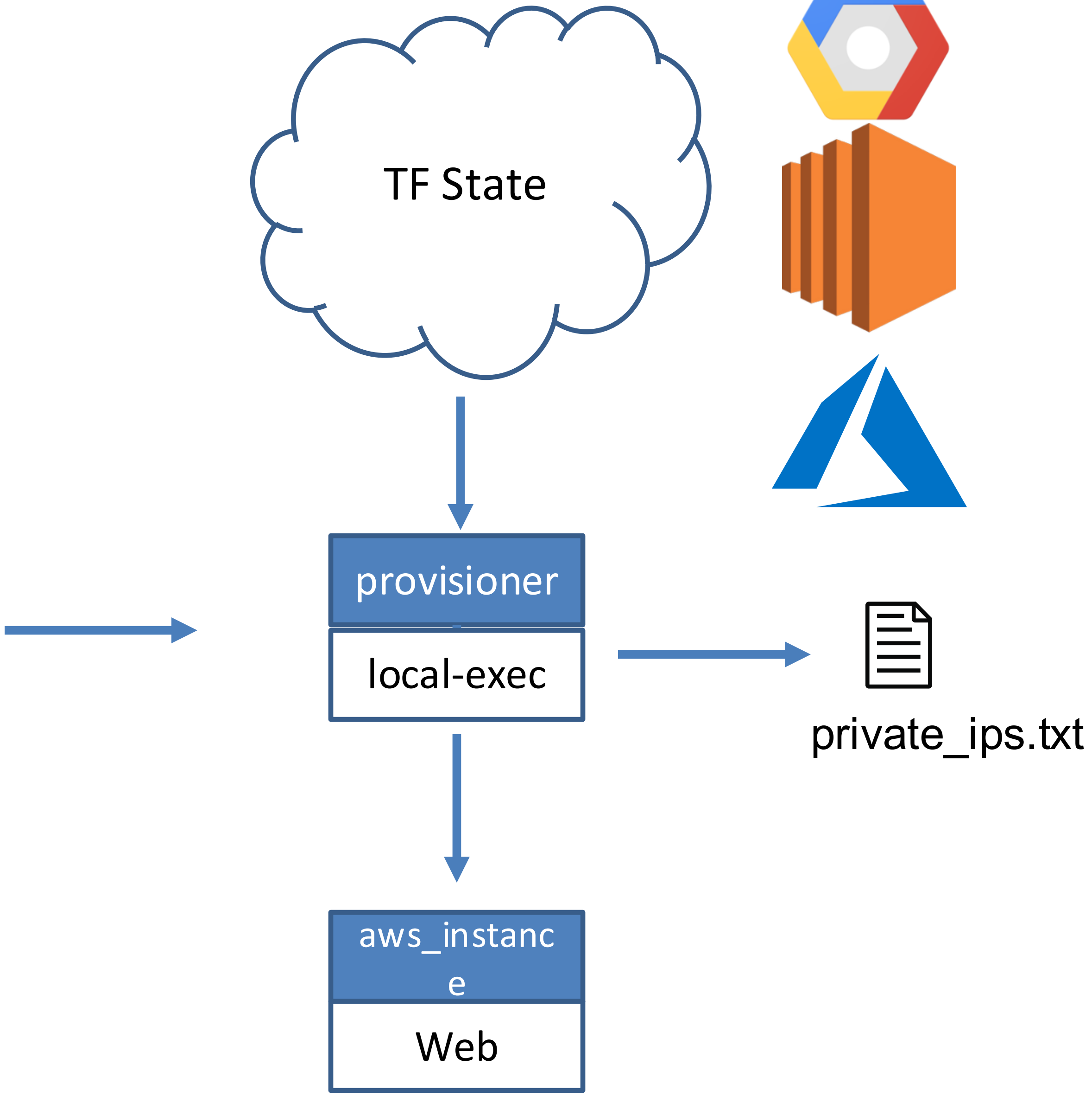


Terraform provisioners

Type

Name

```
resource "aws_instance" "web" {
  #...
# populate an inventory file
  provisioner "local-exec" {
    command = "echo ${self.private_ip}
>> private_ips.txt"     "
  }
}
```

