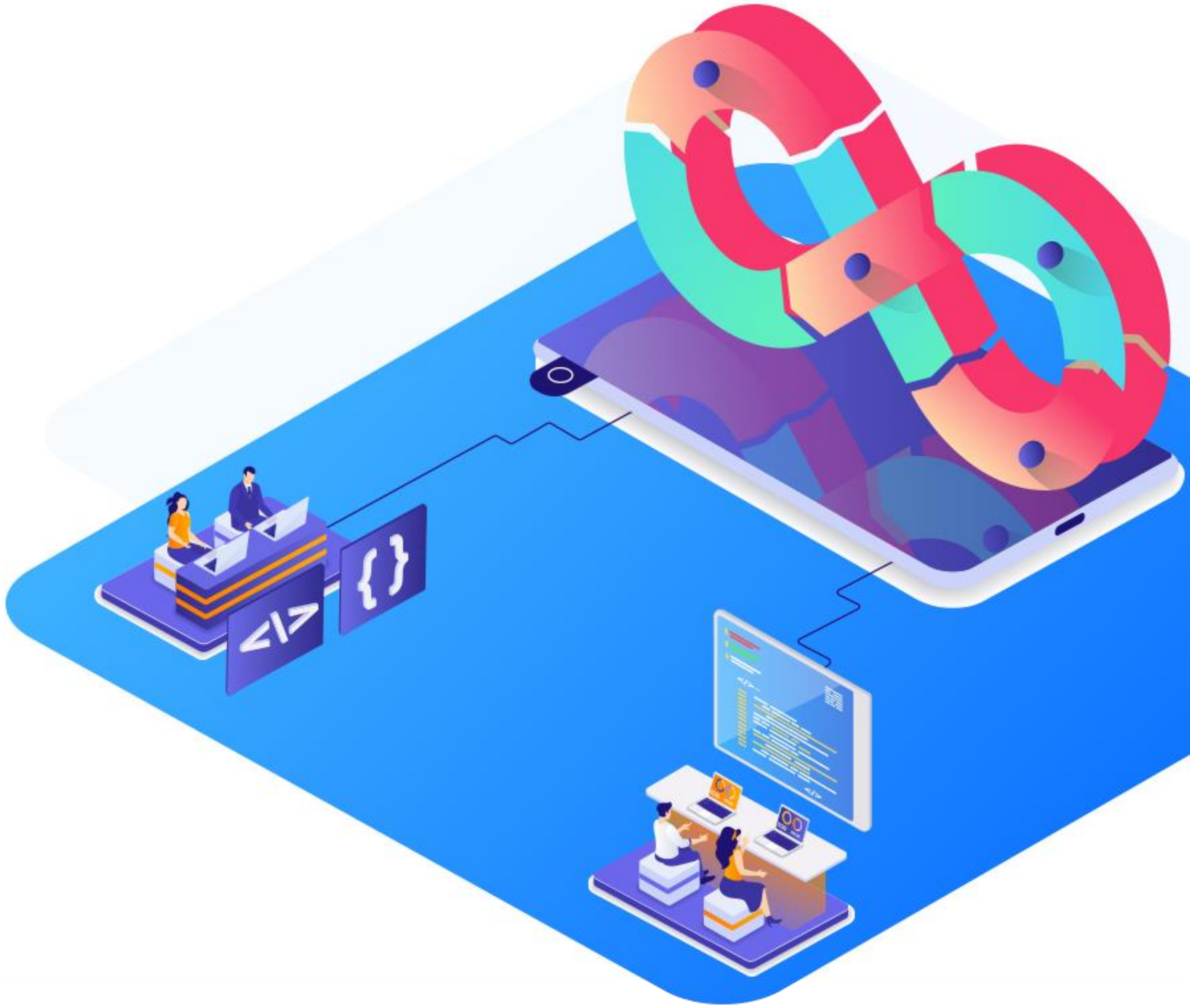
Configuration Management with Ansible and Terraform



Read, Generate, and Modify Configurations

Learning Objectives

By the end of this lesson, you will be able to:

- Identify the steps in the basic workflow of managing configurations in Terraform for efficient infrastructure management
- Determine the use of local variables, input variables, and outputs in Terraform for simplifying, reusing, customizing, and sharing configuration management
- Create a Terraform file to define an AWS EC2 instance for deploying a virtual machine
- Assess the importance of variable validation, suppression techniques, and managing collections and structure types for securing sensitive data and organizing complex datasets in Terraform code

Learning Objectives

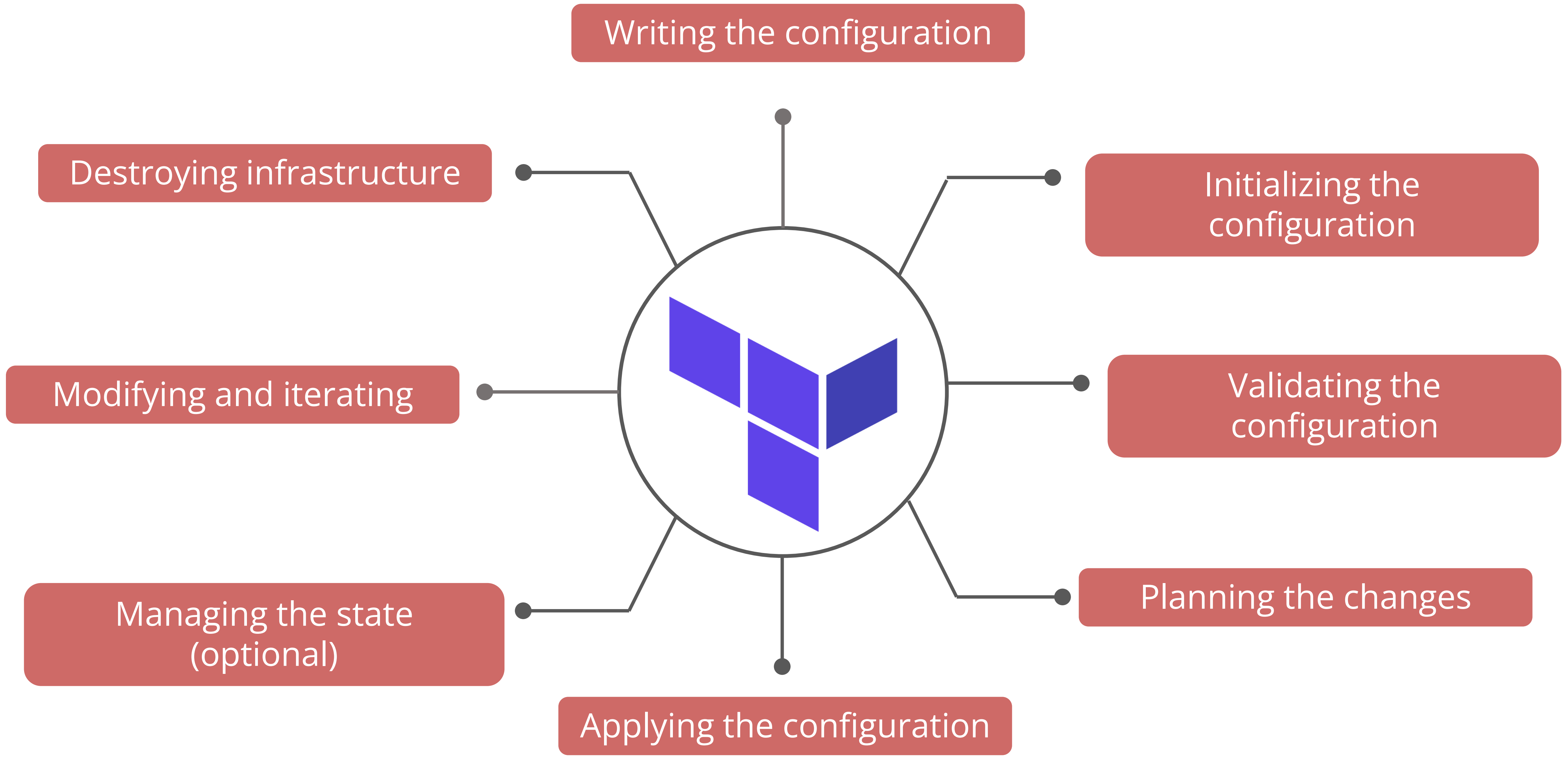
By the end of this lesson, you will be able to:

- Utilize data blocks and built-in functions in Terraform for effectively retrieving and manipulating external data sources
- Develop dynamic blocks within Terraform modules for managing complex and nested infrastructure configurations
- Define Terraform graphs to understand resource dependencies and the overall structure of Terraform configurations for better planning and troubleshooting
- Implement the Terraform resource lifecycle, including creation, update, and deletion of resources, for maintaining infrastructure state effectively

Workflow and Variables

Basic Workflow of Managing Configurations

It involves several key steps designed to efficiently provision and manage infrastructure as code, such as:


Writing the configuration
Initializing the configuration
Validating the configuration
Planning the changes
Applying the configuration
Managing the state (optional)
Modifying and iterating
Destroying infrastructure

Writing the Configuration

Terraform configurations are written in HCL that describe the desired state of the infrastructure. This step defines infrastructure components and their hosting providers, such as AWS or Google Cloud.

