Infrastructure-as-Code & Terraform

TEKsystems Global Services

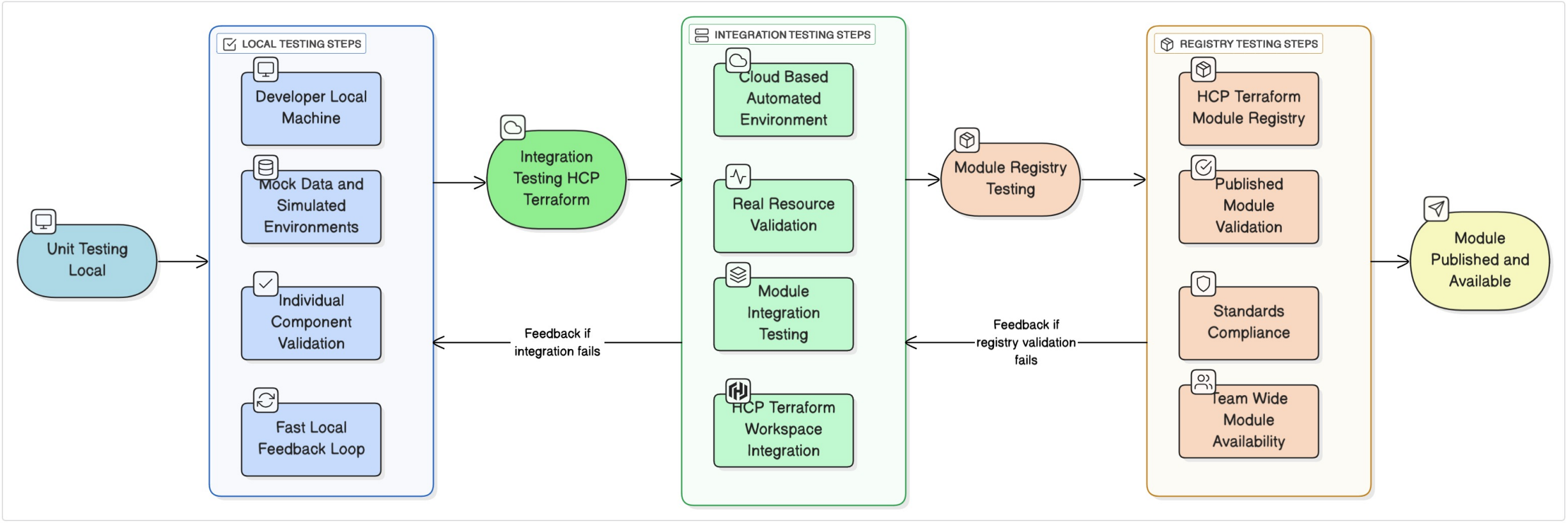
WORKFORCE DEVELOPMENT

Terraform Testing Framework Overview

Terraform tests allow authors to validate that module configuration updates do not introduce breaking changes. Tests run against test-specific, short-lived resources, preventing any risk to your existing infrastructure or state.

The testing framework is available in Terraform v1.6.0 and later, providing a built-in way to validate your infrastructure as code before it reaches production environments.

Terraform Testing Architecture



Why Test Terraform Configurations?

Testing your Terraform configurations provides several critical advantages for infrastructure management. By implementing a testing strategy, you can catch configuration errors before they affect production infrastructure, ensuring that your deployments remain stable and reliable.

Tests run against isolated, temporary resources without impacting existing systems, giving you confidence to experiment and validate changes. This approach serves as both a safety net and a documentation tool, as tests become living examples of how your modules should behave under various conditions.

How Terraform Testing Works

Terraform tests use .tftest.hcl or .tftest.json files that contain:

- Test blocks configure overall test execution (e.g., parallel vs. sequential)
- Run blocks execute specific Terraform operations (plan/apply) with assertions
- Variables blocks set test-specific input values
- Provider blocks configure test-specific provider settings

Tests can run in two modes: integration testing (creating real infrastructure) or unit testing (plan-only operations).