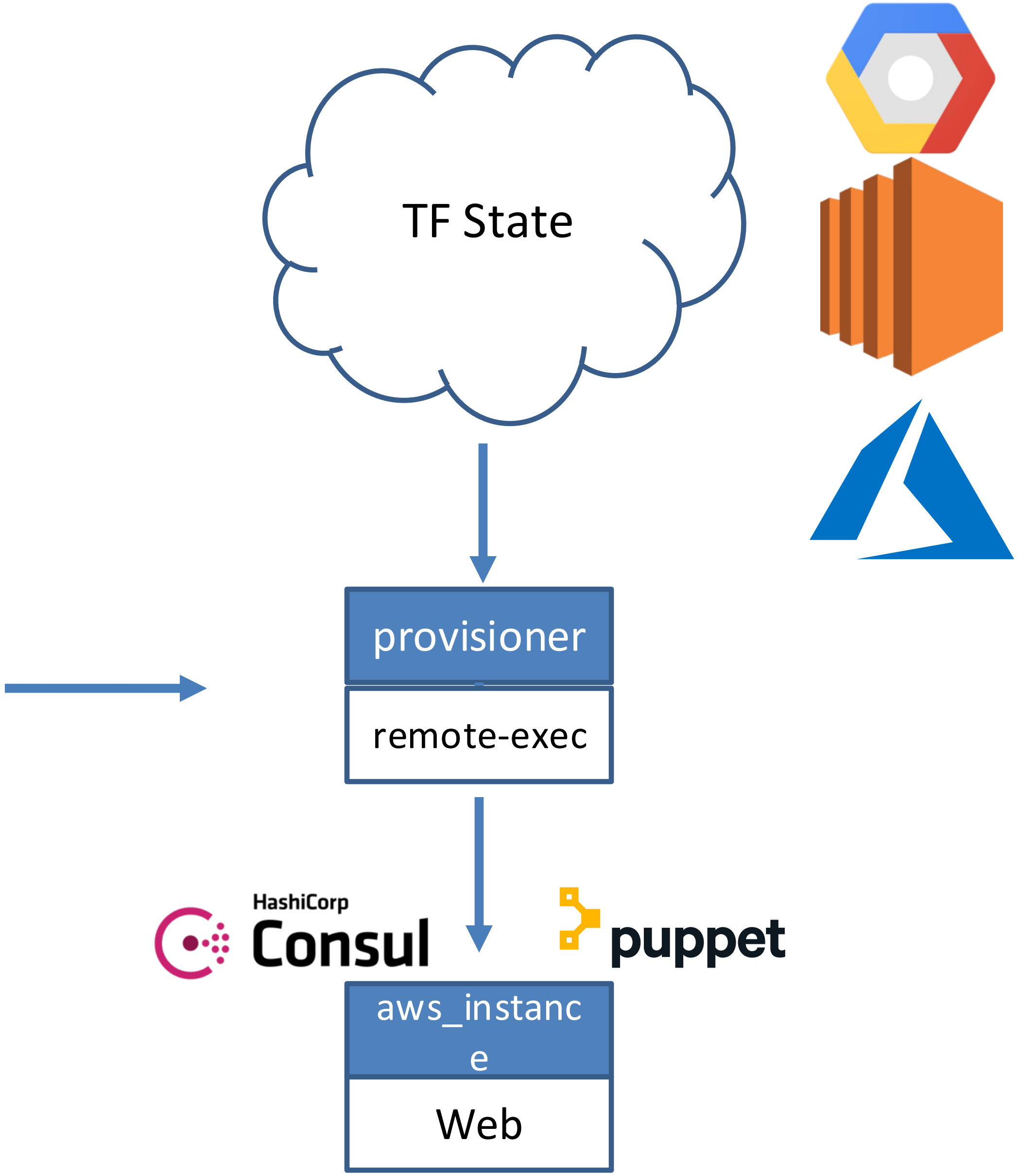


Terraform provisioners

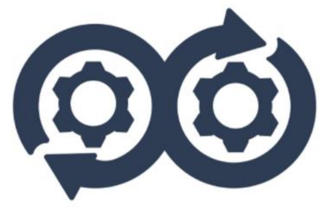
Type

Name

```
resource "aws_instance" "web" {
  #...
# populate an inventory file
  provisioner "remote-exec" {
    inline = [
      "puppet apply",
      "consul join \
      ${self.private.ip} >>
private_ips.txt"",
    ]
  }
}
```



Lab: Provisioners



Don't Repeat Yourself (DRY)

DRY is a principle that discourages repetition, and encourages modularization, abstraction, and code reuse. Applying it to Terraform, using modules is a big step in the right direction.

However, repetitions still happen. You may end up having virtually the same code in different environments, and when you need to make one change, you have to make that change many times.

Don't Repeat Yourself (DRY)

This problem can be addressed in a few ways. A recommended approach is to create a folder for shared or common files and then create symlinks to these files from each environment. This way, you can make a change to the common file(s) once and it is applied in all the environments.

Don't Repeat Yourself (DRY)

There is also an open-source tool, Terragrunt, which solves the same problem differently. It is a wrapper around the Terraform CLI commands, which allows you to write your Terraform once and then, in a separate repository, define only input variables for each environment - no need to repeat Terraform code for each environment. Terragrunt is also handy for orchestrating Terraform in CICD pipelines for multiple separate projects.

Don't Repeat Yourself (DRY)

Common directory structure for managing three environments (prod, qa, stage) with the same infrastructure (an app, a MySQL database and a VPC)

```
└── live
    ├── prod
    │   ├── app
    │   │   └── main.tf
    │   ├── mysql
    │   │   └── main.tf
    │   └── vpc
    │       └── main.tf
    ├── qa
    │   ├── app
    │   │   └── main.tf
    │   ├── mysql
    │   │   └── main.tf
    │   └── vpc
    │       └── main.tf
    └── stage
        ├── app
        │   └── main.tf
        ├── mysql
        │   └── main.tf
        └── vpc
            └── main.tf
```

Don't Repeat Yourself (DRY)

The contents of each environment will be almost identical, except for perhaps a few settings (e.g. the prod environment may run bigger or more servers). As the size of the infrastructure grows, having to maintain all this duplicated code becomes more error prone.

Modules

Modules are Terraforms way of managing multiple resources that are used together. A module consists of a collection of `.tf` files kept together in a directory.

Modules

As you use Terraform to manage your infrastructure, you will create increasingly complex configurations.

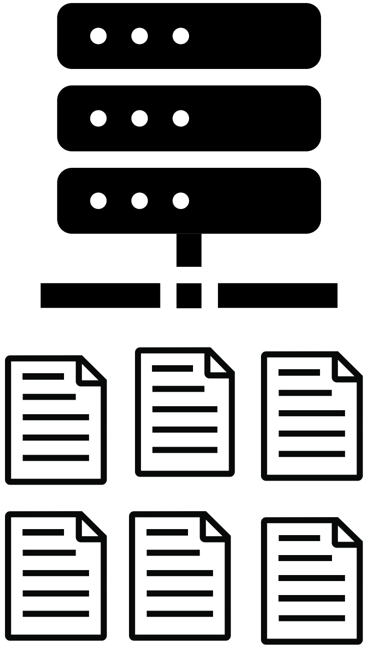
There is no limit to the complexity or size of a single Terraform configuration file or directory, so it's possible to include everything in one directory or even one file.

Modules

If you take a "monolithic" approach, and include all configuration in one file or directory you may run into these problems:

- Understanding and navigating the configuration becomes difficult.
- Making an update to one section may cause unintended consequences in other sections.
- There will be increasing duplication of similar blocks of configuration!
    - Dev, Staging, Production

Modules

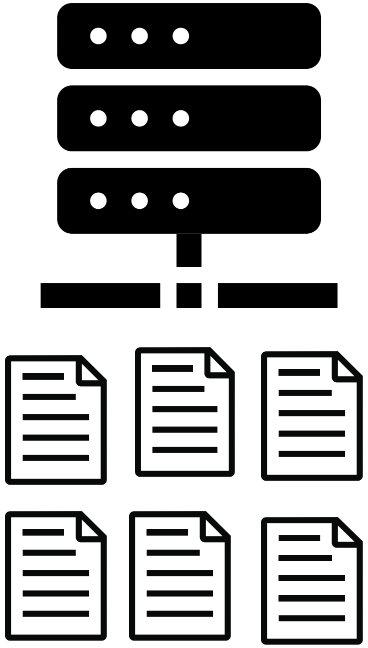


Dev

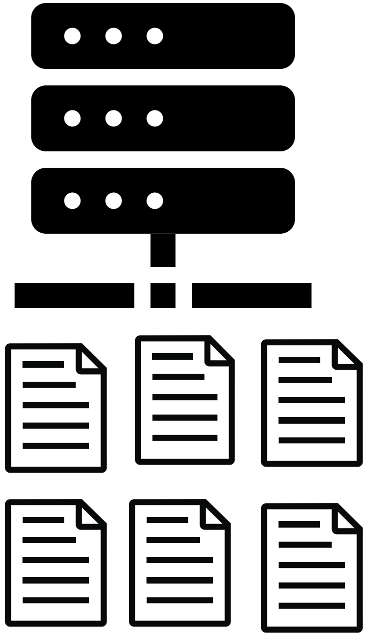
One environment can have many configuration files:

```
- main.tf
- variables.tf
- backend.tf
- provider.tf
- myvariables.tfvar
- outputs.tf
```