Example: Define a simple AWS EC2 instance

```
provider "aws" {
  # Replace with your actual AWS credentials
  access_key = “YOUR_ACCESS_KEY"
  secret_key = “YOUR_SECRET_KEY"
  region     = "us-east-1" # Replace with your desired region
}
```

Writing the Configuration

Example: Define a simple AWS EC2 instance

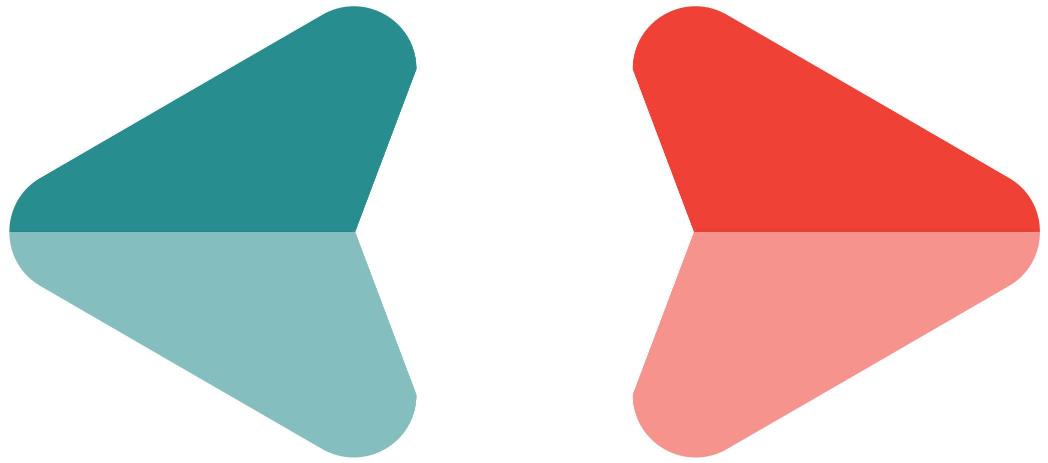
```
data "aws_ami" "ubuntu" {
  most_recent = true
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }
  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
  owners = ["099720109477"] # Canonical
}

resource "aws_instance" "example" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

    tags = {
    Name = "Mymachine"
  }
}
```

Initializing the Configuration

The **terraform init** command is used to initialize Terraform. Initialization prepares the working directory for Terraform operations and is particularly focused on two main actions:

Download providers

Terraform downloads the necessary provider plugins that are required to manage the resources.



Backend initialization

Terraform initializes the backend, which will be used for state management.

Validating the Configuration

The **terraform validate** command is used to check the syntax and configuration of the Terraform files without altering any infrastructure.

The typical outcomes of running terraform validation:

If the Terraform configuration is valid and error-free, the command will exit with a success message, indicating that the configuration is syntactically correct.

If there are any syntax errors or issues in the Terraform files, the command will display specific error messages pointing to the problematic lines or elements in the configuration.

Planning the Changes

The **terraform plan** command is used to create an execution plan.

This command helps to:

01 Analyze and adjust configurations to achieve the desired state

02 Provide an overview of resources to be created, modified, or deleted

03 Detail actions for review, preventing unintended changes

Applying the Configuration

The **terraform apply** command is used to execute the plan and apply the changes to the infrastructure.

- Terraform requires confirmation before it starts.
- Once confirmed, it provisions the specified resources and updates the state file.

Managing the State (Optional)

The terraform state file, by default, is named **terraform.tfstate** and is held in the same directory where Terraform is run. It is created after running **terraform apply**.

Syntax

```
terraform state <subcommand> [options] [args]
```

- The **terraform state** command is used for state management.
- With increased Terraform usage, this command allows for necessary state adjustments without direct modifications.

Modifying and Iterating

As requirements change, users can modify the configuration files, repeat the plan, and apply cycle.

Terraform is designed to update the infrastructure incrementally.

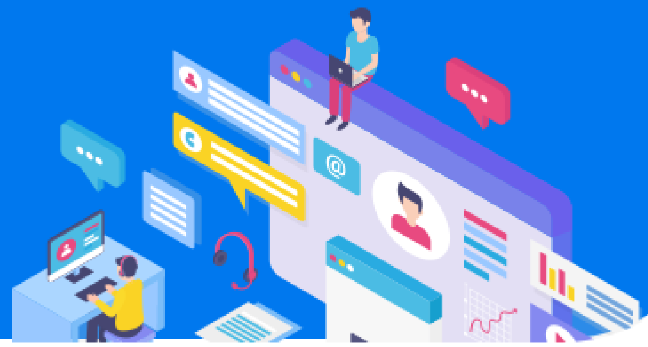
Terraform configurations can be refactored to increase reuse and maintainability without affecting the existing infrastructure.

Destroying Infrastructure

The **terraform destroy** command is used to remove all resources defined in the Terraform configuration when they are no longer needed.

Terraform handles the dependencies to ensure that resources are destroyed in a safe order.

Assisted Practice



Implementing Workflow of Managing Configurations in Terraform **Duration: 10 Min.**

Problem statement:

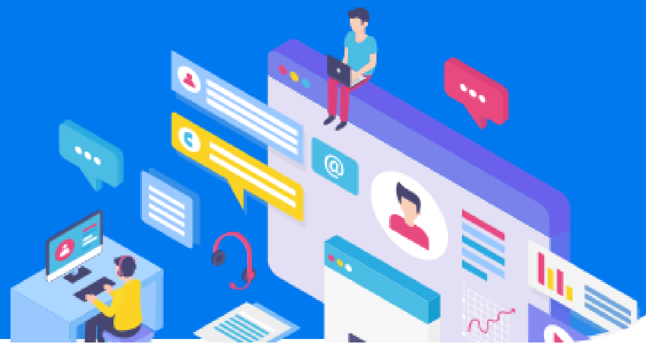
You have been assigned a task to implement a workflow for managing AWS infrastructure configurations in Terraform efficiently and effectively

Outcome:

By completing this demo, you will gain proficiency in managing AWS infrastructure configurations using Terraform efficiently and effectively.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



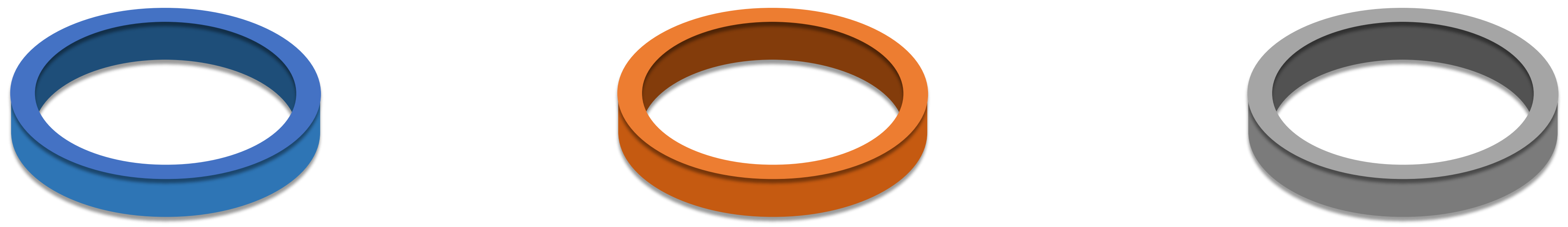
Steps to be followed:

1. Create an IAM user
2. Create access and secret key
3. Implement a workflow for managing configurations in Terraform

ASSISTED PRACTICE

Variables and Outputs

The Terraform language incorporates various types of blocks used to request or publish named values, which are as follows:



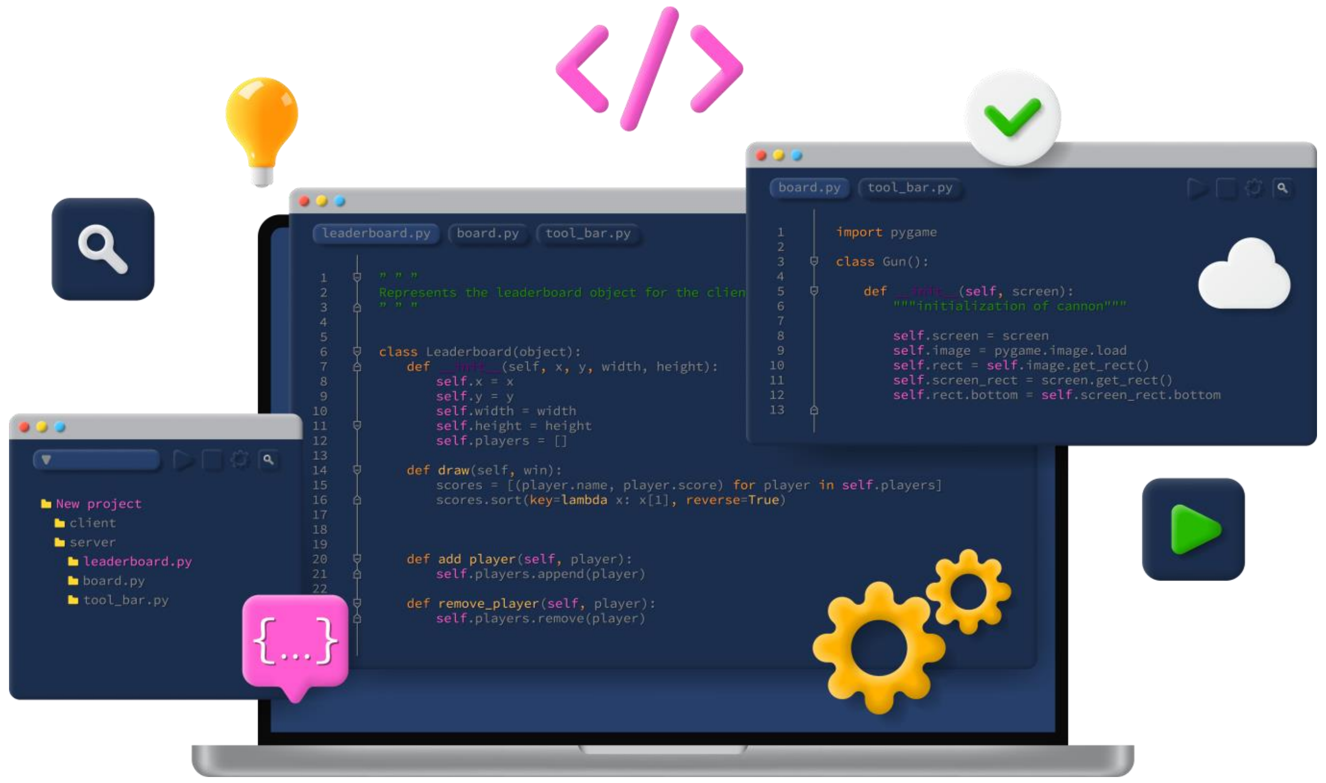
Input variables

Output values

Local values

Input Variables

They allow external customization of Terraform configurations without altering the code. They are defined in a module and act as parameters that can be passed to the module.



