**What is a terraform?**

Terraform is an open-source infrastructure as code software tool that provides a consistent CLI workflow to manage hundreds of cloud services

Terraform is an infrastructure as code (IaC) tool that allows you to build, change, and version infrastructure safely and efficiently. This includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries

**Terraform basic operations**

There are 3 basic operations of terraform 1) Write

2) Plan

3) Apply

**Write**

Write infrastructure as code using declarative configuration files. HashiCorp Configuration Language (**HCL**) allows for concise descriptions of **resources** using ***blocks, arguments, and expressions.***

## Plan

Run terraform plan to check whether the execution plan for a configuration matches your expectations before provisioning or changing infrastructure.

## Apply

Apply changes to hundreds of cloud providers with terraform apply to reach the ***desired state of the configuration.***

## Features of Terraform

## i) Declarative config files –

## Define infrastructure as code to manage the full lifecycle — create new resources, manage existing ones, and destroy.

## ii) Installable modules - Provides access to community or different cloud partner modules *(terraform init)* from registry rather than writing code from scratch. Which help to reuse code instead writing it every time. (Its similar to any programming language where we are including external library and calling particular function or procedure from main program)

**e.g** This **s3\_user** module will create an IAM user and allow read access to all objects in the S3 bucket examplebucket

module "s3\_user" {

source = "cloudposse/iam-s3-user/aws"

namespace = "eg"

stage = "test"

name = "app"

s3\_actions = ["s3:GetObject"]

s3\_resources = "arn:aws:s3:::examplebucket/\*"

}

iii) **Plan and predict changes** - Terraform allows operators to safely and predictably make changes to infrastructure, with clearly mapped resource dependencies and **separation of plan and apply**.

The terraform plan command creates an execution plan, which lets you preview the changes that Terraform plans to make to your infrastructure. By default, when Terraform creates a plan it:

* Reads the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date.
* Compares the current configuration to the prior state and noting any differences.
* Proposes a set of change actions that should, if applied, make the remote objects match the configuration.

The plan command alone will not actually carry out the proposed changes, and so you can use this command to check whether the proposed changes match what you expected before you apply the changes or share your changes with your team for broader review.

If Terraform detects that no changes are needed to resource instances or to root module output values, terraform plan will report that no actions need to be taken.

## iv) Dependency graphing –

## Easily generate terraform plan, refresh state, and more, with Terraform config dependency graphing.

## v) State Management –

## State is a necessary requirement for Terraform to function. It is often asked if it is possible for Terraform to work without state, or for Terraform to not use state and just inspect cloud resources on every run

## Terraform requires some sort of database to map Terraform config to the real world. When you have a resource resource "aws\_instance" "foo" in your configuration, Terraform uses this map to know that instance i-abcd1234 is represented by that resource.

Therefore, for mapping configuration to resources in the real world, Terraform uses its own state structure.

Terraform expects that each remote object is bound to only one resource instance, which is normally guaranteed by Terraform being responsible for creating the objects and recording their identities in the state. If you instead import objects that were created outside of Terraform, you'll need to check yourself that each distinct object is imported to only one resource instance.

If one remote object is bound to two or more resource instances then Terraform may take unexpected actions against those objects, because the mapping from configuration to the remote object state has become ambiguous.

Terraform retains a copy of the most recent set of dependencies within the state. Now Terraform can still determine the correct order for destruction from the state when you delete one or more items from the configuration.

# The Core Terraform Workflow

The core Terraform workflow has three steps:

1. **Write** - Author infrastructure as code.
2. **Plan** - Preview changes before applying.
3. **Apply** - Provision reproducible infrastructure.

## Working as an Individual Practitioner

Let's first walk through how these parts fit together as an individual working on infrastructure as code.

### Step1 - Write

You write Terraform configuration just like you write code: in your editor of choice. It's common practice to store your work in a version controlled repository even when you're just operating as an individual.

*# Create repository*

*$ git init my-infra && cd my-infra*

*Initialized empty Git repository in /.../my-infra/.git/*

*# Write initial config*

*$ vim main.tf*

*# Initialize Terraform*

*$ terraform init*

*Initializing provider plugins...*

*# ...*

*Terraform has been successfully initialized!*

As you make progress on authoring your config, repeatedly running plans can help flush out syntax errors and ensure that your config is coming together as you expect.