Modules

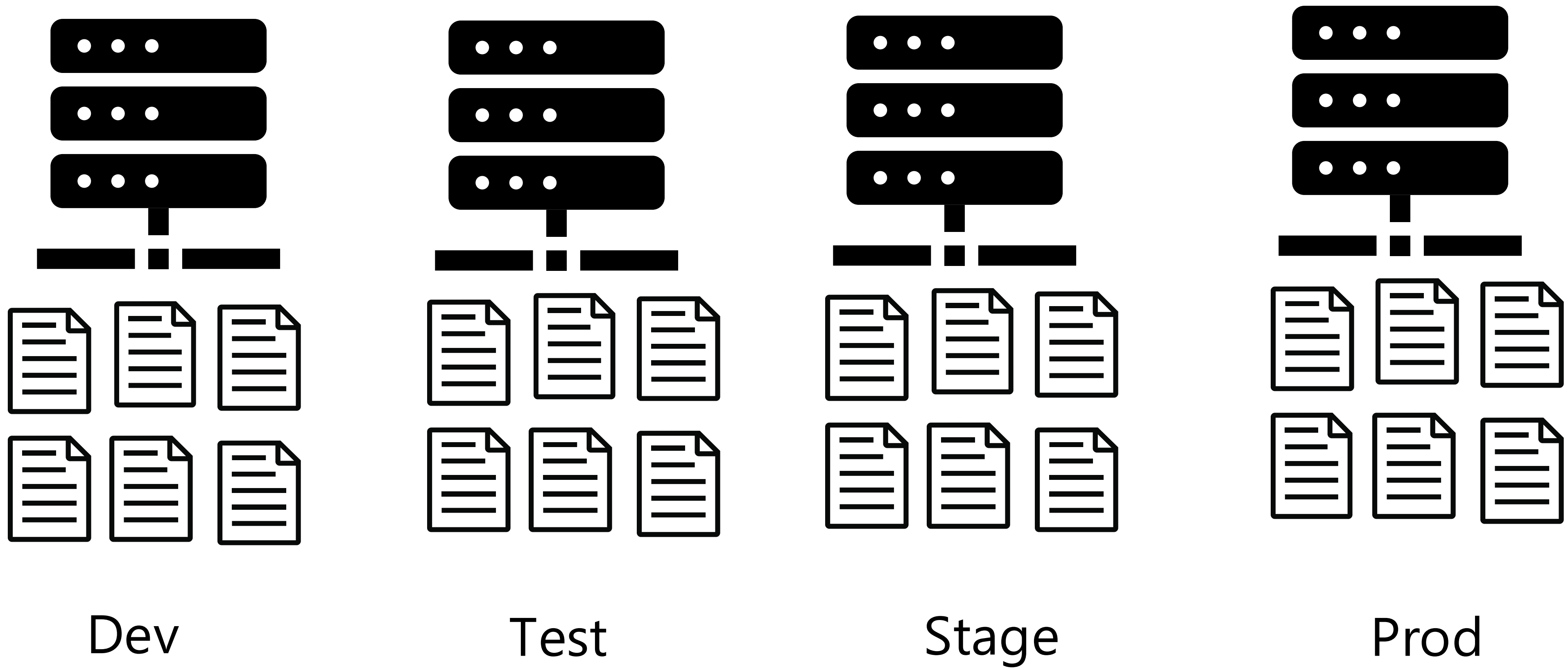


Dev



Test

Modules



Modules

How modules help:

- Organize configuration - Modules make it easier to navigate, understand and update your configuration by keeping related parts of the configuration together.

- Group the configuration into logical components.

Modules

- Encapsulate configuration - Modules encapsulate the configuration into distinct logical components. This prevents unintended consequences, such as a change in one part affecting other parts of the infrastructure.
- Reduces simple errors like using the same name for two different resources.

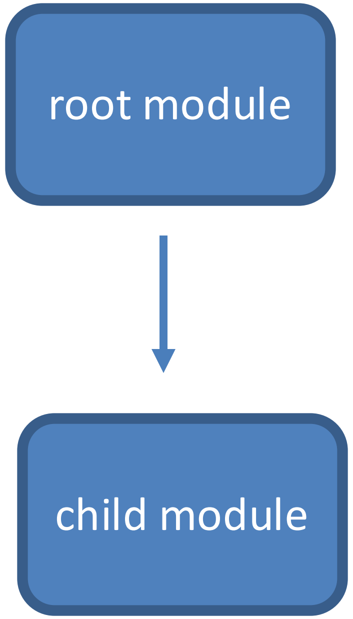
Modules

- Re-use configuration - Writing all configuration from scratch is time consuming and error prone. Modules save time and reduce errors by re-using configuration written by you, other members of your team, or experts who published modules in the registry.
- Save time!
  - Don't reinvent the wheel

Modules

- Consistency - Modules help provide consistency in your configurations. Consistency is important for readability of configuration, but also helps to ensure best practices are applied to all the configuration.
- Writing configuration from scratch can lead to security issues by having misconfigured attributes:
  - S3, Google Cloud Buckets
  - Security groups, old AMIs

Modules



- Provided by you:
- main.tf
- variables.tf
- outputs.tf

- Provided by module:
- resources
    - aws_instance
    - elb
    - ebs
    - ...

Terraform commands will only directly use the configuration files in one directory, which is usually the current working directory. However, your configuration can use module blocks to call modules in other directories. When Terraform encounters a module block, it loads and processes that module's configuration files.

Modules

- Modules can be loaded from a local directory, or from a remote source.
  - GitHub, Private registry, Public registry
  - The Terraform public registry includes:
    - Official modules: maintained by HashiCorp employees
    - Partner: maintained by vendors
    - Community: maintained by community volunteers.

Modules

- Best practices
  - Start writing with modules in mind. Even for modestly complex configurations managed by a single person, you'll find the many benefits of using modules outweigh the time it takes to use them properly.
  - Use the public Terraform Registry to find useful modules (if company policy allows it).
    - No need to reinvent the wheel
    - Reduce burden of writing/maintaining configuration from scratch.

Modules

- Best practices
  - Publish and share modules with your team! Infrastructure is managed by a team of people, and modules are an important way teams can work together to create and maintain infrastructure.

Lab: Modules

Terraform security

Infrastructure requires "secrets"
- Common secrets are:
  - API keys
  - Username/Password
  - SSH/TLS
  - Service Account files

Terrraform resources require credentials to authenticate with providers, resources, etc.

Terraform security

```
resource "aws_db_instance" "example" {
  engine                       = "mysql"
  engine_version               = "5.7"
  instance_class               = "db.t2.micro"
  name                         = "example"
  username                     = ....
  password                     = ....
}
```

POP QUIZ:

How can we provide credentials securely?

POP QUIZ:

How can we provide credentials securely?

- Environment variables
- Encrypted files
- HashiCorp Vault, Cloud Secret Managers, CyberArk

Terraform security

Do NOT store credentials in plain text!

- Anyone who has access to version control has access to secrets
- Every device that ever checked out the repo now has a copy of credentials in plain text
- Every piece of software running on these devices has access to secrets.
- No way to audit or revoke access to secrets

Terraform security

Keep Terraform state secure

The Terraform state file is plain text and holds all of the values for your resources, including secrets!

How you secure your secrets is irrelevant if the state file is insecure.

Store your state file in a remote backend that supports encryption in-transit and at-rest.

- S3
- GCS
- Azure
- more…

Terraform security

Techniques for managing secrets:

- Environment Variables
- Encrypted files (PGP, KMS, SOPS)
- Secret Stores (Vault, AWS Secrets manager)

Terraform security

Technique #1: Environment Variables

This first technique keeps plain text secrets out of your code by taking advantage of Terraforms native support for reading environment variables.

To use this technique, declare variables for the secrets you wish to pass in.

Terraform security

```
variable "username" {
  description           = "DB master username"
  type                  =  string
}

variable "password" {
  description           = "DB master password"
  type                  =  string
}
```

Terraform security

Terraform can mark variables sensitive, which masks them in output/logs.

```
variable "username" {
  description               = "DB master username"
  type                      =  string
  sensitive                 =  true
}

variable "password" {
  description               = "DB master password"
  type                      =  string
  sensitive                 =  true
}
```

Terraform security

Pass variables to the Terraform resource.

```
resource "aws_db_instance" "example" {
  engine                = "mysql"
  engine_version        = "5.7"
  instance_class        = "db.t2.micro"
  name                  = "example"
  username              = var.username
  password              = var.password
}
```

Terraform security

You can pass in a value for each variable by setting environment variables:

```
export TF_VAR_username=(the username)
export TF_VAR_password=(the password)
```

Terraform security

This technique helps you avoid storing secrets in plain text in your code, but it leaves the question of how to securely store and manage the secrets unanswered.