Testing Approaches

Integration Testing (Default)

- Creates real infrastructure during test execution
- Tests complete Terraform operations
- Validates end-to-end functionality
- Uses command = apply (default)

Unit Testing

- Runs only plan operations without creating resources
- Tests logical operations and custom conditions
- Faster execution and lower cost
- Uses command = plan

Test File Discovery and Components

Terraform automatically identifies test files through their specific file extensions: .tftest.hcl or .tftest.json. These files serve as containers for your infrastructure testing logic.

Every test file is organized into several key sections:

- Test blocks (optional, maximum of one)
- Run blocks (required, can have multiple)
- Variables blocks (optional, maximum of one)
- Provider blocks (optional, can have multiple)

Terraform processes run blocks one after another by default. However, you can configure them to run simultaneously using parallel execution features. Each run block represents a complete Terraform operation within your test configuration directory.

Terraform Test Execution Order

Terraform handles the processing order of variables and provider blocks automatically, regardless of where you place them in your test file. All configuration values are loaded and processed at the start of the test operation.

Following best practices, it's recommended to place your variables and provider blocks at the top of your test file. This organization creates a clear structure and ensures all dependencies are properly initialized before any test execution begins.

Testing AWS EC2 Instance Configuration

The following example demonstrates a basic Terraform configuration that creates an AWS EC2 instance, using an input variable to modify the instance type.

We will create a test file that validates the instance type is configured as expected.

```
# main.tf
provider "aws" {
    region = "us-west-2"
}

variable "instance_type" {
  type = string
}

resource "aws_instance" "web" {
  ami           = "ami-12345678"
  instance_type = var.instance_type
}

output "instance_type" {
  value = aws_instance.web.instance_type
}
```

Test File for Validation

This test file executes a Terraform plan operation to validate the EC2 instance configuration, ensuring the instance type logic works correctly by comparing the configured type against the expected value.

```
# test_instance_type.tftest.hcl
variables {
  instance_type = "t2.micro"
}

run "test_instance_type" {
  command = plan

  assert {
    condition     = aws_instance.web.instance_type == "t2.micro"
    error_message = "Instance type did not match expected"
  }
}
```

Test Block Configuration

The optional test block defines the configuration of the test file, allowing you to configure how the framework executes its runs.

Key Configuration Options:

- parallel: Boolean attribute that enables simultaneous execution of eligible run blocks
- Default value: false (sequential execution)
- When true: Terraform executes all eligible run blocks simultaneously

```
# with_config.tftest.hcl
test {
  parallel = true
}
```

Run Blocks Overview

Run blocks are the core execution units of Terraform tests, simulating Terraform commands within your configuration directory.

Each run block represents a complete Terraform operation and contains the logic for testing your infrastructure configuration.

```
variables {
  instance_type = "t2.micro"
}

run "first" {
  assert {
    condition     = aws_instance.example.instance_type == "t2.micro"
    error_message = "Instance type should be t2.micro"
  }
}
```

Run Block Command and Options

The command attribute and plan_options block tell Terraform which command and options to execute for each run block.

The default operation, if you do not specify a command attribute or the plan_options block, is a normal Terraform apply operation.

Attribute	Description	Default Value
command	Specifies the Terraform operation (plan or apply)	apply
plan_options.mode	Sets planning mode (normal or refresh-only)	normal
plan_options.refresh	Controls state refresh behavior	true
plan_options.replace	List of resources to force replacement	-
plan_options.target	List of resources to target	-
state_key	Controls which state file to use	-
parallel	Enables parallel execution	false

Parallel Execution Configuration

This example demonstrates how parallel execution works in Terraform tests. The test block sets parallel = true globally, but individual run blocks can override this setting.

