What we will cover:

1. [What is Terraform import command?](https://spacelift.io/blog/importing-exisiting-infrastructure-into-terraform#what-is-terraform-import-command)
2. [Why use the import command?](https://spacelift.io/blog/importing-exisiting-infrastructure-into-terraform#why-use-the-import-command)
3. [Terraform import use cases](https://spacelift.io/blog/importing-exisiting-infrastructure-into-terraform#terraform-import-use-cases)
4. [How to use Terraform import for resources?](https://spacelift.io/blog/importing-exisiting-infrastructure-into-terraform#how-to-use-terraform-import-for-resources)
5. [How to import Terraform modules?](https://spacelift.io/blog/importing-exisiting-infrastructure-into-terraform#how-to-import-terraform-modules)
6. [Terraform import example – importing a file](https://spacelift.io/blog/importing-exisiting-infrastructure-into-terraform#terraform-import-example--importing-a-file)
7. [Importing IAM roles using Terraform for\_each](https://spacelift.io/blog/importing-exisiting-infrastructure-into-terraform#importing-iam-roles-using-terraform-foreach)
8. [How to use the Terraform import block [Terraform 1.5 import]](https://spacelift.io/blog/importing-exisiting-infrastructure-into-terraform#how-to-use-the-terraform-import-block-terraform-15-import)
9. [How to import multiple resources in Terraform?](https://spacelift.io/blog/importing-exisiting-infrastructure-into-terraform#how-to-import-multiple-resources-in-terraform)
10. [Common questions about importing Terraform resources](https://spacelift.io/blog/importing-exisiting-infrastructure-into-terraform#common-questions-about-importing-terraform-resources)
11. [Best practices for Terraform import](https://spacelift.io/blog/importing-exisiting-infrastructure-into-terraform#best-practices-for-terraform-import)

**What is Terraform import command?**

Terraform import is a Terraform CLI command used to read real-world infrastructure and update its state so that future updates to the same set of infrastructure can be applied via Infrastructure as Code (IaC). It imports the pre-existing cloud resources into the Terraform state.

The import functionality helps update the state locally but does not automatically create the corresponding configuration. However, the Terraform team is working hard to improve this function in upcoming releases.

**Terraform import syntax and parameters**

The terraform import command takes the following arguments:

* ADDR – the address of the resource in terraform (e.g. aws\_instance.instance\_name)
* ID – the resource ID in the cloud provider/k8s/database service/vcs service/etc

An example terraform import would look like this:

terraform import aws\_iam\_role.role\_name my\_role

**Why use the import command?**

Terraform is a relatively new technology, and adopting it to manage an organization’s cloud resources might take some time and effort. The lack of human resources and the steep learning curve involved in using Terraform effectively causes teams to start using cloud infrastructure directly via their respective web consoles.

For that matter, any kind of IaC method (CloudFormation, Azure ARM templates, Pulumi, etc.) requires some training and real-time scenario handling experience. Things get especially complicated when dealing with concepts like states and remote backends. In a worst-case scenario, you can **lose the** [terraform.tfstate file](https://spacelift.io/blog/terraform-state). Luckily, you can use the import functionality to rebuild it.

**Terraform import use cases**

Terraform import is used to add existing infrastructure components into Terraform management. Here are a couple of use cases in which terraform import can be a lifesaver:

* **Bringing unmanaged resources under Terraform management** – your company may have started its infrastructure adventure by doing ClickOps or by using some custom scripts. Now, to bring everything under Terraform, terraform import will be the solution.
* **Migrating between Terraform states** – terraform import can help a lot if you want to split up your state file into multiple files, as managing a lot of things under a single file can be cumbersome.
* **Disaster recovery** – if your state file gets corrupted or you don’t have access to it anymore for whatever reason, you can use terraform import to rebuild it.
* **Adopting Terraform in phases** – with terraform import you can start by importing a couple of resources, helping you start small with Terraform.

**How to use Terraform import for resources?**

Now that we understand why we need to import cloud resources into the Terraform state, let’s begin by importing a simple resource – an EC2 instance in AWS.

We assume the Terraform installation and configuration of AWS credentials in AWS CLI is already done locally. We will not go into the details of that in this tutorial.

To import a simple resource into Terraform, follow the step-by-step guide below.

**1. Prepare the EC2 instance**

Assuming the Terraform installation and configuration of AWS credentials in AWS CLI is already done locally, begin by importing a simple resource — EC2 instance in AWS. For the sake of this tutorial, we will create an EC2 resource manually to be imported. This could be **optional if you already have a target resource to be imported**.