Terraform security

Technique #2 `Encrypted Files`

The second technique relies on encrypting the secrets, storing the cipher text in a file, and checking that file into version control.

- AWS KMS
- GCP KMS
- Azure Key Vault

Terraform security

To encrypt some data, such as some secrets in a file, you need an encryption key. This key is itself a secret!
This creates a bit of a problem:
how do you securely store that key? You can't check the key into version control as plain text, as then there's no point of encrypting anything with it.

Terraform security

You could encrypt the key with another key, but then you have to figure out where to store that second key. So, you still must find a secure way to store your encryption key.

Terraform security

AWS KMS example:

```
#db-creds.yml
username: admin
password: password
```

Now encrypt `db-creds.yml` using:
`aws kms encrypt`

A new file `db-creds.yml.encrypted` is created. You can safely check this file into version control.

Terraform security

To decrypt the secrets from `db-creds.yml.encrypted` in your Terraform config, you can use aws_kms_secrets data source.

```
data "aws_kms_secrets" "creds" {
  secret {
    name          = "db"
    payload       = file("${path.module}/db-creds.yml.encrypted"
}
```

Terraform security

Pass unencrypted values to Terraform resource.

```
resource "aws_db_instance" "example" {
  #...
  name                = "example"
  username            = data.aws_kms_secrets.example.plaintext["master_username"]
  password            = data.aws_kms_secrets.example.plaintext["master_password"]
}
```

Terraform security

Technique #3: Secret stores (Vault, AWS Secrets Manager)

The third technique relies on storing your secrets in a dedicated secret store: that is, a database that is designed specifically for securely storing sensitive data and tightly controlling access to it.

- Vault
- AWS Secrets Manager (ASM)
- GCP Secret Manager (GSM)

Terraform security

Create a secret using:
- AWS Console/CLI tools
- Vault
- GSM Console/CLI tools
- etc.

Terraform security

After creating our secrets, we can read them using a `data source` and declare them for our resource

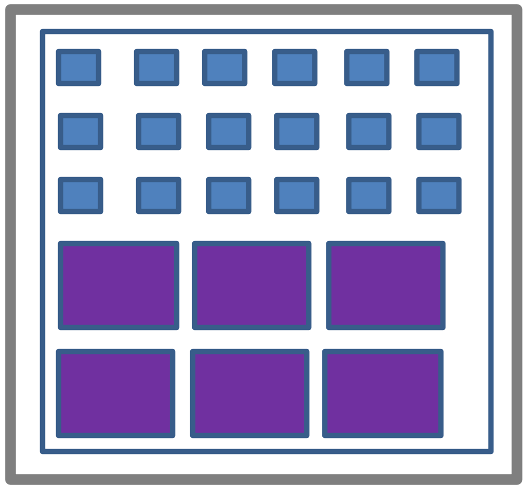
```
data "vault_kv_secret_v2" "example" {
  mount = "kv"
  name  = "test-secret"
}

resource "aws_instance" "example" {
  ami           = "ami-0c7217cdde317cfec"
  instance_type = "t2.micro"

  tags = {
    secret = data.vault_kv_secret_v2.example.data["username"]
  }
}
```

Discussion:

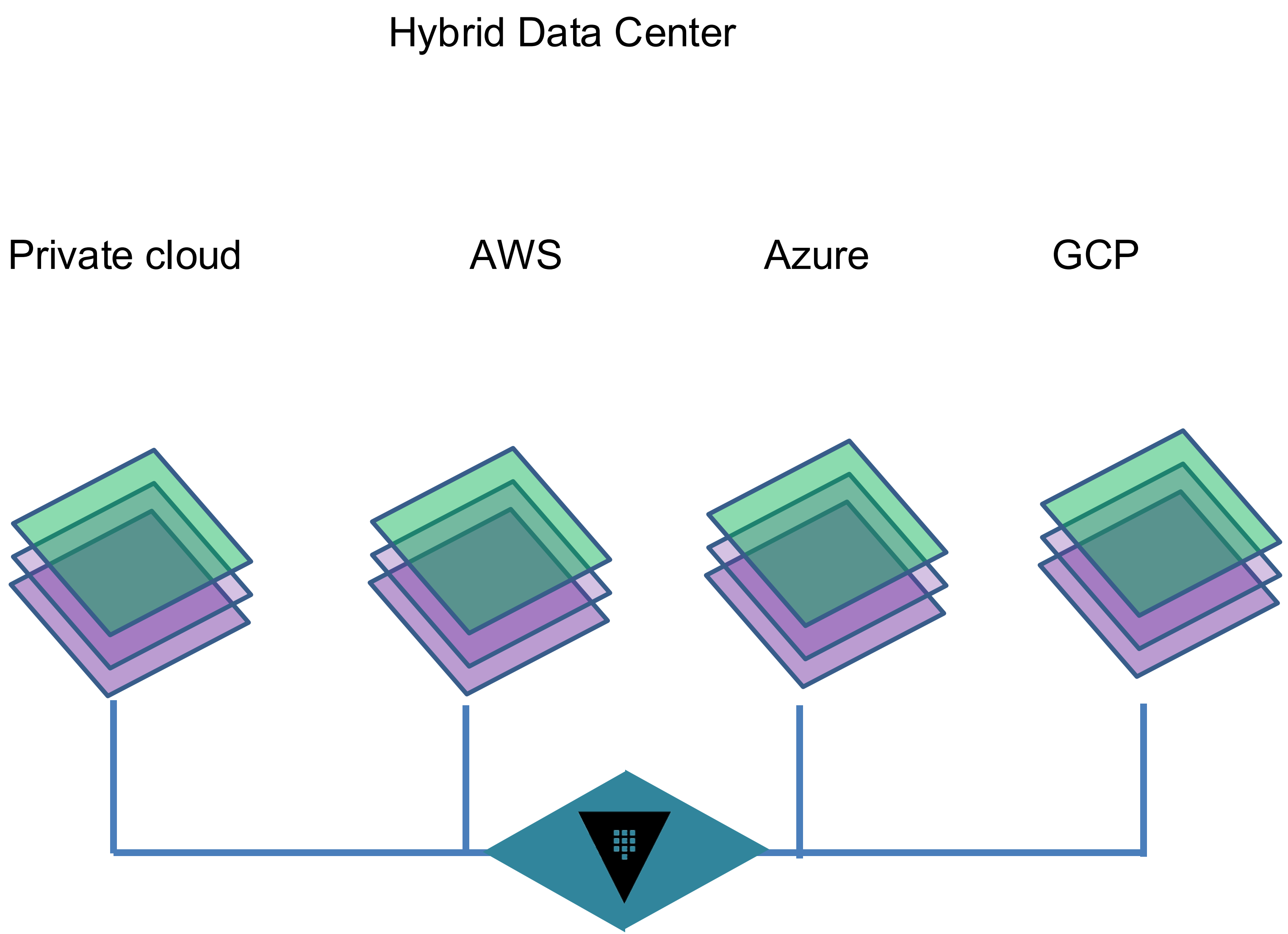
What are the Pros & Cons of the three techniques for managing secrets?