Input variables are like function arguments.

Declaring Input Variables

Here is the syntax to declare the input variables:

Example

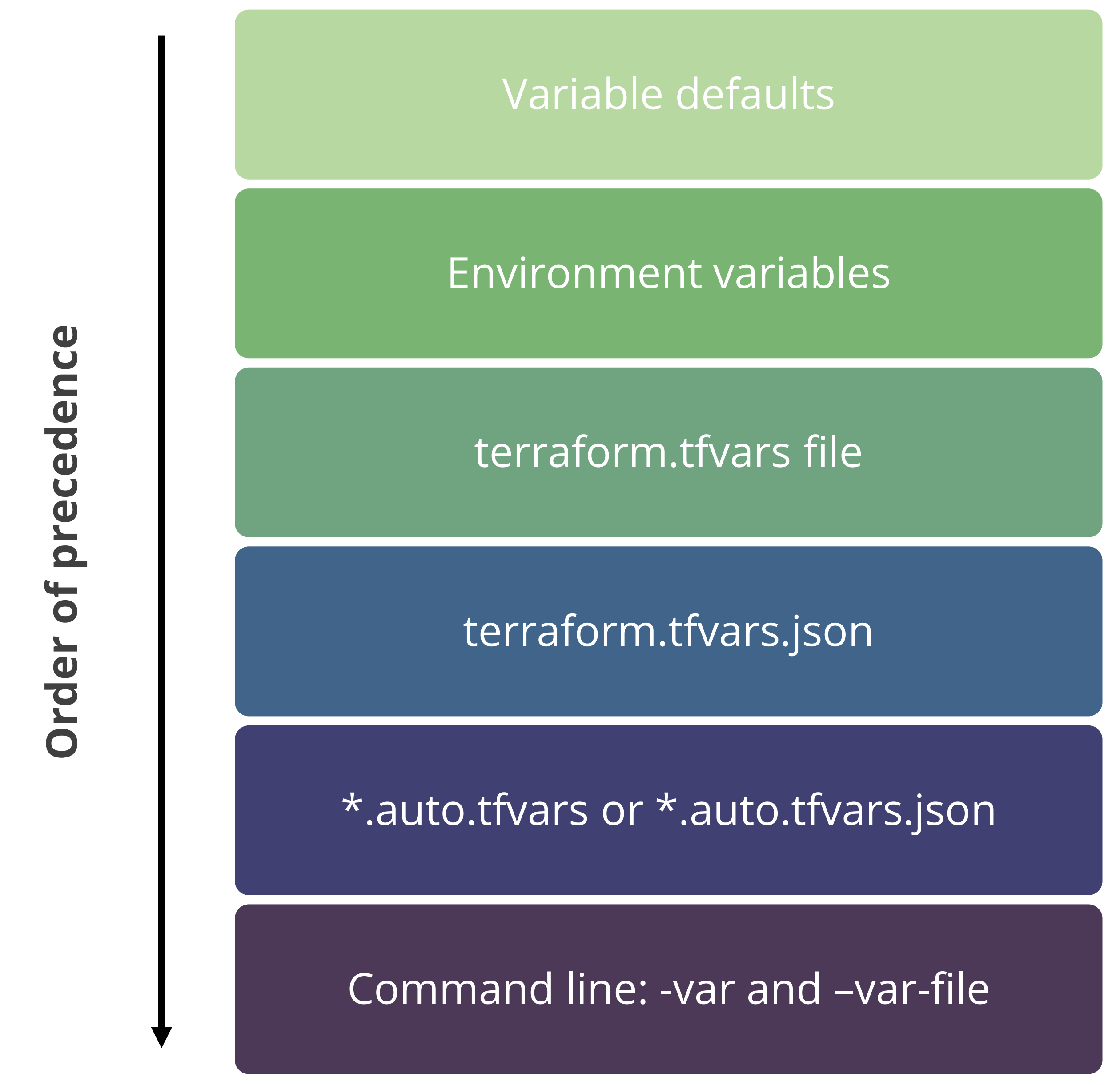
```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Input Variables: Arguments

Terraform CLI defines the following optional arguments for variable declarations:

default	A default value makes the variable optional.
type	This argument specifies the value types accepted for the variable.
description	This specifies the input variable documentation.
validation	It is a block used to define validation rules, usually in addition to type constraints.
sensitive	It limits Terraform UI output when the variable is used in the configuration.

Variable Definition Precedence



Terraform input variables

Output Values

They are used to organize data to be easily accessible to Terraform users and other Terraform configurations and can be used to extract information from a module or resources.

Uses

- A parent module can access a subset of a child module's resource attributes using outputs.
- After running ***terraform apply***, a root module can use outputs to print specific values in the CLI output.
- Root module outputs can be accessed by other configurations using a terraform_remote_state data source when using a remote state.

Note

When the meaning is clear from context, output values are frequently referred to as outputs.

Declaring Output Values

Example

```
output "instance_ip_addr" {
  value = aws_instance.server.private_ip
}
```

- Each output value exported by a module must be declared using an output block.
- The label immediately after the output keyword is the name, which must be a valid identifier.
- The value argument takes an expression whose result will be returned to the user.

When Terraform implements a plan, the outputs are rendered; they will not be rendered if a user runs a ***Terraform plan.***

Accessing Child Module Outputs

In a parent module, outputs of child modules are available in expressions as:
module.<MODULE NAME>.<OUTPUT NAME>

Example

If a child module named ***web_server*** declared an output named ***instance_ip_addr***, a user could access that value as ***module.web_server.instance_ip_addr***.

Output blocks can optionally include **description**, **sensitive**, and **depends_on** arguments.
(Discussed in the following sections)

Local Values

A local value associates a name with an expression, allowing users to reuse the name multiple times within a module instead of duplicating the expression.

Local values are like a function's temporary local variables.

Declaring a Local Value

Here is the syntax to declare the local variables:

Example

```
locals {
  service_name = "forum"
  owner        = "Community Team"
}
```

Local values in a module can reference and manipulate other values, including variables, resource attributes, and other local values, as shown below:

Example

```
locals {
  # Ids for multiple sets of EC2 instances, merged together
  instance_ids = concat(aws_instance.blue.*.id, aws_instance.green.*.id)
}

locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Service = local.service_name
    Owner   = local.owner
  }
}
```

Using Local Values

Once a local value is declared, a user can reference it in expressions as **local.<NAME>**.

Example

```
resource "aws_instance" "example" {
  # ...