The first two run blocks execute in parallel, while the third run block sets parallel = false, creating a synchronization point that forces it to wait for all preceding runs to complete.

```
# parallel_execution.tftest.hcl
test {
  parallel = true
}

variables {
  instance_type = "t2.micro"
}

run "first" {
  assert {
    condition     = aws_instance.example.instance_type == "t2.micro"
    error_message = "Instance type should be t2.micro"
  }
}

run "second" {
  assert {
    condition     = aws_instance.example.instance_type == "t2.micro"
    error_message = "Instance type should be t2.micro"
  }
}

run "third" {
  parallel = false
  assert {
    condition     = aws_instance.example.instance_type == "t2.micro"
    error_message = "Instance type should be t2.micro"
  }
}
```

Parallel Execution Configuration II

Terraform can execute run blocks in parallel when the parallel attribute is set to true. Run blocks can execute simultaneously if they don't reference each other's outputs, don't share the same state file, and have parallel=true set.

Parallel execution can be configured globally in the test block or individually in run blocks. A single run block with parallel=false creates a synchronization point, dividing the workflow into groups that must complete sequentially.

```
test {
  parallel = true
}

run "primary_db" {
  state_key = "primary"
}

run "secondary_db" {
  state_key = "secondary"
}

run "site_one" {
  parallel = false  # Creates synchronization point
}
```

Assertions in Terraform Tests

Assertions are the core validation mechanism in Terraform tests, allowing you to verify that your infrastructure configuration behaves as expected. Each assert block contains a condition that must evaluate to true for the test to pass.

Assert blocks have two required attributes:

- condition: A boolean expression that must evaluate to true
- error_message: A descriptive message displayed when the assertion fails

```
run "test_instance_configuration" {
  command = plan

  assert {
    condition     = aws_instance.example.instance_type == "t2.micro"
    error_message = "Instance type should be t2.micro"
  }
```

Variables in Terraform Tests

Variables blocks in Terraform tests allow you to set input values that will be used throughout your test execution. These blocks provide a way to configure your test environment and override default values from your main configuration.

Variables defined in the variables block serve as the default values for your entire test file. However, individual run blocks can override these values using their own variables block, allowing you to test different scenarios with the same configuration.

```
variables {
  bucket_prefix = "test"
  environment   = "development"
  region        = "us-west-2"
}

run "test_bucket_creation" {
  command = plan

  variables {
    bucket_prefix = "production"  # Override for this specific run
  }

  assert {
    condition     = aws_s3_bucket.bucket.bucket == "production-bucket"
    error_message = "Bucket name should use production prefix"
  }
}
```

Variable References in Terraform Tests

Variables in Terraform tests can reference outputs from earlier run blocks and variables defined at higher precedence levels, enabling powerful data flow between test executions.

- Run block variables can reference file-level variables
- Run block variables can reference outputs from previous run blocks
- File-level variables can only reference global variables
- This enables sharing values across multiple run blocks and passing data between different test executions

```
variables {
  global_value = "some value"
}

run "run_block_one" {
  variables {
    local_value = var.global_value
  }

  # Test assertions here
}

run "run_block_two" {
  variables {
    local_value = run.run_block_one.output_one
  }

  # Test assertions here
}
```

Provider Configuration

Providers in Terraform tests can be configured and overridden to control how your infrastructure is deployed during testing. This allows you to customize provider settings specifically for your test environment without affecting your main configuration.

Provider configuration in tests gives you the flexibility to test your infrastructure with different provider configurations, such as using specific regions, credentials, or other provider-specific settings that may differ from your production environment.

```
# customised_provider.tftest.hcl

provider "aws" {
  region = "eu-central-1"
}

variables {
  bucket_prefix = "test"
}

run "valid_string_concat" {
  command = plan

  assert {
    condition     = aws_s3_bucket.bucket.bucket == "test-bucket"
    error_message = "S3 bucket name did not match expected"
  }
}
```

Advanced Provider References in Terraform Tests

Starting from Terraform v1.7.0, provider blocks can reference test file variables and run block outputs, enabling sophisticated provider setup workflows. This capability allows you to dynamically configure providers based on information retrieved from other providers or previous test executions.