1. Environment Variables
2. Encrypt local files
3. Key Manager

Secure Infrastructure using Vault

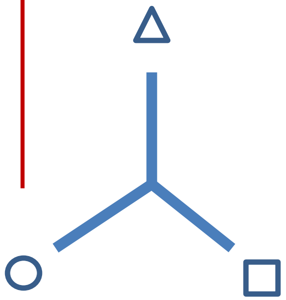


With each new cloud, network topologies become more complex.



Vault Objectives

- Provide single source of secrets for humans and machines .
- Scale to meet security needs of largest organizations.
- Allow for complete secret lifecycle management.



Eliminate Secret Sprawl

Securely Store any Secret

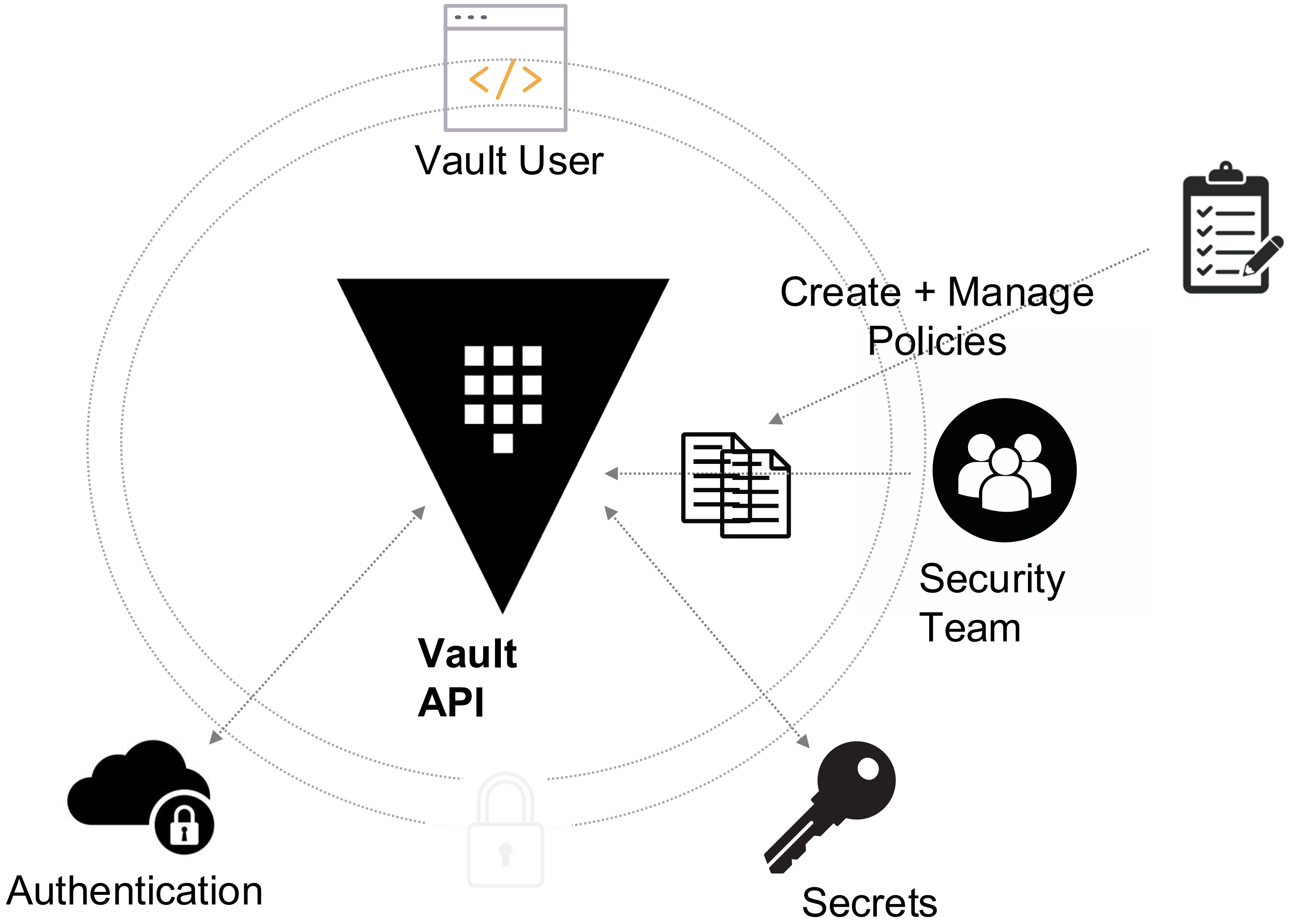
Secret Governance

Use Cases

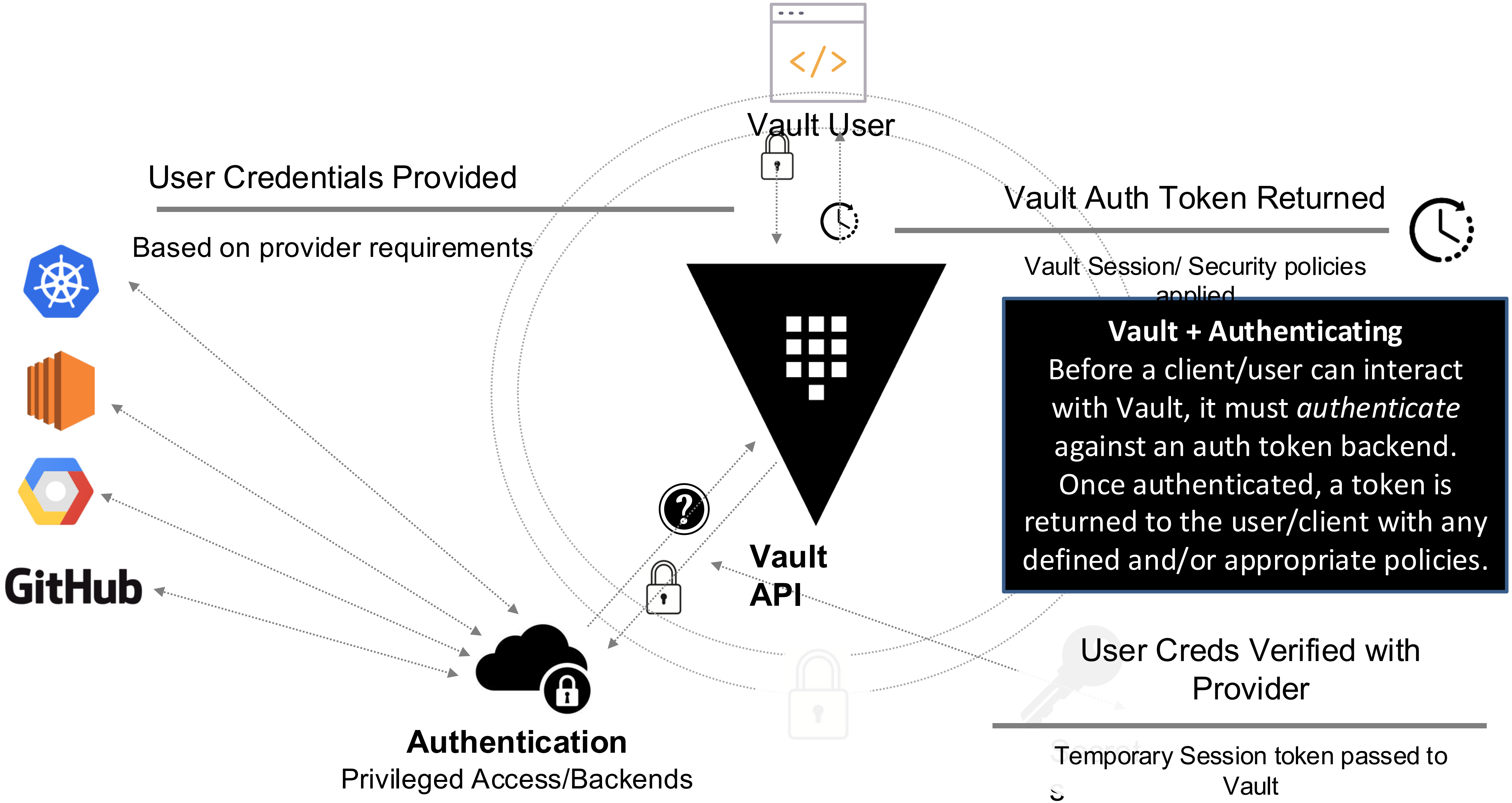
Secrets Management	Encryption as a Service	Identity Access Management
Secrets, identity, and access policy management workflow to secure any infrastructure and application resources.	One workflow to create and control the keys used to encrypt your data	Empower developers and operators to securely make application and infrastructure changes.