*# Make edits to config*

*$ vim main.tf*

*# Review plan*

*$ terraform plan*

*# Make additional edits, and repeat*

*$ vim main.tf*

This parallels working on application code as an individual, where a tight feedback loop between editing code and running test commands is useful.

### Step2 - Plan

When the feedback loop of the Write step has yielded a change that looks good, it's time to commit your work and review the final plan.

*$ git add main.tf*

*$ git commit -m 'Managing infrastructure as code!'*

*[main (root-commit) f735520] Managing infrastructure as code! 1 file changed, 1 insertion(+)*

Because terraform apply will display a plan for confirmation before proceeding to change any infrastructure, that's the command you run for final review.

*$ terraform apply*

*An execution plan has been generated and is shown below.# ...*

### Step3 - Apply

After one last check, you are ready to tell Terraform to provision real infrastructure.

*Do you want to perform these actions?*

*Terraform will perform the actions described above. Only 'yes' will be accepted to approve. Enter a value: yes*

*# ...*

*Apply complete! Resources: 1 added, 0 changed, 0 destroyed.*

At this point, it's common to push your version control repository to a remote location for safekeeping.

*$ git remote add origin https://github.com/\*user\*/\*repo\*.git$ git push origin main*

This core workflow is a loop; the next time you want to make changes, you start the process over from the beginning.

Notice how closely this workflow parallels the process of writing application code or scripts as an individual? This is what we mean when we talk about Terraform enabling infrastructure as code.

## Working as a Team

Once multiple people are collaborating on Terraform configuration, new steps must be added to each part of the core workflow to ensure everyone is working together smoothly. You'll see that many of these steps parallel the workflow changes we make when we work on application code as teams rather than as individuals.

### [»](https://www.terraform.io/intro/core-workflow#write-1)Write

While each individual on a team still makes changes to Terraform configuration in their editor of choice, they save their changes to version control branches to avoid colliding with each other's work. Working in branches enables team members to resolve mutually incompatible infrastructure changes using their normal merge conflict workflow.

$ git checkout -b add-load-balancer

Switched to a new branch 'add-load-balancer'

Running iterative plans is still useful as a feedback loop while authoring configuration, though having each team member's computer able to run them becomes more difficult with time. As the team and the infrastructure grows, so does the number of sensitive input variables (e.g. API Keys, SSL Cert Pairs) required to run a plan.

To avoid the burden and the security risk of each team member arranging all sensitive inputs locally, it's common for teams to migrate to a model in which Terraform operations are executed in a shared Continuous Integration (CI) environment. The work needed to create such a CI environment is nontrivial, and is outside the scope of this core workflow overview, but a full deep dive on this topic can be found in our [Running Terraform in Automation](https://learn.hashicorp.com/tutorials/terraform/automate-terraform?in=terraform/automation&utm_source=WEBSITE&utm_medium=WEB_IO&utm_offer=ARTICLE_PAGE&utm_content=DOCS) guide.

This longer iteration cycle of committing changes to version control and then waiting for the CI pipeline to execute is often lengthy enough to prohibit using speculative plans as a feedback loop while authoring individual Terraform configuration changes. Speculative plans are still useful before new Terraform changes are applied or even merged to the main development branch, however, as we'll see in a minute.

### [»](https://www.terraform.io/intro/core-workflow#plan-1)Plan

For teams collaborating on infrastructure, Terraform's plan output creates an opportunity for team members to review each other's work. This allows the team to ask questions, evaluate risks, and catch mistakes before any potentially harmful changes are made.

The natural place for these reviews to occur is alongside pull requests within version control--the point at which an individual proposes a merge from their working branch to the shared team branch. If team members review proposed config changes alongside speculative plan output, they can evaluate whether the intent of the change is being achieved by the plan.

The problem becomes producing that speculative plan output for the team to review. Some teams that still run Terraform locally make a practice that pull requests should include an attached copy of speculative plan output generated by the change author. Others arrange for their CI system to post speculative plan output to pull requests automatically.

In addition to reviewing the plan for the proper expression of its author's intent, the team can also make an evaluation whether they want this change to happen now. For example, if a team notices that a certain change could result in service disruption, they may decide to delay merging its pull request until they can schedule a maintenance window.