This feature enables scenarios where one provider can retrieve credentials or configuration data that is then used to initialize a second provider.

```
provider "vault" {
  # ... vault configuration ...
}

provider "aws" {
  region     = "us-east-1"
  access_key = run.vault_setup.aws_access_key
  secret_key = run.vault_setup.aws_secret_key
}

run "vault_setup" {
  module {
    source = "./testing/vault-setup"
  }
}

run "use_aws_provider" {
  # This run block can use both providers
}
```

Module Configuration

Modules in Terraform tests can be modified to execute different configurations within each run block.

The module block in test files is simplified compared to traditional Terraform modules, supporting only the source and version attributes. Other module configuration options are handled through alternative attributes and blocks within the run block, providing flexibility while maintaining test simplicity.

```
run "test_module_v1" {
  module {
    source  = "./modules/example"
    version = "1.0.0"
  }

  # Other configuration handled in run block
}

run "test_module_v2" {
  module {
    source  = "./modules/example"
    version = "2.0.0"
  }

  # Test different module version
}
```

Advanced Module Usage

When executing alternate modules through the module block, all other run block attributes and blocks remain fully supported. Assert blocks execute against values from the alternate module, enabling comprehensive testing of different module configurations.

Two primary use cases for the modules block include setup modules that create required infrastructure for testing, and loading modules that validate secondary infrastructure like data sources.

```
# instance_count.tftest.hcl

run "setup" {
  # Create the VPC we will use later.

  module {
    source = "./testing/setup"
  }
}

run "execute" {
  # This is empty, we just run the configuration under test using all the default settings.
}

run "verify" {
  # Load and count the instances created in the "execute" run block.

  module {
    source = "./testing/loader"
  }

  assert {
    condition = length(data.aws_instances.instances.ids) == 3
    error_message = "created the wrong number of EC2 instances"
  }
}
```

Module State Management in Terraform Tests

Terraform maintains multiple state files in memory during test execution, with each state file assigned a unique state key for internal tracking. The state key can be overridden using the state_key attribute of a run block to control state file sharing between different modules.

There is always at least one state file for the main configuration under test, shared by all run blocks without alternate modules.

```
run "setup" {
  module {
    source = "./testing/setup"
  }
}

run "init" {
  # Uses main configuration state file
}

run "update_setup" {
  # Reuses setup module state file
  module {
    source = "./testing/setup"
  }
}
```

State Key Override Examples

The state_key attribute allows you to override Terraform's default state file behavior. By default, Terraform creates separate state files for the main configuration and each alternate module, but you can force multiple run blocks to share the same state file regardless of their module source.

This capability is particularly useful when you want to share infrastructure state between run blocks that reference different modules.

```
run "setup" {
  state_key = "main"
  module {
    source = "./testing/setup"
  }
}

run "init" {
  state_key = "main"  # Shares state with setup run
}
```

Module Cleanup in Terraform Tests

Terraform automatically destroys all resources created during test execution when a test file concludes. The destruction order follows reverse run block execution order, which is crucial for maintaining proper resource dependencies during cleanup.

When using alternate modules, Terraform destroys resources in reverse order of when their state files were last referenced. This ensures that dependent resources are destroyed before the resources they depend on.

```
run "setup" {
  # Creates S3 bucket - destroyed LAST
  module {
    source = "./testing/setup"
  }
}

run "execute" {
  # Creates objects in bucket - destroyed SECOND
}

run "verify" {
  # References loader module - destroyed FIRST
  module {
    source = "./testing/loader"
  }
}
```

Expecting Failures in Terraform Tests

Terraform tests support the expect_failures attribute to test failure scenarios. By default, any failed custom conditions cause test failure, but expect_failures allows you to specify which checkable objects should fail for the test to pass.