Vault - Security Policies

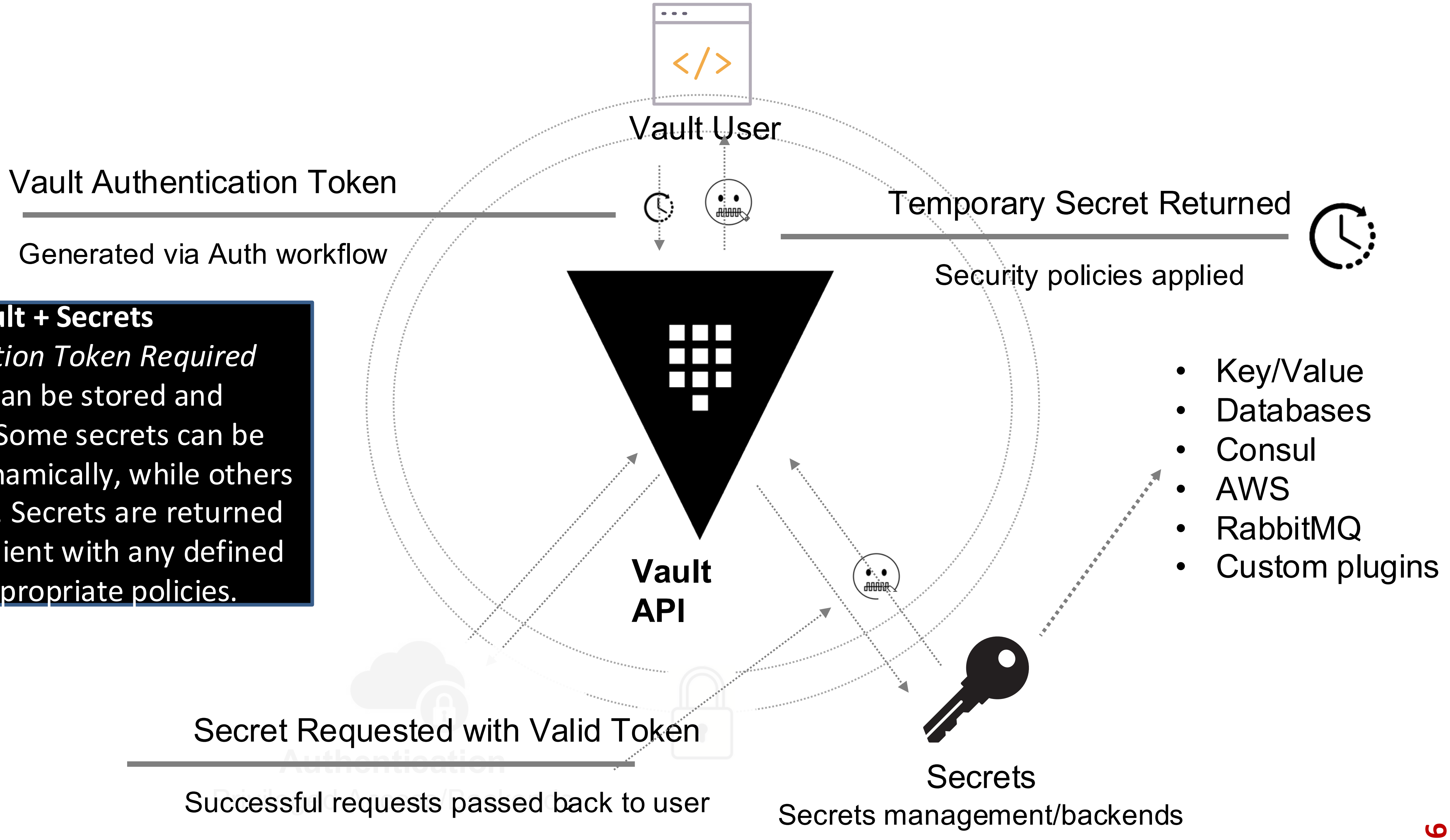
Policies with Vault
Vault uses policies to manage and safeguard access and secret distribution to applications and infrastructure. Policies provide a declarative way to **grant** and **deny** access to operations and paths.