  tags = local.common_tags
}
```

- A local value can only be accessed in expressions within the module where it was declared.

Note

Local values are defined within a **locals** block (plural) but are accessed as attributes of an object named **local** (singular).

When to Use Local Values?

Local values are particularly useful in Terraform configurations for several scenarios:

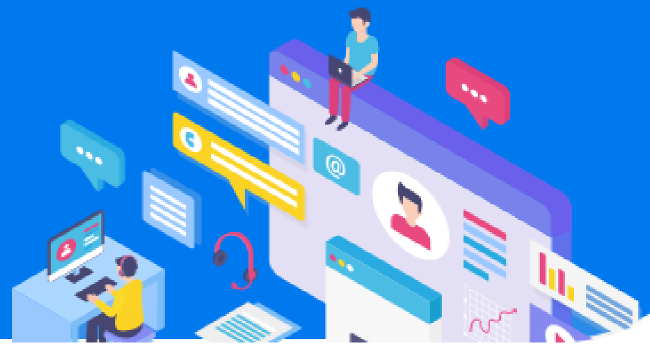
Local values are used for:

01 **Reducing duplication:** Use local values to avoid repeating the same expression or value across multiple parts of the configuration.

02 **Simplifying complex expressions:** When complex calculations or expressions are needed multiple times, local values can simplify the configuration by storing these in one place.

03 **Facilitating future changes:** If a value might require frequent updates or modifications, using a local value simplifies and speeds up these changes.

Assisted Practice



Working with Variables in Terraform **Duration: 10 Min.**

Problem statement:

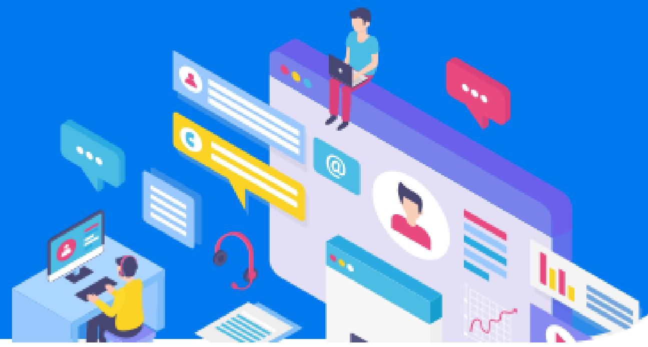
You have been assigned a task to utilize Terraform variables and local values for efficient and flexible infrastructure configurations

Outcome:

By completing this demo, you will gain proficiency in utilizing Terraform variables and local values for efficient and flexible infrastructure configurations.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Define local values and variables
2. Apply configuration using defined variables
3. Verify and utilize output values

Quick Check

You have written a Terraform configuration file to provision an AWS EC2 instance. Which of the following actions should you take to ensure the instance is provisioned with the correct variables and outputs?

A. Initialize the configuration, plan the changes, apply the configuration, and define local variables

B. Define input variables, initialize the configuration, validate the configuration, and apply the configuration

C. Write the configuration, initialize the configuration, plan the changes, and define Terraform outputs

D. Define input variables, write the configuration, plan the changes, and apply the configuration

Advanced Variable Management in Terraform

Custom Validation Rules

Users can specify custom validation rules for a particular variable by adding a **validation** block within the corresponding **variable** block.

This feature was introduced in Terraform CLI v0.13.0.

Custom Validation Rules: Example

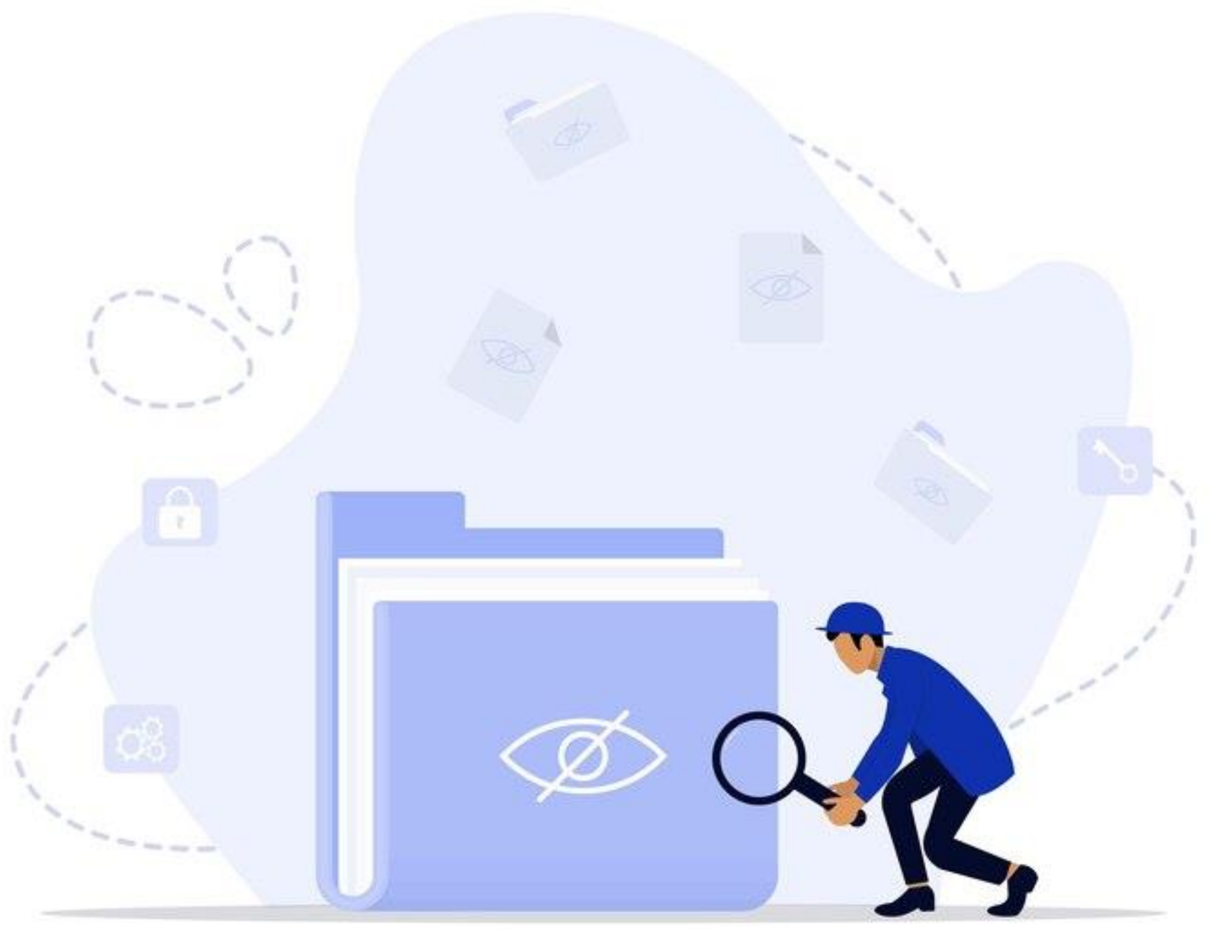
The example below checks whether the AMI ID has the correct syntax:

Example

```
variable "image_id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    condition     = length(var.image_id) > 4 && substr(var.image_id, 0, 4) == "ami-"
    error_message = "The image_id value must be a valid AMI id, starting with \"ami-\"."
  }
}
```

Suppressing Values in CLI Output

A variable marked as **sensitive** in Terraform will not show its value in the **plan** or **apply** output when used in the configuration.



Suppressing Values in CLI Output: Example

Expressions dependent on a sensitive variable are also treated as sensitive, and so in the below example, the two arguments of **resource "some_resource" "a"** will also be hidden in the plan output:

Example

```
variable "user_information" {
  type = object({
    name    = string
    address = string
  })
  sensitive = true
}

resource "some_resource" "a" {
  name    = var.user_information.name
  address = var.user_information.address
}
```

Declare a variable as **sensitive** by setting the sensitive argument to **true** to prevent its values from appearing in CLI outputs.

Securing Secrets in Terraform Code

It is crucial to maintain the integrity and security of the infrastructure.

Effective management of these secrets minimizes the risk of data breaches and unauthorized access.

Secure Secrets in Terraform Code

Here are the key practices and tools to ensure that sensitive data like passwords, API keys, and other credentials are protected when using Terraform:

- Use sensitive data flag
- Use environment variables
- Use Terraform vault provider

Secure Secrets in Terraform Code

Here are the key practices and tools to ensure that sensitive data like passwords, API keys, and other credentials are protected when using Terraform:

- Encrypt state files
- Use least privilege access
- Audit and rotate secrets regularly

Using Sensitive Data Flag

Mark variables as sensitive in Terraform to prevent their values from being displayed in the CLI output

Example

```
variable "api_key" {
  type      = string
  sensitive = true
}
```

This helps in minimizing exposure to sensitive data.

Using Environment Variables

Pass secrets at runtime through environment variables instead of hardcoding them in the Terraform configurations; this practice keeps sensitive data out of the version control systems.

Example

```
export TF_VAR_secret_key="your_secret_key"
```

In Terraform, access it using:

```
variable "secret_key" {}
```

Using Terraform Vault Provider

Integrate HashiCorp vault with Terraform to manage secrets and sensitive data

Example: Configuration to retrieve a secret from vault

```
provider "vault" {}

data "vault_generic_secret" "db_password" {
  path = "secret/data/db_password"
}

output "db_password" {
  value     =
data.vault_generic_secret.db_password.data["password"]
  sensitive = true
}
```

Vault securely stores and tightly controls access to tokens, passwords, certificates, API keys, and other secrets.

Encrypting State Files

Store Terraform state files on secure remote backends like AWS S3, which support encryption, to protect sensitive infrastructure data

Example: S3 backend configuration with encryption

```
terraform {
  backend "s3" {
    bucket         = "your-terraform-state-bucket"
    key            = "path/to/your/state/file"
    region         = "us-east-1"
    encrypt        = true
    dynamodb_table = "your-lock-table"
  }
}
```

Other Practices for Terraform Data Protection

Use least privilege access → Set up the cloud provider and Terraform access based on the principle of least privilege

Audit and rotate secrets regularly → Audit the Terraform project’s secrets and credentials regularly and rotate them periodically

Variable Collection and Structure Types

In Terraform, effective data management is crucial for creating reusable, maintainable, and scalable configurations supported by various variable types that structure complex data.

Here is an overview of the variable collection and structure types used in Terraform:

Primitive types

Complex types

Primitive Types

These are the basic types that can hold single values:

String	Represents a sequence of Unicode characters, such as "hello"
Number	Represents numeric values, including both whole numbers like 15 and fractional values like 6.283185
Bool	Represents a boolean value, either true or false, useful in conditional logic

Complex Types

These types are used to organize and structure more complex data:

List (or Tuple)

Represents a sequence of values, like ["us-west-1a", "us-west-1c"], indexed from zero

Example

```
variable "availability_zones" {
  type    = list(string)
  default = ["us-west-1a", "us-west-1b"]
}

output "first_zone" {
  value = var.availability_zones[0]
}
```

Set

Represents an unordered collection of unique values, without any specific order or element identifiers

Example

```
variable "subnet_ids" {
  type    = set(string)
  default = ["subnet-abc123", "subnet-def456"]
}

output "subnets" {
  value = var.subnet_ids
}
```

Complex Types

Map (or Object)

Represents a collection of key-value pairs identified by names, such as {name = "Mabel", age = 52}

Example