The expect_failures attribute accepts a list of resources, data sources, check blocks, input variables, and outputs that should fail their custom conditions. This enables testing both positive and negative scenarios within the same test file.

```
variable "input" {
  type = number

  validation {
    condition = var.input % 2 == 1
    error_message = "must be odd number"
  }
}
```

Expecting Failures Implementation

When using expect_failures, be aware that custom conditions (except check blocks) halt Terraform execution. This means you can only reliably include a single checkable object per run block, unless you're only testing check block failures.

Assertions can still be written alongside expect_failures, but they must only reference values computed before the expected failure occurs. Use references or depends_on meta-arguments to manage execution order.

```
variables {
  input = 1
}

run "one" {
  # The variable defined above is odd, so we expect the validation to pass.

  command = plan
}

run "zero" {
  # This time we set the variable is even, so we expect the validation to fail.

  command = plan

  variables {
    input = 0
  }

  expect_failures = [
    var.input,
  ]
}
```

Expecting Failures with Apply Commands

When using expect_failures with command = apply, be aware that the run block will fail if the custom condition fails during the plan phase. This occurs because the apply operation cannot proceed after a failed plan, even when the failure was expected.

While Terraform may not execute some custom conditions during planning (when they depend on computed attributes), it's generally recommended to use expect_failures only with command = plan operations.

```
# This will FAIL because plan fails before apply can run
run "test_apply_failure" {
  command = apply

  variables {
    input = 0  # Even number, will fail validation
  }

  expect_failures = [
    var.input,  # Expects failure but apply never runs
  ]
}

# This works correctly with plan
run "test_plan_failure" {
  command = plan

  variables {
    input = 0  # Even number, will fail validation
  }

  expect_failures = [
    var.input,  # Expects failure and plan shows it
  ]
}
```

Test Mocking

Terraform lets you mock providers, resources, and data sources for your tests. This allows you to test parts of your module without creating infrastructure or requiring credentials. In a Terraform test, a mocked provider or resource will generate fake data for all computed attributes that would normally be provided by the underlying provider APIs.

Mocking functionality can only be used with the terraform test language. Readers of this documentation should be familiar with the testing syntax and language features.

Mock Providers

In Terraform tests, you can mock a provider with the mock_provider block. Mock providers return the same schema as the original provider and you can pass the mocked provider to your tests in place of the matching provider.

All resources and data sources retrieved by a mock provider will set the relevant values from the configuration, and generate fake data for any computed attributes.

```
# bucket_name.tftest.hcl
mock_provider "aws" {}

run "sets_correct_name" {
  variables {
    bucket_name = "my-bucket-name"
  }

  assert {
    condition     = aws_s3_bucket.my_bucket.bucket == "my-bucket-name"
    error_message = "incorrect bucket name"
  }
}
```

Mock Provider Behavior

From the perspective of a plan or apply operation executed in a Terraform test file, the mocked provider is creating actual resources with values that match the configuration.

These resources are stored in the Terraform state files that terraform test creates and holds in memory during test executions.

```
# main.tf
resource "aws_s3_bucket" "my_bucket" {
  bucket = var.bucket_name
}

# test.tftest.hcl
mock_provider "aws" {}

run "test_bucket_creation" {
  variables {
    bucket_name = "test-bucket"
  }

  assert {
    condition     = aws_s3_bucket.my_bucket.bucket == "test-bucket"
    error_message = "Bucket name should match input variable"
  }

  assert {
    condition     = can(aws_s3_bucket.my_bucket.arn)
    error_message = "Bucket should have a generated ARN"
  }
}
```

Mixed Real and Mocked Providers

You can use mocked providers and real providers simultaneously in a Terraform test. The following example defines two AWS providers, one real and one mocked.