Authentication Workflow



Secrets Workflow

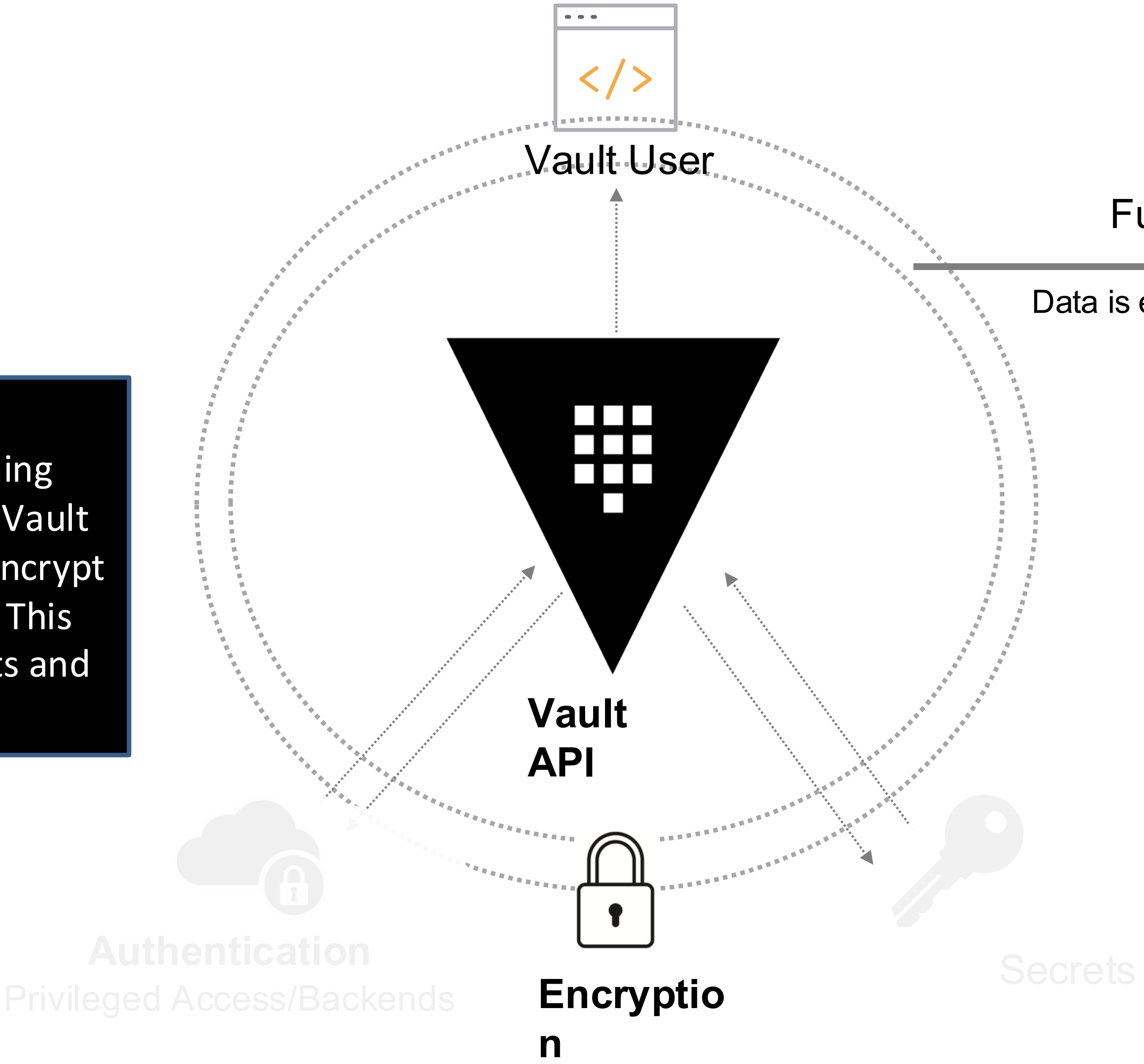


Vault + Secrets

Authentication Token Required

Secrets can be stored and generated. Some secrets can be generated dynamically, while others are verbatim. Secrets are returned to the user/client with any defined and/or appropriate policies.

Encryption(as a Service)



Full Data Encryption

Data is encrypted in-flight and at rest.

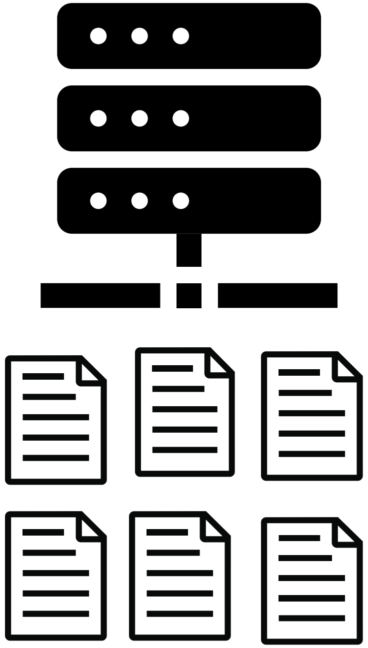
Encrypt everything
Vault believes that everything should always be encrypted. Vault uses ciphertext wrapping to encrypt all data at rest and in-flight. This minimizes exposure of secrets and sensitive information.

Directory structure

If you take a "monolithic" approach, and include all configuration in one file or directory you may run into these problems:

- Understanding and navigating the configuration becomes difficult.
- Making an update to one section may cause unintended consequences in other sections.
- There will be increasing duplication of similar blocks of configuration!
  - Dev, Staging, Production

Directory structure

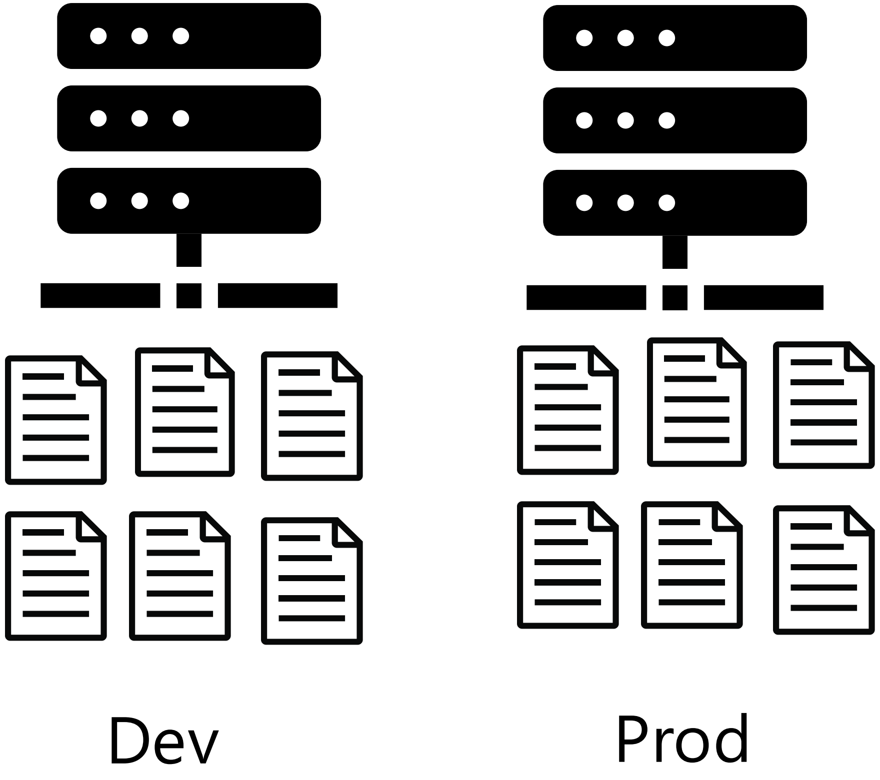


Monolith

Monolithic approach can have multiple environments with the same configuration and state files.