```
variable "user_details" {
  type = map(string)
  default = {
    name = "Mabel"
    age  = "52"
  }
}

output "user_name" {
  value = var.user_details["name"]
}
```

Indices and Attributes

In Terraform, elements of lists, tuples, and maps can be accessed using different notations, such as:

Square-bracket index notation → For lists and tuples, indices (whole numbers) are used to access elements, such as **local.list[3]**. For maps and objects, keys (strings) are used, like **local.map["keyname"]**.

Dot-separated attribute notation → Objects with attribute names that are valid identifiers can be accessed using dot notation, such as **local.object.attrname**.

Use square-bracket notation to prevent syntax errors for maps with arbitrary keys, especially user-specified or dynamically generated ones

Type Conversion

Terraform is designed to facilitate flexible type conversions to ease the integration and manipulation of different data types that include:

Automatic conversion

Terraform automatically converts values between types (numbers, bools, strings) to match the expected types of resource arguments or module variables.

Examples

- Boolean values **true** and **false** convert to **"true"** and **"false"** when strings are needed.
- Numeric values like **15** become **"15"** as strings, and numeric strings convert to numbers.

Type Conversion

Terraform is designed to facilitate flexible type conversions to ease the integration and manipulation of different data types that include:

Type mismatch handling

- If automatic conversion is not possible, Terraform will raise a type mismatch error.
- This requires the expression in the configuration to be adjusted to explicitly match the required type, potentially using built-in functions such as **tostring()**, **tonumber()**, or **tobool()** to convert values manually.

Assisted Practice

Validating Variables and Securing Secrets in Terraform Code **Duration: 10 Min.**

Problem statement:

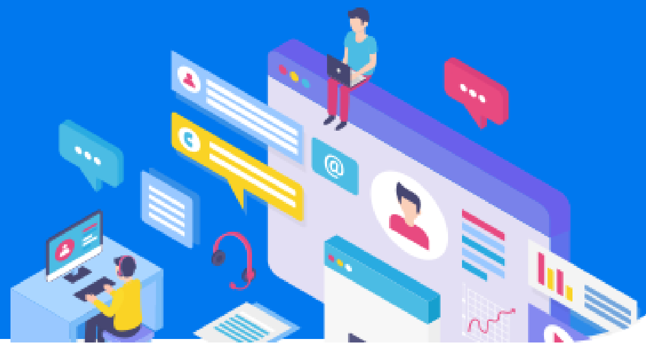
You have been assigned a task to validate variables and secure sensitive data within the Terraform code for enhanced configuration security and reliability

Outcome:

By completing this demo, you will gain proficiency in validating variables and securing sensitive data within Terraform code for enhanced configuration security and reliability.

Note: Refer to the demo document for detailed steps

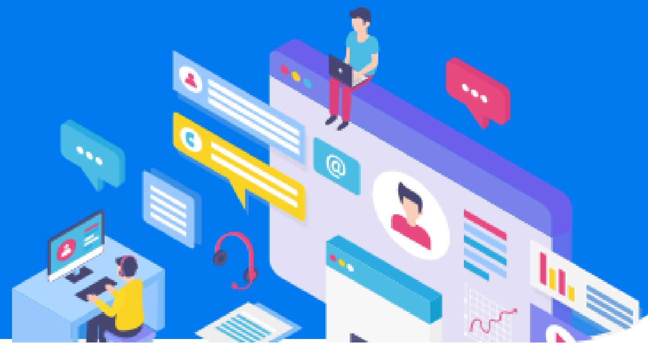
Assisted Practice: Guidelines



Steps to be followed:

1. Set up and validate variables
2. Secure and manage sensitive data
3. Apply configuration and review outputs

Assisted Practice



Working with Collections and Structure Types **Duration: 10 Min.**

Problem statement:

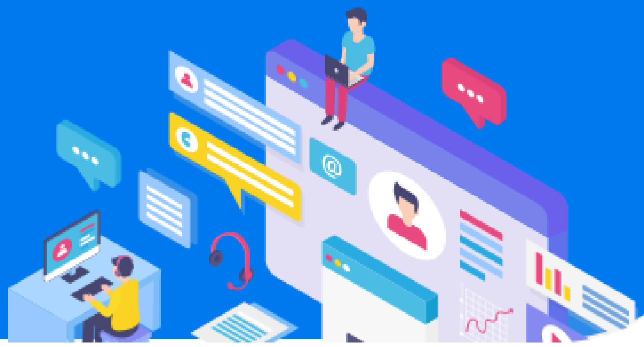
You have been assigned a task to utilize Terraform collections and structure types for enhanced configuration flexibility and readability

Outcome:

By completing this demo, you will gain proficiency in utilizing Terraform collections and structure types for enhanced configuration flexibility and readability.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Create and reference a new list variable
2. Add a new map variable to replace static values
3. Iterate over a map to create multiple resources
4. Utilize a complex map variable to simplify configuration readability

Quick Check

You need to securely manage sensitive information, such as API keys, in your Terraform configuration, while ensuring the variables are correctly validated. Which steps should you take?

A. Define sensitive variables, use environment variables for secrets, and validate using custom rules

B. Store secrets in plain text, define variables, and use built-in functions for validation

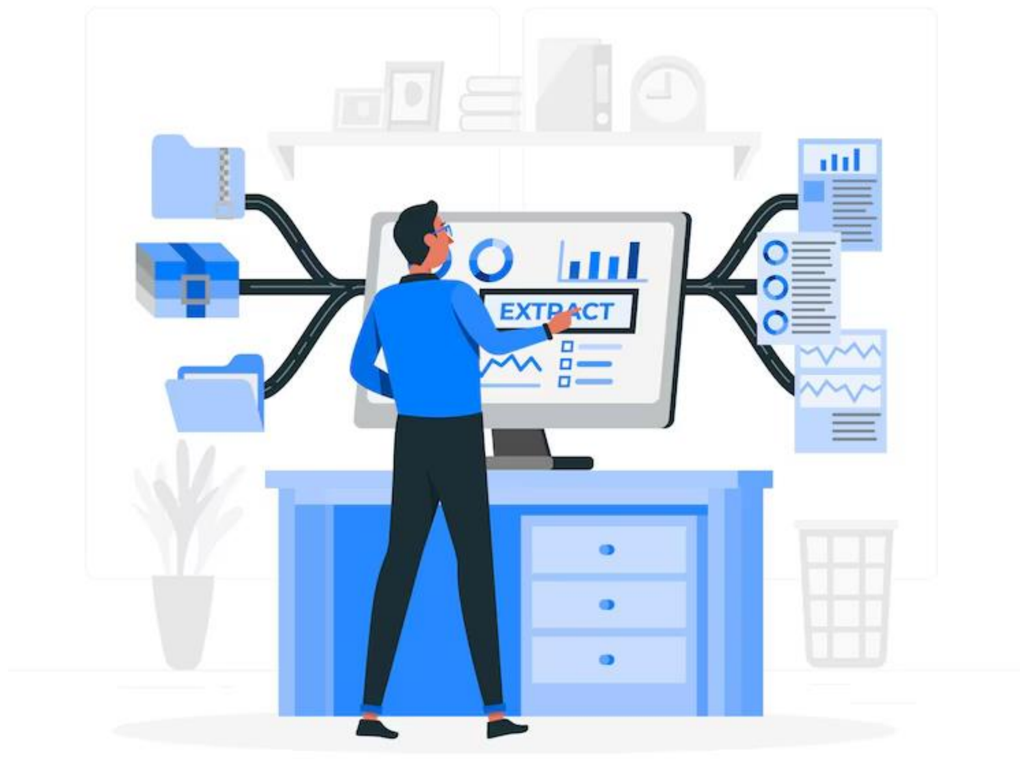
C. Use dynamic blocks for sensitive data, store secrets in the state file, and ignore validation

D. Hardcode API keys in the configuration, use local variables for secrets, and validate with default rules

Block and Functions

What Are Data Sources?

Data sources in Terraform are special configurations that allow users to pull data from various external sources.



These can be other Terraform states, cloud providers, or any service that exposes an API.

Data Block

It defines a data source within a Terraform module.

Example

```
data "aws_ami" "example" {
  most_recent = true
  owners = ["self"]
  tags = {
    Name   = "app-server"
    Tested = "true"
  }
}
```

This block requests data from an external source, specifying an Amazon Machine Image (AMI) based on certain filters.

Data Block: Characteristics

1. Data sources are declared using a **data** block that specifies the type of data and a local name.

2. The block includes query constraints to refine what data is fetched, such as **most_recent**, **owners**, and **tags**.

3. Data fetched from the source is exposed under the given local name and can be referenced within the Terraform module.

Data Resources vs. Managed Resources

Terraform distinguishes between managed resources and data resources in the following ways:

Data resources	Managed resources
They are only read by Terraform; they do not manage any objects in the infrastructure but provide necessary information that can be utilized in configuring other parts of the Terraform module.	They are referred to as **resources** and are handled by Terraform to create, update, and delete infrastructure objects.

Both resource types accept arguments and export attributes, enabling dynamic configurations that adapt to changes in data or external system states.

Data Source Arguments

Every data resource in Terraform is linked to a specific data source, which determines the types of objects it can access and the available query constraint arguments.

Each data source belongs to a provider, a Terraform plugin that offers resources and data sources for specific cloud or on-premises platforms.

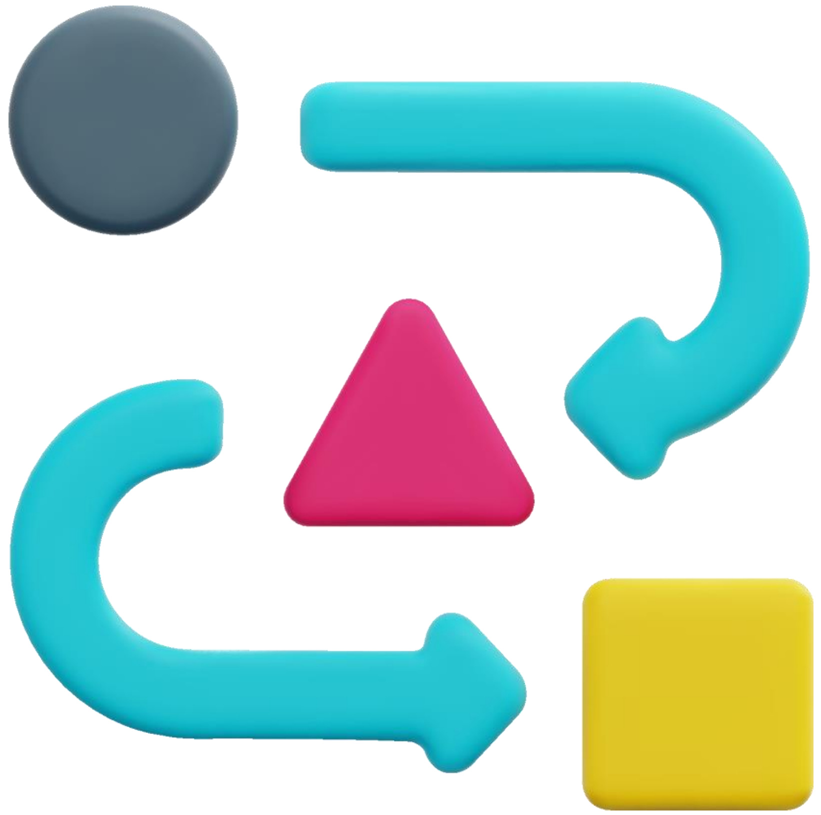
Data Source Arguments

The configurations within a data block are specific to each data source, allowing for the use of dynamic expressions and other Terraform language features to fully define these settings.

Terraform uses **meta-arguments** common to all data sources, with limitations on language features detailed in subsequent sections.

Data Resource Dependencies

Data resources follow the same dependency resolution as managed resources; setting the **depends_on** meta-argument in data blocks delays data reading until all dependencies are updated.



- Arguments referencing managed resources are treated as **depends_on** to ensure data sources access the latest information.
- This can be bypassed with indirect references through local values, except under specific conditions.

Terraform Built-in Functions

The Terraform language offers various built-in functions that can be used within expressions to modify and merge values.

Example

