Day 5: Provisioners and Advanced Resource Management

Introduction to Provisioners:

- Provisioners in Terraform are used to execute scripts or other actions on local or remote machines as part of resource creation or destruction.
- They allow you to configure systems or make changes that are not directly supported by Terraform resource types.

Types of Provisioners:

1. local-exec: Executes a command on the machine running Terraform

2. remote-exec: Executes a script on a remote resource after it's created

3. file: Copies files or directories to a remote resource

Understanding when and how to use provisioners:

Provisioners should be used sparingly and as a last resort. They are often used for tasks such as:

1. Installing software not covered by existing providers

2. Executing custom scripts for configuration

3. Performing database migrations

4. Joining instances to a cluster

Example of a `local-exec` provisioner:

```
resource "aws_instance" "web" {

# ... other configuration ...

provisioner "local-exec" {

command = "echo ${self.private_ip} >> private_ips.txt"

}

}
```

Example of a `remote-exec` provisioner:

```
resource "aws_instance" "web" {

 # ... other configuration ...
 provisioner "remote-exec" {
 inline = [
 "sudo apt-get update",
 "sudo apt-get install -y nginx",
 ]
 }
}
```

Handling failure scenarios and retries:

Provisioners can be configured to handle failures in different ways:

1. **`on_failure** = continue`: Terraform will continue even if the provisioner fails

2. **`on_failure** = fail`: Terraform will treat the resource as tainted if the provisioner fails (default behavior)

For retries, you can use the `self.triggers` map to force a provisioner to run again:

```
resource "null_resource" "example" {
 triggers = {
 always_run = "${timestamp()}"
 }
 provisioner "local-exec" {
 command = "echo 'This will always run'"
 }
}
```

Resource Management:

Advanced techniques for resource addressing and data dependencies

1. Resource Addressing:

- Use `terraform state list` to see all resources

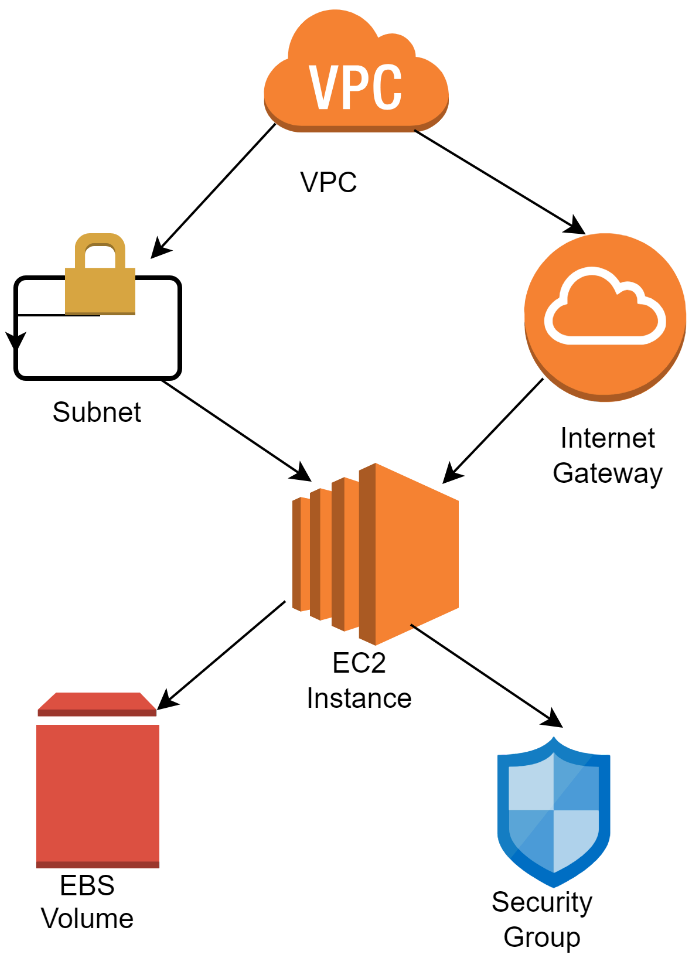
- Use `terraform state show [address]` to see details of a specific resource

- Use `terraform state mv` to move resources within a state or between states

2. Data Dependencies:

- Use `depends_on` for explicit dependencies

- Use `terraform graph` to visualize the dependency graph



This diagram illustrates the dependencies between different AWS resources in a typical Terraform configuration.