You must provide an alias to one of them, as they share the same global aws provider namespace. You can then use the providers attribute in test run blocks to customize which AWS provider to use for each run block.

```
# mocked_providers.tftest.hcl
provider "aws" {}

mock_provider "aws" {
  alias = "fake"
}

run "use_real_provider" {
  providers = {
    aws = aws
  }
}

run "use_mocked_provider" {
  providers = {
    aws = aws.fake
  }
}
```

Mock Provider Data

You can specify specific values for targeted resources and data sources. In a mock_provider block, you can write any number of mock_resource and mock_data blocks.

Both the mock_resource and mock_data blocks accept a type argument that should match the resource or data source you want to provide values for. They also accept a defaults object attribute that you can use to specify the values that should be returned for specific attributes.

```
mock_provider "aws" {
  mock_resource "aws_instance" {
    defaults = {
      arn = "arn:aws:ec2:us-west-1:123456789012:instance/i-12345678"
    }
  }

  mock_data "aws_instance" {
    defaults = {
      arn = "arn:aws:ec2:us-west-1:123456789012:instance/i-12345678"
    }
  }
}
```

Mock Provider Overrides

In addition to mocking providers, you can use the following block types to override specific resources, data sources, and modules:

• override_resource: Override the values of a resource. Terraform does not call the underlying provider.

• override_data: Override the values of a data source. Terraform does not call the underlying provider.

• override_module: Override the outputs of a module. Terraform does not create any resource in the module.

```
mock_provider "aws" {}

override_data {
  target = data.aws_availability_zones.available
  values = {
    names = ["us-west-1a", "us-west-1b", "us-west-1c"]
  }
}
```

Terraform Unit Test vs integration Test

Unit tests check the logic inside your Terraform modules without touching real cloud resources. Integration tests apply the configuration for real and validate actual behavior in the environment.

Unit Test
Tests module logic, variable defaults, outputs, and structure
Uses tools like Terraform Validate, Check, or Terratest mocks

Integration Test
Deploys real infrastructure and verifies it works
Confirms permissions, networking, dependencies, and runtime behavior

How Unit and Integration Tests Fit Into Workflow

Unit tests support Test Driven Development. You write the tests before the module is built, ensuring the module behaves exactly as expected. Integration tests validate the real infrastructure after deployment.

Unit tests (TDD): define expected behavior first, then write code to satisfy the tests. Unit tests catch logic errors early.

Integration tests: confirm the deployed infrastructure matches the intent of your code. Integration tests catch real-world issues like permissions, networking, and naming

Unit tests shape the module. Integration tests verify the infrastructure.

Integration Testing

```
tests/
├── integration.tftest.hcl      # Main integration test suite
└── setup/
    └── main.tf                 # Infrastructure created before tests run
```

The setup directory contains Terraform code that prepares shared infrastructure for the tests, such as subnets or security groups. The integration.tftest.hcl file defines test runs that apply your module with different variables and check the actual AWS resources created.

Integration Tests

The integration test file deploys real infrastructure and then verifies that the resources in AWS match the variables passed into each test run.

- The setup module is applied first to create required networking and security

- Each test run applies your module with different input variables

- Terraform then checks the actual AWS infrastructure created

```
# tests/integration.tftest.hcl

run "setup_infrastructure" {
  module {
    source = "./tests/setup"
  }
}

run "test_ec2_instance_creation" {
  command = apply

  variables {
    instance_count = 2
    instance_type  = "t2.micro"
    security_group_ids = [run.setup_infrastructure.security_group_id]
    subnet_ids = [run.setup_infrastructure.subnet_id]
  }

  # Test that instances are created
  assert {
    condition     = length(aws_instance.app[*].id) == 2
    error_message = "Should create exactly 2 EC2 instances"
  }

  # Test that instance IDs are valid
  assert {
    condition     = alltrue([for id in aws_instance.app[*].id : can(regex("^i-", id))])
    error_message = "All EC2 instances should have valid instance IDs"
  }
}
```

Lab: Terraform Unit Testing