```
max(5, 12, 9)
```

The general syntax for function calls is a function name followed by comma-separated arguments in parentheses.

Note

Terraform does not allow the creation of user-defined functions; it only supports built-in functions.

Dynamic Blocks

They provide a powerful way to dynamically generate repeatable nested configuration blocks within resources, data resources, providers, and provisioners.

Example

```
resource "aws_elastic_beanstalk_environment" "tfenvtest"
{
  name = "tf-test-name"  # Expressions can be used here

  setting {
    # This is a literal block, typically static in nature
  }
}
```

- In Terraform, expressions are used to assign values with the format **name = expression.**
- Complex resources often require multiple nested configuration blocks beyond simple assignments.

An example is an AWS Elastic Beanstalk environment, which may need various settings configurations, each in a nested block.

Dynamic Block Structure

A dynamic block simplifies the creation of multiple instances of a block using a single template-like structure. The below sample code shows how it is structured using the same AWS Elastic Beanstalk example:

Example

```
resource "aws_elastic_beanstalk_environment" "tfenvtest" {
  name                = "tf-test-name"
  application         = "${aws_elastic_beanstalk_application.tftest.name}"
  solution_stack_name = "64bit Amazon Linux 2018.03 v2.11.4 running Go 1.12.6"

  dynamic "setting" {
    for_each = var.settings  # Iterates over each item in settings
    content {
      namespace = setting.value["namespace"]
      name      = setting.value["name"]
      value     = setting.value["value"]
    }
  }
}
```

Key Elements of Dynamic Blocks

Label	Specifies the type of block being dynamically generated, indicated by a label following the **dynamic** keyword
For_each	Iterates over each element in a collection or map, determining the number of times the block will be instantiated
Content	Defines the contents of each dynamically generated block, using the current item from the **for_each** iteration
Iterator	Sets a custom variable name for each element (optionally) during iteration and defaults to the label name if unspecified
Labels argument	Assigns additional labels to each generated block (optionally), utilizing values from the iterator

Constraints and Capabilities

Below are some important considerations regarding the constraints and capabilities of Terraform's dynamic configuration:

Integration	Dynamic blocks can only generate configurations that are valid within the context of the enclosing resource, data source, provider, or provisioner.
Meta-arguments	Dynamic blocks cannot be used to generate meta-arguments like **lifecycle** or **provisioner** blocks. These require earlier processing by Terraform before expressions can be safely evaluated.
Complex collections	The **for_each** argument can use any expression that results in a collection suited for iteration, including those transformed by functions like **flatten** or **setproduct**.

Multi-Level Nested Block Structures

They enable the dynamic generation of complex configurations, particularly for resources requiring multiple layers of nested settings.

- This approach is especially effective for managing resources with extensive and variable configurations.
- **Example:** An application includes load balancers that feature multiple origin groups and origins.

Multi-Level Nested Block Structures: Example

Suppose there is a complex variable structure for load balancer configurations that includes multiple origin groups, each with its own set of origins.

This structure can be dynamically represented in Terraform using nested dynamic blocks:

Example

```
variable "load_balancer_origin_groups" {
  type = map(object({
    origins = set(object({
      hostname = string
    }))
  }))
}
```

Multi-Level Nested Block Structures: Example

Example

```
resource "some_resource" {
  # Assuming this resource accepts nested blocks for origin groups and origins
  dynamic "origin_group" {
    for_each = var.load_balancer_origin_groups
    content {
      name = origin_group.key  # Naming each group by its map key

      dynamic "origin" {
        for_each = origin_group.value.origins
        content {
          hostname = origin.value.hostname  # Setting hostname for each origin
        }
      }
    }
  }
}
```

Key Considerations for Nested Dynamic Blocks

Iterator symbols → Managing iterator symbols (like **origin_group** and **origin**) is essential for clarity in accessing properties at different nesting levels.

Naming and scope → Utilizing the iterator argument helps differentiate nested blocks and prevent conflicts arising from similar block names.

Assisted Practice

Implementing Terraform Built-in Functions **Duration: 10 Min.**

Problem statement:

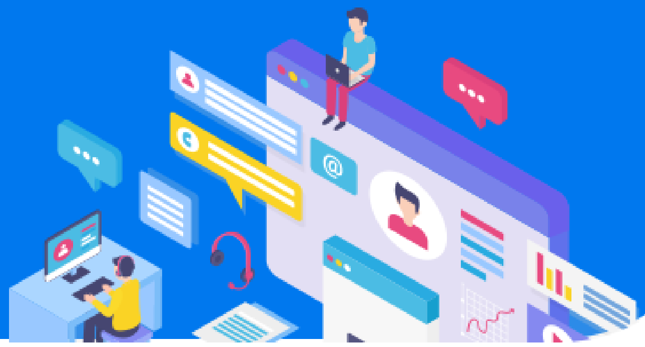
You have been assigned a task to implement Terraform built-in functions to manipulate and manage data efficiently in infrastructure configurations

Outcome:

By completing this demo, you will gain proficiency in implementing Terraform built-in functions to manipulate and manage data efficiently in infrastructure configurations.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Utilize basic numerical functions
2. Manipulate strings using Terraform functions
3. Implement the cidrsubnet function to create subnets

Assisted Practice

Working with Dynamic Data Blocks **Duration: 10 Min.**

Problem statement:

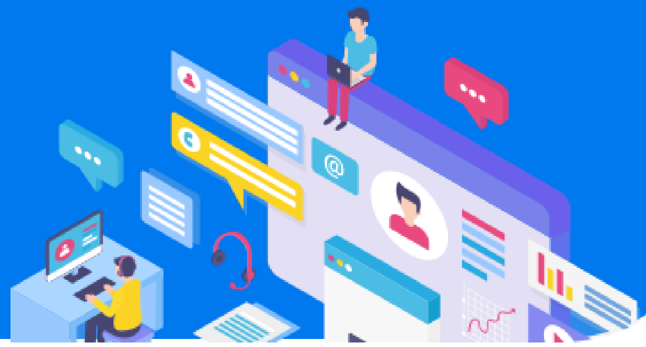
You have been assigned a task to implement dynamic blocks and local variables in Terraform for efficient and flexible infrastructure configuration management

Outcome:

By completing this demo, you will gain proficiency in implementing dynamic blocks and local variables in Terraform for efficient and flexible infrastructure configuration management.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Set up basic AWS infrastructure
2. Implement dynamic blocks for security groups
3. Utilize local variables for resource configuration
4. Apply configuration changes

Quick Check

You are tasked with creating a dynamic configuration in Terraform that retrieves data from an external source and uses built-in functions to manipulate the data. Which approach should you take?

A. Define a static block, use a built-in function to retrieve data, and hardcode the values.

B. Use a data block to fetch external data, apply a built-in function to process the data, and implement a dynamic block for repeated configurations

C. Create local variables to store data, use a built-in function to manipulate it, and manually repeat the block for each configuration

D. Fetch external data using environment variables, manipulate it with dynamic blocks, and use a static block for configuration

Graphs

Terraform Graph

The **terraform graph** command is a powerful tool for visualizing the relationships and dependencies between the resources and data elements defined in a Terraform configuration.

It uses the DOT language for graph description, which can be rendered into visual graphs using tools like Graphviz.

Command and Usage

By default, the terraform graph generates a simplified graph that primarily shows the dependency order of resources and data blocks within the Terraform configuration.

Syntax