### [»](https://www.terraform.io/intro/core-workflow#apply-1)Apply

Once a pull request has been approved and merged, it's important for the team to review the final concrete plan that's run against the shared team branch and the latest version of the state file.

This plan has the potential to be different than the one reviewed on the pull request due to issues like merge order or recent infrastructural changes. For example, if a manual change was made to your infrastructure since the plan was reviewed, the plan might be different when you merge.

It is at this point that the team asks questions about the potential implications of applying the change. Do we expect any service disruption from this change? Is there any part of this change that is high risk? Is there anything in our system that we should be watching as we apply this? Is there anyone we need to notify that this change is happening?

Depending on the change, sometimes team members will want to watch the apply output as it is happening. For teams that are running Terraform locally, this may involve a screen share with the team. For teams running Terraform in CI, this may involve gathering around the build log.

Just like the workflow for individuals, the core workflow for teams is a loop that plays out for each change. For some teams this loop happens a few times a week, for others, many times a day.

# Terraform vs. Other Software

# Terraform vs. Chef, Puppet, etc.

Configuration management tools install and manage software on a machine that already exists. Terraform is not a configuration management tool, and it allows existing tooling to focus on their strengths: bootstrapping and initializing resources.

Terraform focuses on the higher-level abstraction of the datacenter and associated services, while allowing you to use configuration management tools on individual systems. It also aims to bring the same benefits of codification of your system configuration to infrastructure management.

**If you are using traditional configuration management within your compute instances, you can use Terraform to configure bootstrapping software like cloud-init to activate your configuration management software on first system boot**.

# Terraform vs. CloudFormation, Heat, etc.

Tools like CloudFormation, Heat, etc. allow the details of an infrastructure to be codified into a configuration file. The configuration files allow the infrastructure to be elastically created, modified and destroyed. Terraform is inspired by the problems they solve.

Terraform similarly uses configuration files to detail the infrastructure setup, but it goes further by being both cloud-agnostic and enabling multiple providers and services to be combined and composed. For example, Terraform can be used to orchestrate an AWS and OpenStack cluster simultaneously, while enabling 3rd-party providers like Cloudflare and DNSimple to be integrated to provide CDN and DNS services. This enables Terraform to represent and manage the entire infrastructure with its supporting services, instead of only the subset that exists within a single provider.

Terraform also separates the planning phase from the execution phase, by using the concept of an execution plan. By running terraform plan, the current state is refreshed and the configuration is consulted to generate an action plan. The plan includes all actions to be taken: which resources will be created, destroyed or modified. It can be inspected by operators to ensure it is exactly what is expected. Using terraform graph, the plan can be visualized to show dependent ordering. Once the plan is captured, the execution phase can be limited to only the actions in the plan.

# Terraform vs. Boto, Fog, etc.

Libraries like Boto, Fog, etc. are used to provide native access to cloud providers and services by using their APIs. Some libraries are focused on specific clouds, while others attempt to bridge them all and mask the semantic differences. Using a client library only provides low-level access to APIs, requiring application developers to create their own tooling to build and manage their infrastructure.

Terraform is not intended to give low-level programmatic access to providers, but instead provides a high level syntax for describing how cloud resources and services should be created, provisioned, and combined. Terraform is very flexible, using a plugin-based model to support providers and provisioners, giving it the ability to support almost any service that exposes APIs.

Language Reference

# Variables and Outputs

The Terraform language includes a few kinds of blocks for requesting or publishing named values.

* [Input Variables](https://www.terraform.io/language/values/variables) serve as parameters for a Terraform module, so users can customize behavior without editing the source.
* [Output Values](https://www.terraform.io/language/values/outputs) are like return values for a Terraform module.
* [Local Values](https://www.terraform.io/language/values/locals) are a convenience feature for assigning a short name to an expression.

Input variables let you customize aspects of Terraform modules without altering the module's own source code. This allows you to share modules across different Terraform configurations, making your module composable and reusable.

When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables. When you declare them in [child modules](https://www.terraform.io/language/modules), the calling module should pass values in the module block.

If you're familiar with traditional programming languages, it can be useful to compare Terraform modules to function definitions:

* Input variables are like function arguments.
* [Output values](https://www.terraform.io/language/values/outputs) are like function return values.
* [Local values](https://www.terraform.io/language/values/locals) are like a function's temporary local variables.