- main.tf
- variables.tf
- backend.tf
- provider.tf
- myvariables.tfvar
- outputs.tf

Directory structure



Separate your configuration and state using directories

- Dev
  - dev.tf
  - variables.tf
  - backend.tf
  - outputs.tf

- Prod
  - prod.tf
  - variables.tf
  - backend.tf
  - outputs.tf

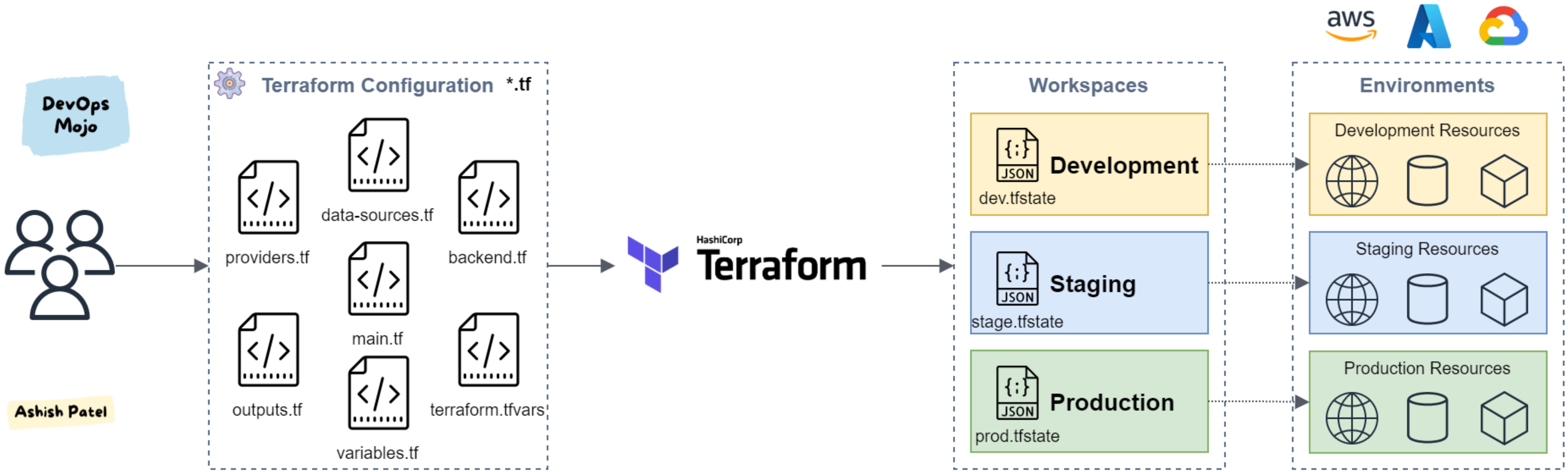
Workspaces

- Workspaces in Terraform are simply independently managed state files.
- A workspace contains everything that Terraform needs to manage a given collection of infrastructure, and separate Workspaces function like separate working directories.
- We can manage multiple environments with Workspaces.

Workspaces

- Workspaces allow you to separate your state and infrastructure without changing anything in your code.
- Use the same code base to deploy to multiple environments without overlap.
- Workspaces help to create multiple state files for the same terraform configuration files.

Workspaces



DevOps Mojo
Ashish Patel
Terraform Configuration *.tf
data-sources.tf
providers.tf
backend.tf
main.tf
outputs.tf
terraform.tfvars
variables.tf
HashiCorp
Terraform
Workspaces
Development
dev.tfstate
Staging
stage.tfstate
Production
prod.tfstate
aws
Environments
Development Resources
Staging Resources
Production Resources

Workspaces

Create the "dev" Workspace

```
$ terraform workspace new dev
```

Select the "dev" Workspace

```
$ terraform workspace select dev
```

Apply the configuration for "dev" environment

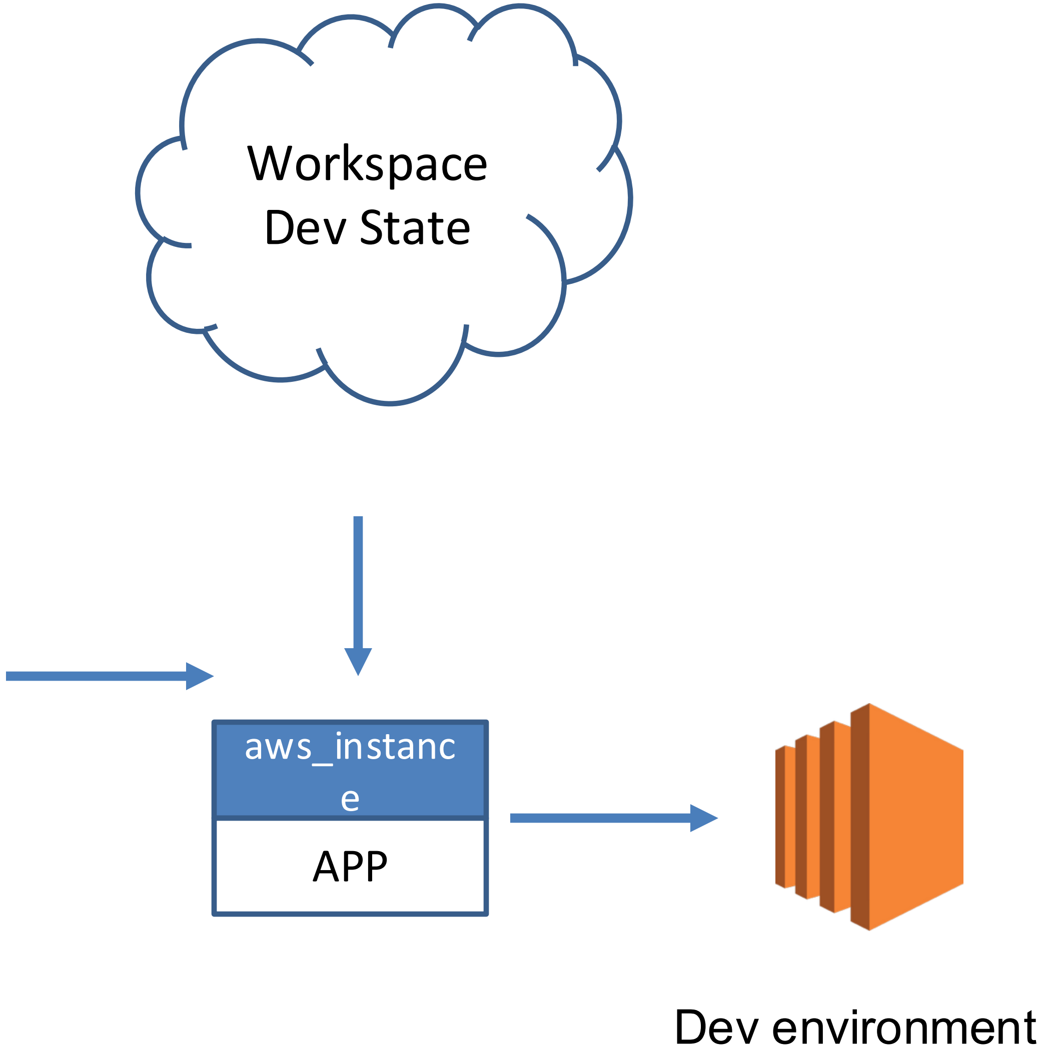
```
$ terraform apply
```

Workspaces

- Within your Terraform configuration, you may include the name of the current workspace using the `${terraform.workspace}` interpolation sequence.
- This can be used anywhere interpolations are allowed.

Workspaces

```
resource "aws_s3_bucket" "bucket" {
  bucket   = "${var.bucket}-${-terraform.workspace}"
  acl      = "private"
}
```



Workspaces

- If you don’t declare a workspace, Terraform uses the `default` workspace, which cannot be deleted.
- If you don’t specify a workspace explicitly, it means you are using the default workspace.

Workspaces

- Local state:
  - Terraform stores the workspace state in a directory called terraform.tfstate.d
  - TF creates a sub-directory for every workspace, which contain individual state files for the particular workspace.
  - All state files are stored in .terraform.tfstate.d/<workspacename>

Workspaces

- Remote state:
  - Workspaces are stored directly in the configured backend.
  - Usually, the state file has the workspace name appended to it

Lab: Write your own module

Lab: Refactor monolithic codebase