```
terraform graph [options]
```

This simplification is focused on resource interdependencies, ideal for understanding the basic structure and execution plan.

Options

They are used to modify the behavior of the command to suit different needs or to provide different levels of detail in the output.

Common options for terraform graph include:

-type=<type>

-draw-cycles

-plan=<path to plan file>

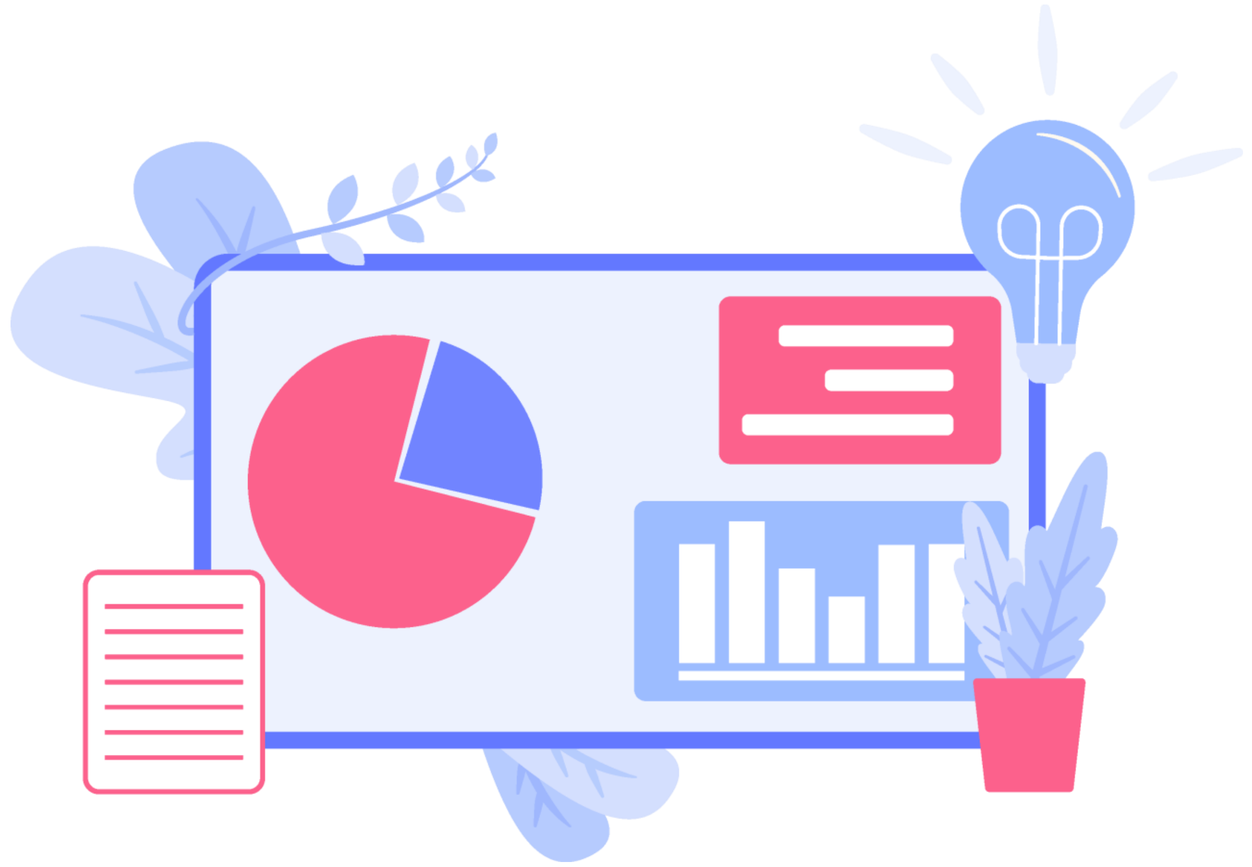
-type=<type>

It specifies the type of graph to generate, which can vary based on the Terraform operation or phase you are interested in visualizing. Common options for **-type=<type>** include:

plan	Generates a graph based on the plan phase, showing planned changes
apply	Generates a graph that would be applicable during the apply phase, detailing resource creation, update, or destruction
plan-refresh-only	Creates a graph showing dependencies relevant only to the refresh phase, which updates the state file with real-world resources
plan-destroy	Shows what the graph will look like when a terraform destroy operation is planned, illustrating resources that will be removed

-draw-cycles

It adds visual indications (colored edges) to the graph to highlight dependency cycles, which are circular dependencies that could cause Terraform operations to fail.



This is particularly useful when troubleshooting complex configurations that result in cycle errors during the plan or apply phases.

-plan=<path to plan file>

It uses a specific saved Terraform plan to generate the graph. This is useful when you want to visualize the impact of a pre-calculated plan without running a new plan during the session.

It sets the graph type to **apply**, as it visualizes the actions that Terraform would take if the plan were applied.

Generating Images

The output of **terraform graph** is in the DOT language, which can be processed by Graphviz tools to produce visual graphs.

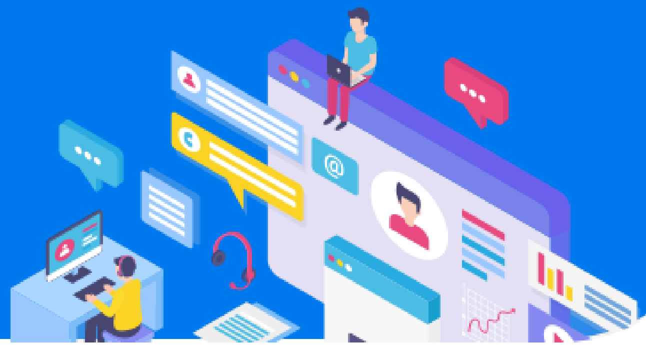
To create an image from the DOT output, use the dot command from Graphviz:

Syntax

```
terraform graph -type=plan | dot -Tpng >graph.png
```

This command pipelines the output of terraform graph into Graphviz, which renders it into a PNG image file.

Assisted Practice



Working with Graphs **Duration: 10 Min.**

Problem statement:

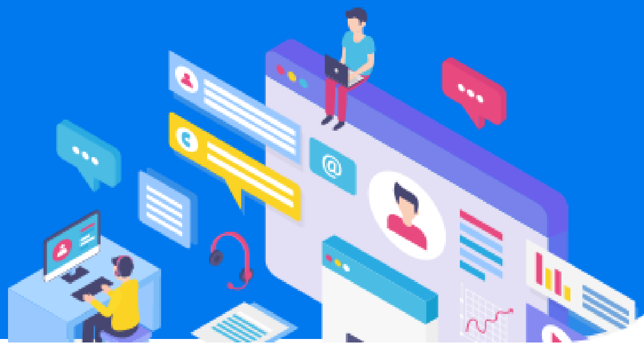
You have been assigned a task to create and visualize Terraform resource dependency graphs for better infrastructure management

Outcome:

By completing this demo, you will gain proficiency in creating and visualizing Terraform resource dependency graphs for better infrastructure management.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Set up infrastructure with dependencies
2. Generate and visualize the resource graph

ASSISTED PRACTICE

Terraform Resource Lifecycles

Terraform supports parallel resource management through its resource graph, which optimizes deployment speeds.

The resource graph determines Terraform's resource creation and destruction sequence, which is generally suitable but sometimes requires adjustments to the default lifecycle behavior.

Lifecycle Arguments

These are special directives within Terraform that customize behavior for resource creation, updating, and destruction. Key lifecycle arguments include:

prevent_destroy	Prevents the accidental deletion of critical resources
create_before_destroy	Ensures that new resources are created before the old ones are destroyed during updates, minimizing downtime
ignore_changes	Allows excluding certain resource attributes from triggering updates

Creating Infrastructure

To create the infrastructure with Terraform, follow these steps:

Step 1: Clone the example repository

Clone HashiCorp's example repository, containing essential Terraform files (main.tf, variables.tf, terraform.tfvars) to set up an EC2 instance and a security group

```
git clone https://github.com/hashicorp/learn-terraform-lifecycle-management
cd learn-terraform-lifecycle-management
```

Creating Infrastructure

Step 2: Configure AWS region

Confirm that the AWS CLI is configured to the correct region for deploying resources. This can be checked and set using the AWS CLI

```
aws configure get region
# If the region is not set to us-east-2, you can set it using:
aws configure set region us-east-2
```

Creating Infrastructure

Step 3: Update terraform variables

Edit the **terraform.tfvars** file to ensure the region matches your AWS CLI configuration

```
region = "us-east-2"
```

Creating Infrastructure

Step 4: Review and initialize terraform configuration

Open and review the **main.tf** file to understand the following resources being created:

EC2 instance	Is configured to use a specific AMI and instance type, with user data for installing and configuring Apache to serve on port 8080
Security group	Allows inbound TCP traffic on port 8080 and unrestricted outbound traffic

Creating Infrastructure

Example

```
##...

resource "aws_instance" "example" {
  ami                    = data.aws_ami.ubuntu.id
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.sg_web.id]
  user_data              = <<-EOF
              #!/bin/bash
              apt-get update
              apt-get install -y apache2
              sed -i -e 's/80/8080/' /etc/apache2/ports.conf
              echo "Hello World" > /var/www/html/index.html
              systemctl restart apache2
              EOF
  tags = {
    Name          = "terraform-learn-state-ec2"
    drift_example = "v1"
  }
}
```

Creating Infrastructure

Example

```
resource "aws_security_group" "sg_web" {
  name = "sg_web"
  ingress {
    from_port   = "8080"
    to_port     = "8080"
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  // connectivity to ubuntu mirrors is required to run `apt-get update` and `apt-get install
apache2`
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Initialize the Terraform configuration to install necessary provider plugins

```
terraform init
```

Creating Infrastructure

Step 5: Apply terraform configuration

Execute the **terraform apply** command to deploy the AWS resources. Terraform will show a plan and ask for confirmation before making any changes:

```
terraform apply
# Confirm the action by typing 'yes' when prompted
```

Creating Infrastructure

Step 6: Verify deployment

Once the apply operation completes, verify the resources created by Terraform using:

```
terraform state list
# Outputs might include:
# data.aws_ami.ubuntu
# aws_instance.example
# aws_security_group.sg_web
```

Preventing Resource Deletion

The **prevent_destroy** lifecycle attribute in Terraform is used to safeguard resources from accidental deletion.