## Declaring an Input Variable

Each input variable accepted by a module must be declared using a variable block:

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"

}

]

}

The label after the variable keyword is a name for the variable, which must be unique among all variables in the same module. This name is used to assign a value to the variable from outside and to reference the variable's value from within the module.

The name of a variable can be any valid [identifier](https://www.terraform.io/language/syntax/configuration#identifiers) except the following: source, version, providers, count, for\_each, lifecycle, depends\_on, locals.

These names are reserved for meta-arguments in [module configuration blocks](https://www.terraform.io/language/modules/syntax), and cannot be declared as variable names.

**Arguments**

Terraform CLI defines the following optional arguments for variable declarations:

* [default](https://www.terraform.io/language/values/variables#default-values) - A default value which then makes the variable optional.
* [type](https://www.terraform.io/language/values/variables#type-constraints) - This argument specifies what value types are accepted for the variable.

[string](https://www.terraform.io/language/values/variables#string), [number](https://www.terraform.io/language/values/variables#number), [bool](https://www.terraform.io/language/values/variables#bool), [list(<TYPE>)](https://www.terraform.io/language/values/variables#list), [set(<TYPE>)](https://www.terraform.io/language/values/variables#set), [map(<TYPE>)](https://www.terraform.io/language/values/variables#map), [object({<ATTR NAME> = <TYPE>, ... })](https://www.terraform.io/language/values/variables#object), [tuple([<TYPE>, ...])](https://www.terraform.io/language/values/variables#tuple)

* [description](https://www.terraform.io/language/values/variables#input-variable-documentation) - This specifies the input variable's documentation.
* [validation](https://www.terraform.io/language/values/variables#custom-validation-rules) - A block to define validation rules, usually in addition to type constraints.
* [sensitive](https://www.terraform.io/language/values/variables#suppressing-values-in-cli-output) - Limits Terraform UI output when the variable is used in configuration.
* [nullable](https://www.terraform.io/language/values/variables#disallowing-null-input-values) - Specify if the variable can be null within the module.

### Custom Validation Rules

This feature was introduced in Terraform CLI v0.13.0.

In addition to Type Constraints as described above, a module author can specify arbitrary custom validation rules for a particular variable using a validation block nested within the corresponding variable block:

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-\"." }}

The condition argument is an expression that must use the value of the variable to return true if the value is valid, or false if it is invalid. The expression can refer only to the variable that the condition applies to, and must not produce errors.

If the failure of an expression is the basis of the validation decision, use [the can function](https://www.terraform.io/language/functions/can) to detect such errors. For example:

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

validation { # regex(...) fails if it cannot find a match condition = can(regex("^ami-", var.image\_id)) error\_message = "The image\_id value must be a valid AMI id, starting with \"ami-\"." }}

If condition evaluates to false, Terraform will produce an error message that includes the sentences given in error\_message. The error message string should be at least one full sentence explaining the constraint that failed, using a sentence structure similar to the above examples.

Multiple validation blocks can be declared in which case error messages will be returned for all failed conditions.

### Suppressing Values in CLI Output

Setting a variable as sensitive prevents Terraform from showing its value in the plan or apply output, when you use that variable elsewhere in your configuration.

Terraform will still record sensitive values in the [state](https://www.terraform.io/language/state), and so anyone who can access the state data will have access to the sensitive values in cleartext. For more information, see [Sensitive Data in State](https://www.terraform.io/language/state/sensitive-data).

Declare a variable as sensitive by setting the sensitive argument to true:

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

}

Any expressions whose result depends on the sensitive variable will be treated as sensitive themselves, and so in the above example the two arguments of resource "some\_resource" "a" will also be hidden in the plan output:

Terraform will perform the following actions:

# some\_resource.a will be created

+ resource "some\_resource" "a" {

+ name = (sensitive)

+ address = (sensitive) }

Plan: 1 to add, 0 to change, 0 to destroy.