Industry Insight:

How companies manage complex provisioning workflows with Terraform:

1. Modular Architecture: Companies break down complex infrastructures into smaller, manageable modules. This allows for easier maintenance and reusability.

2. CI/CD Integration: Terraform is often integrated into CI/CD pipelines to automate infrastructure deployments alongside application releases.

3. State Management: For large-scale deployments, companies use remote state storage (e.g., S3 with DynamoDB locking) to enable collaboration and maintain state consistency.

4. Custom Providers: Some organizations develop custom Terraform providers to manage internal or legacy systems not covered by existing providers.

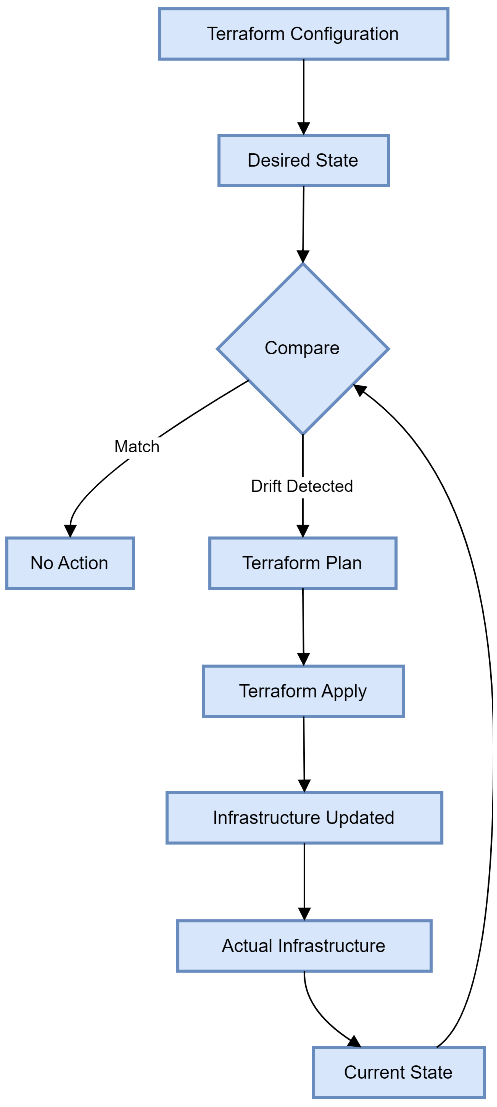
5. Terragrunt: Many companies use Terragrunt, a thin wrapper for Terraform, to keep their configurations DRY and manage multi-environment deployments.

Real-world examples of Terraform managing configuration drift:

1. Self-Healing Infrastructure: A financial services company uses Terraform to periodically reconcile their actual infrastructure state with the desired state defined in their Terraform configurations. This helps catch and correct any manual changes made outside of Terraform.

2. Compliance Enforcement: A healthcare organization uses Terraform to enforce compliance standards by regularly applying a baseline configuration to all resources. Any deviations are automatically corrected during the next Terraform run.

3. Multi-Cloud Consistency: A global e-commerce company uses Terraform to maintain consistency across multiple cloud providers. Terraform helps ensure that equivalent resources are provisioned and configured similarly across different clouds.



This diagram illustrates the process of detecting and correcting configuration drift using Terraform.

Hands-On Lab:

Implementing provisioners for post-deployment configurations

In this lab, we'll create an EC2 instance and use provisioners to install and configure a web server.

```
# Configure the AWS Provider

provider "aws" {

 region = "us-west-2"

}

# Create a VPC

resource "aws_vpc" "main" {

 cidr_block = "10.0.0.0/16"

 enable_dns_hostnames = true

 tags = {

 Name = "main-vpc"

 }

}

# Create an Internet Gateway

resource "aws_internet_gateway" "main" {

 vpc_id = aws_vpc.main.id

  tags = {

 Name = "main-igw"

 }

}

 # Create a Subnet

resource "aws_subnet" "main" {

 vpc_id = aws_vpc.main.id

 cidr_block = "10.0.1.0/24"

 availability_zone = "us-west-2a"
 map_public_ip_on_launch = true
  tags = {
 Name = "main-subnet"
 }
}
# Create a Route Table
resource "aws_route_table" "main" {
 vpc_id = aws_vpc.main.id
 route {
 cidr_block = "0.0.0.0/0"
 gateway_id = aws_internet_gateway.main.id
 }
tags = {
 Name = "main-route-table"
 }
}
 # Associate the Route Table with the Subnet
resource "aws_route_table_association" "main" {
 subnet_id = aws_subnet.main.id
 route_table_id = aws_route_table.main.id
}
 # Create a Security Group
resource "aws_security_group" "web" {
 name = "allow_web"
 description = "Allow inbound web traffic"
 vpc_id = aws_vpc.main.id
 ingress {
```

```
 description = "HTTP from anywhere"

 from_port = 80

 to_port = 80

 protocol = "tcp"

 cidr_blocks = ["0.0.0.0/0"]

 }

 ingress {

 description = "SSH from anywhere"

 from_port = 22

 to_port = 22

 protocol = "tcp"

 cidr_blocks = ["0.0.0.0/0"]

 }

egress {

 from_port = 0

 to_port = 0

 protocol = "-1"

 cidr_blocks = ["0.0.0.0/0"]

 }

  tags = {

 Name = "allow_web"

 }

}

 # Create an EC2 Instance

resource "aws_instance" "web" {

 ami = "ami-0c55b159cbfafe1f0" # Amazon Linux 2 AMI (HVM), SSD Volume Type

 instance_type = "t2.micro"
```

```
key_name = "your-key-pair-name" # Replace with your key pair name

vpc_security_group_ids = [aws_security_group.web.id]

subnet_id = aws_subnet.main.id

 tags = {

Name = "WebServer"

}

 # Use a provisioner to install and start Apache

provisioner "remote-exec" {

inline = [

"sudo yum update -y",

"sudo yum install -y httpd",

"sudo systemctl start httpd",

"sudo systemctl enable httpd",

"echo '<h1>Hello from Terraform provisioner!</h1>' | sudo tee /var/www/html/index.html"

]

}

# Use a provisioner to output the public IP

provisioner "local-exec" {

command = "echo ${self.public_ip} > instance_ip.txt"

}

 connection {

type = "ssh"

user = "ec2-user"

private_key = file("~/.ssh/your-key-pair.pem") # Replace with your key pair path

host = self.public_ip

}

}
```

```
# Output the public IP of the instance

output "instance_public_ip" {

 description = "Public IP address of the EC2 instance"

 value = aws_instance.web.public_ip

}
```

To use this configuration:

1. Save the code in a file named `main.tf`.

2. Replace `"your-key-pair-name"` with the name of your AWS key pair.

3. Replace `"~/.ssh/your-key-pair.pem"` with the path to your private key file.

4. Initialize Terraform:

- **terraform init**

5. Review the plan:

- **terraform plan**

6. Apply the configuration:

- **terraform apply**

This will create a VPC, subnet, internet gateway, route table, security group, and an EC2 instance. The provisioners will install and configure Apache web server on the instance.

Questions on provisioners and resource management

1. Which of the following is NOT a type of Terraform provisioner?

a) local-exec

b) remote-exec

c) file

d) cloud-exec

2. What is the default behavior of Terraform when a provisioner fails?

a) Continue with the rest of the configuration

b) Retry the provisioner indefinitely

c) Mark the resource as tainted

d) Destroy the resource immediately

3. How can you force a null_resource to always run its provisioners?

a) Set `always_run = true` in the resource block

b) Use the `timestamp()` function in the triggers map

c) Add `force_run = true` to the provisioner block

d) Use the `ignore_changes` lifecycle meta-argument

4. What command would you use to see the current state of all your Terraform-managed resources?

a) terraform show

b) terraform list

c) terraform state list

d) terraform resources

5. Which Terraform command can help you visualize resource dependencies?

a) terraform visualize

b) terraform graph

c) terraform show-dependencies

d) terraform map

Answers:

1. d

2. c

3. b

4. c

5. b

- This concludes the content for Day 5 on Provisioners and Advanced Resource Management.
- The structure follows the provided reference, including sections on industry insights, real-world scenarios, a hands-on lab, and exam practice questions.
- The content is enhanced with Diagrams as Code to visualize key concepts.