Add **prevent_destroy** to the EC2 instance, as shown below:

```
resource "aws_instance" "example" {
  ami                    = data.aws_ami.ubuntu.id
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.sg_web.id]
  user_data              = <<-EOF
              #!/bin/bash
              apt-get update
              apt-get install -y apache2
              sed -i -e 's/80/8080/' /etc/apache2/ports.conf
              echo "Hello World" > /var/www/html/index.html
              systemctl restart apache2
              EOF
  tags = {
    Name          = "terraform-learn-state-ec2"
    drift_example = "v1"
  }

+ lifecycle {
+   prevent_destroy = true
+ }
}
```

Preventing Resource Deletion

Execute **terraform destroy** to observe the behavior

- Attempting to destroy resources with terraform destroy will result in an error if **prevent_destroy** is set, indicating the resource cannot be removed.

- This attribute is particularly useful when changes force a replacement, potentially causing downtime.

Creating Resources before They Are Destroyed

The **create_before_destroy** lifecycle attribute can be used to minimize downtime during infrastructure updates.

Step 1: Update the security group configuration

Modify the security group to allow traffic on port 80 instead of port 8080, as shown below; this change will update the security group rules to redirect traffic properly:

```
resource "aws_security_group" "sg_web" {
  name = "sg_web"
  ingress {
-   from_port   = "8080"
+   from_port   = "80"
-   to_port   = "8080"
+   to_port   = "80"
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ## ...
}
```

Creating Resources before They Are Destroyed

Step 2: Update EC2 instance configuration

Add the **create_before_destroy** attribute to ensure that a new instance is provisioned before the existing one is destroyed, as shown below; this attribute is crucial to avoid service interruption:

```
resource "aws_instance" "example" {
  ami                    = data.aws_ami.ubuntu.id
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.sg_web.id]
  user_data              = <<-EOF
              #!/bin/bash
              apt-get update
              apt-get install -y apache2
-             sed -i -e 's/80/8080/' /etc/apache2/ports.conf
              echo "Hello World" > /var/www/html/index.html
              systemctl restart apache2
              EOF
```

Creating Resources before They Are Destroyed

Step 2: Update EC2 instance configuration

```
tags = {
    Name           = "terraform-learn-state-ec2"
    Drift_example = "v1"
  }

  lifecycle {
-   prevent_destroy = true
+   create_before_destroy = true
  }
}
```

Creating Resources before They Are Destroyed

Step 3: Apply configuration changes

Run **terraform apply** to execute the changes. With the **create_before_destroy** attribute, Terraform will first create a new instance before destroying the old one.

```
terraform apply
# Confirm the action by typing 'yes' when prompted
```

Ignoring Changes

The **ignore_changes** attribute is used to manage external modifications that should not trigger Terraform to perform any operations. It is applied within a resource's lifecycle configuration.

Step 1: Update the resource tag externally

Update the tag on the AWS instance using the AWS CLI by running the following command:

```
aws ec2 create-tags --resources $(terraform output -raw instance_id) --tags Key=drift_example,Value=v2
```

This demonstrates an external change that typically would cause Terraform to update the resource to match the configuration defined in the code.

Ignoring Changes

Step 2: Modify terraform configuration

Add the **ignore_changes** attribute within the lifecycle block of the EC2 instance resource to ignore changes to the tags, as shown below:

```
resource "aws_instance" "example" {
##...
  lifecycle {
    create_before_destroy = true
+   ignore_changes        = [tags]
  }
}
```

This step ensures that Terraform does not try to revert the tag to its original state defined in Terraform files.

Ignoring Changes

Step 3: Apply configuration changes

Execute **terraform apply** to refresh the Terraform state:

```
terraform apply

#Output
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.Outputs:instance_id = "i-
0b2fd8a0df19c215d
public_ip = "18.116.49.153
```

This operation confirms the effectiveness of the **ignore_changes** attribute by not attempting to revert the external changes.

Ignoring Changes

Step 4: Verify the ignored change

Check the AWS instance details using the **terraform state show** to confirm that the **drift_example** tag reflects the externally applied value (v2) without any attempt by Terraform to overwrite it.

```
terraform state show aws_instance.example
```

Ignoring Changes

Output examination

The output should display the tag **drift_example** set to **v2**, demonstrating that Terraform has effectively ignored the external change.

```
# aws_instance.example:
resource "aws_instance" "example" {
    tags = {
        "Name"          = "terraform-learn-state-ec2"
        "drift_example" = "v2"
    }
}
```

Cleaning up the Resources

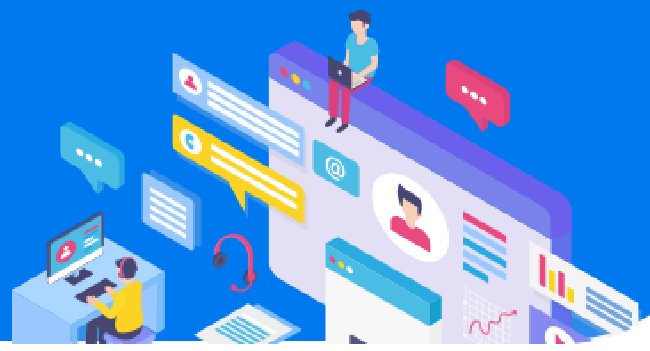
Conclude work with resources managed by Terraform, especially in learning environments or temporary setups, by properly cleaning up to prevent unnecessary costs or security risks.

Execute the Terraform destroy command

The following command initiates the destruction of all resources managed by the Terraform configuration:

```
terraform destroy
```

Assisted Practice



Managing Terraform Resource Lifecycle **Duration: 10 Min.**

Problem statement:

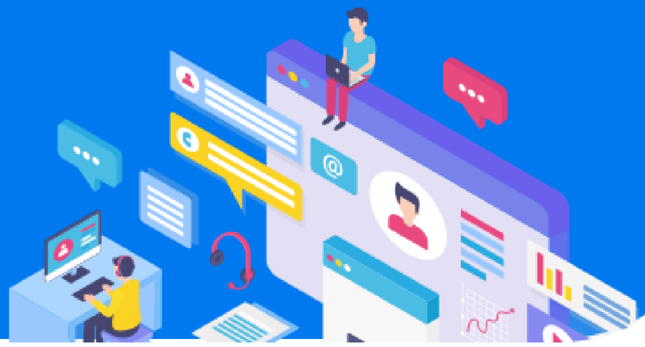
You have been assigned a task to manage the Terraform resource lifecycle effectively for infrastructure deployment

Outcome:

By completing this demo, you will gain proficiency in managing the Terraform resource lifecycle effectively for infrastructure deployment.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Set up a basic AWS infrastructure
2. Implement a dynamic security group with lifecycle modifications
3. Apply configuration changes
4. Implement and test **prevent_destroy**

Quick Check

You need to visualize the dependencies between resources in your Terraform configuration and ensure proper lifecycle management of these resources. Which steps should you follow?

A. Use terraform plan to view dependencies, then apply prevent_destroy to manage the lifecycle

B. Execute terraform graph to generate a visual dependency graph and use lifecycle arguments like create_before_destroy to manage resource updates

C. Run terraform validate to check dependencies and use environment variables to manage resource lifecycle

D. Initialize the configuration with terraform init and use terraform apply to manage resource dependencies and lifecycle

Key Takeaways

- The basic workflow of managing configurations in Terraform includes writing, initializing, validating, planning, applying configurations, optionally managing state, modifying, iterating, and destroying infrastructure.
- Terraform uses input variables, output values, and local values to manage and share configuration data.
- Terraform uses primitive, complex, and complex structured types for effective data management, enabling reusable, maintainable, and scalable configurations.
- Dynamic blocks provide a powerful way to dynamically generate repeatable nested configuration blocks within resources, data resources, providers, and provisioners.

Key Takeaways

- The Terraform language offers various built-in functions that can be used within expressions to modify and merge values.
- The terraform graph command is a powerful tool for visualizing the relationships and dependencies between the resources and data elements defined in a Terraform configuration.
- Common options for terraform graph include **-type=<type>**, **-draw-cycles**, and **-plan=<path to plan file>** to modify the command's behavior and detail level.
- Terraform lifecycle arguments, such as **prevent_destroy**, **create_before_destroy**, and **ignore_changes**, customize resource creation, updating, and destruction to ensure safe and efficient infrastructure management.

Implementing and Managing Terraform Configurations

Duration: 25 Min.

Project agenda: To deploy and manage a scalable web application on AWS infrastructure using Terraform for enhanced security, automation, and operational efficiency

Description: Imagine you are a cloud engineer tasked with implementing and managing AWS infrastructure using Terraform. The project involves setting up and initializing a Terraform configuration, defining and utilizing variables, locals, and outputs, and implementing resources with these variables and locals. Additionally, you will secure and manage sensitive data, work with collections and structure types, utilize Terraform's built-in functions and dynamic blocks, and generate and visualize a resource graph. This project aims to provide a comprehensive understanding of Terraform configuration and management practices, reinforcing key concepts and best practices.

Implementing and Managing Terraform Configurations

Duration: 25 Min.

Perform the following:
1. Set up and initialize Terraform configuration
2. Define variables, locals, and outputs
3. Implement resources with variables and locals
4. Secure and manage sensitive data
5. Utilize collections and structure types
6. Utilize Terraform built-in functions and dynamic blocks
7. Generate and visualize the resource graph

Expected Deliverables: A fully deployed AWS web application infrastructure using Terraform with initialized configurations, security settings for sensitive data, and a visual resource dependency graph

Thank You