## Using Input Variable Values

Within the module that declared a variable, its value can be accessed from within [expressions](https://www.terraform.io/language/expressions) as var.<NAME>, where <NAME> matches the label given in the declaration block:

**Note:** Input variables are created by a variable block, but you reference them as attributes on an object named var.

resource "aws\_instance" "example" {

instance\_type = "t2.micro"

ami = var.image\_id}

The value assigned to a variable can only be accessed in expressions within the module where it was declared.

## Assigning Values to Root Module Variables

When variables are declared in the root module of your configuration, they can be set in a number of ways:

* [In a Terraform Cloud workspace](https://www.terraform.io/cloud-docs/workspaces/variables).
* Individually, with the -var command line option.
* In variable definitions (.tfvars) files, either specified on the command line or automatically loaded.
* As environment variables.

The following sections describe these options in more detail. This section does not apply to child modules, where values for input variables are instead assigned in the configuration of their parent module, as described in [Modules](https://www.terraform.io/language/modules).

### [»](https://www.terraform.io/language/values/variables#variables-on-the-command-line)Variables on the Command Line

To specify individual variables on the command line, use the -var option when running the terraform plan and terraform apply commands:

terraform apply -var="image\_id=ami-abc123"terraform apply -var='image\_id\_list=["ami-abc123","ami-def456"]' -var="instance\_type=t2.micro"terraform apply -var='image\_id\_map={"us-east-1":"ami-abc123","us-east-2":"ami-def456"}'

The above examples show appropriate syntax for Unix-style shells, such as on Linux or macOS. For more information on shell quoting, including additional examples for Windows Command Prompt, see [Input Variables on the Command Line](https://www.terraform.io/cli/commands/plan#input-variables-on-the-command-line).

You can use the -var option multiple times in a single command to set several different variables.

### [»](https://www.terraform.io/language/values/variables#variable-definitions-tfvars-files)Variable Definitions (.tfvars) Files

To set lots of variables, it is more convenient to specify their values in a variable definitions file (with a filename ending in either .tfvars or .tfvars.json) and then specify that file on the command line with -var-file:

terraform apply -var-file="testing.tfvars"

**Note:** This is how Terraform Cloud passes [workspace variables](https://www.terraform.io/cloud-docs/workspaces/variables) to Terraform.

A variable definitions file uses the same basic syntax as Terraform language files, but consists only of variable name assignments:

image\_id = "ami-abc123"availability\_zone\_names = [ "us-east-1a", "us-west-1c",]

Terraform also automatically loads a number of variable definitions files if they are present:

* Files named exactly terraform.tfvars or terraform.tfvars.json.
* Any files with names ending in .auto.tfvars or .auto.tfvars.json.

Files whose names end with .json are parsed instead as JSON objects, with the root object properties corresponding to variable names:

{

"image\_id": "ami-abc123",

"availability\_zone\_names": ["us-west-1a", "us-west-1c"]

}

### Environment Variables

As a fallback for the other ways of defining variables, Terraform searches the environment of its own process for environment variables named TF\_VAR\_ followed by the name of a declared variable.

This can be useful when running Terraform in automation, or when running a sequence of Terraform commands in succession with the same variables. For example, at a bash prompt on a Unix system:

$ export TF\_VAR\_image\_id=ami-abc123$ terraform plan

# Output Values

Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use. Output values are similar to return values in programming languages.

**Hands-on:** Try the [Output Data From Terraform](https://learn.hashicorp.com/tutorials/terraform/outputs) tutorial on HashiCorp Learn.

Output values have several uses:

* A child module can use outputs to expose a subset of its resource attributes to a parent module.
* A root module can use outputs to print certain values in the CLI output after running terraform apply.
* When using [remote state](https://www.terraform.io/language/state/remote), root module outputs can be accessed by other configurations via a [terraform\_remote\_state data source](https://www.terraform.io/language/state/remote-state-data).

Resource instances managed by Terraform each export attributes whose values can be used elsewhere in configuration. Output values are a way to expose some of that information to the user of your module.

**Note:** For brevity, output values are often referred to as just "outputs" when the meaning is clear from context.

## [»](https://www.terraform.io/language/values/outputs#declaring-an-output-value)Declaring an Output Value

Each output value exported by a module must be declared using an output block:

output "instance\_ip\_addr" { value = aws\_instance.server.private\_ip}

The label immediately after the output keyword is the name, which must be a valid [identifier](https://www.terraform.io/language/syntax/configuration#identifiers). In a root module, this name is displayed to the user; in a child module, it can be used to access the output's value.

The value argument takes an [expression](https://www.terraform.io/language/expressions) whose result is to be returned to the user. In this example, the expression refers to the private\_ip attribute exposed by an aws\_instance resource defined elsewhere in this module (not shown). Any valid expression is allowed as an output value.

**Note:** Outputs are only rendered when Terraform applies your plan. Running terraform plan will not render outputs.

## [»](https://www.terraform.io/language/values/outputs#accessing-child-module-outputs)Accessing Child Module Outputs

In a parent module, outputs of child modules are available in expressions as module.<MODULE NAME>.<OUTPUT NAME>. For example, if a child module named web\_server declared an output named instance\_ip\_addr, you could access that value as module.web\_server.instance\_ip\_addr.

## [»](https://www.terraform.io/language/values/outputs#optional-arguments)Optional Arguments

output blocks can optionally include description, sensitive, and depends\_on arguments, which are described in the following sections.

### [»](https://www.terraform.io/language/values/outputs#description-output-value-documentation)description — Output Value Documentation

Because the output values of a module are part of its user interface, you can briefly describe the purpose of each value using the optional description argument:

output "instance\_ip\_addr" { value = aws\_instance.server.private\_ip description = "The private IP address of the main server instance."}

The description should concisely explain the purpose of the output and what kind of value is expected. This description string might be included in documentation about the module, and so it should be written from the perspective of the user of the module rather than its maintainer. For commentary for module maintainers, use comments.

### [»](https://www.terraform.io/language/values/outputs#sensitive-suppressing-values-in-cli-output)sensitive — Suppressing Values in CLI Output

An output can be marked as containing sensitive material using the optional sensitive argument:

output "db\_password" { value = aws\_db\_instance.db.password description = "The password for logging in to the database." sensitive = true}

Terraform will hide values marked as sensitive in the messages from terraform plan and terraform apply. In the following scenario, our root module has an output declared as sensitive and a module call with a sensitive output, which we then use in a resource attribute.

# main.tf

module "foo" { source = "./mod"}

resource "test\_instance" "x" { some\_attribute = module.mod.a # resource attribute references a sensitive output}

output "out" { value = "xyz" sensitive = true}

# mod/main.tf, our module containing a sensitive output

output "a" { value = "secret" sensitive = true}

When we run a plan or apply, the sensitive value is redacted from output:

Terraform will perform the following actions:

# test\_instance.x will be created + resource "test\_instance" "x" { + some\_attribute = (sensitive) }

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs: + out = (sensitive value)

**Note:** In Terraform versions prior to Terraform 0.14, setting an output value in the root module as sensitive would prevent Terraform from showing its value in the list of outputs at the end of terraform apply. However, the value could still display in the CLI output for other reasons, like if the value is referenced in an expression for a resource argument.

Terraform will still record sensitive values in the [state](https://www.terraform.io/language/state), and so anyone who can access the state data will have access to the sensitive values in cleartext. For more information, see [Sensitive Data in State](https://www.terraform.io/language/state/sensitive-data).

### [»](https://www.terraform.io/language/values/outputs#depends_on-explicit-output-dependencies)depends\_on — Explicit Output Dependencies

Since output values are just a means for passing data out of a module, it is usually not necessary to worry about their relationships with other nodes in the dependency graph.

However, when a parent module accesses an output value exported by one of its child modules, the dependencies of that output value allow Terraform to correctly determine the dependencies between resources defined in different modules.

Just as with [resource dependencies](https://www.terraform.io/language/resources/behavior#resource-dependencies), Terraform analyzes the value expression for an output value and automatically determines a set of dependencies, but in less-common cases there are dependencies that cannot be recognized implicitly. In these rare cases, the depends\_on argument can be used to create additional explicit dependencies:

output "instance\_ip\_addr" { value = aws\_instance.server.private\_ip description = "The private IP address of the main server instance."

depends\_on = [ # Security group rule must be created before this IP address could # actually be used, otherwise the services will be unreachable. aws\_security\_group\_rule.local\_access, ]}

The depends\_on argument should be used only as a last resort. When using it, always include a comment explaining why it is being used, to help future maintainers understand the purpose of the additional dependency.