**Terraform: Create EC2 instance in the existing VPC**

Go ahead and provision an EC2 instance in your AWS account. Here are the example details of the EC2 instance thus created:

**Name: MyVM**  
**Instance ID: i-0b9be609418aa0609**  
**Type: t2.micro**  
**VPC ID: vpc-1827ff72**  
**…**

**2. Create main.tf and set provider configuration**

The aim of this step is to import this EC2 instance into our Terraform configuration. Create main.tf in your desired path and configure the [AWS provider](https://spacelift.io/blog/terraform-aws-provider). The file should look like below.

**Importing EC2 instance into Terraform configuration: Example**

// Provider configuration

terraform {

required\_providers {

aws = {

source = "hashicorp/aws"

version = "~> 3.0"

}

}

}

provider "aws" {

region = "eu-central-1"

}

Run terraform init to initialize the [Terraform modules](https://spacelift.io/blog/what-are-terraform-modules-and-how-do-they-work). Below is the output of a successful initialization.

Initializing the backend...

Initializing provider plugins...

- Finding hashicorp/aws versions matching "~> 3.0"...

- Installing hashicorp/aws v3.51.0...

- Installed hashicorp/aws v3.51.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider

selections it made above. Include this file in your version control repository

so that Terraform can guarantee to make the same selections by default when

you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see

any changes that are required for your infrastructure. All Terraform commands

should now work.

If you ever set or change modules or backend configuration for Terraform,

rerun this command to reinitialize your working directory. If you forget, other

commands will detect it and remind you to do so if necessary.

**3. Write config for the resource to be imported**

As discussed earlier, Terraform import does not generate the configuration files by itself. Thus, you need to create the corresponding configuration for the EC2 instance manually. This doesn’t need many arguments, as we will have to add or modify them when we import the EC2 instance into our state file.

However, if you don’t mind not seeing colorful output on CLI, you can begin adding all the arguments you know. But this is not a foolproof approach because normally, the infrastructure you may have to import will not have been created by you. So, it is best to **skip a few arguments** anyway.

In a moment, we will take a look at how to adjust our configuration to reflect the exact resource. For now, append the main.tf file with EC2 config. For example, we have used the below config. The only reason I have included ami and instance\_type attribute, is that they are the required arguments for aws\_instance resource block.

resource "aws\_instance" "myvm" {

ami = "unknown"

instance\_type = "unknown"

}

**4. Run the import command**

Think of it as if the cloud resource (EC2 instance) and its corresponding configuration were available in our files. All that’s left to do is to map the two into our state file. We do that by running the import command as follows.

terraform import aws\_instance.myvm <Instance ID>

A successful output should look like this:

aws\_instance.myvm: Importing from ID "i-0b9be609418aa0609"...

aws\_instance.myvm: Import prepared!

Prepared aws\_instance for import

aws\_instance.myvm: Refreshing state... [id=i-0b9be609418aa0609]

Import successful!

The resources that were imported are shown above. These resources are now in

your Terraform state and will henceforth be managed by Terraform.

The above command maps the aws\_instance.myvm configuration to the EC2 instance using the ID. By **mapping**, we mean that the state file now “knows” the existence of the EC2 instance with the given ID. The state file also contains information about each attribute of this EC2 instance, as it has fetched the same using the import command.

**5. Observe state files and plan output**

Please note that the directory now also contains the terraform.tfstate file. This file was generated after the import command was successfully run. Take a moment to review its contents.

Right now our configuration does not reflect all the attributes. Any attempt to plan/apply this configuration will **fail** because we have not adjusted the values of its attributes. To close the gap between configuration and state files, run the terraform plan and observe the output.

.

.

.

} -> (known after apply)

~ throughput = 0 -> (known after apply)

~ volume\_id = "vol-0fa93084426be508a" -> (known after apply)

~ volume\_size = 8 -> (known after apply)

~ volume\_type = "gp2" -> (known after apply)

}

- timeouts {}

}

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

───────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply"

now.

The plan indicates that it would attempt to replace the EC2 instance. But this goes completely against our purpose. We could do it anyway by simply not caring about the existing resources, and creating new resources using configuration.

The good news is that Terraform has noted the existence of an EC2 instance associated with its state.

**6. Improve config to avoid replacement**

At this point, it is important to understand that the terraform.tfstate file is a vital piece of reference for Terraform. All of its future operations are performed with consideration for this state file. You need to investigate the state file and update your configuration accordingly so there is a minimum difference between them.

The use of the word “minimum” is intentional here. Right now, you need to focus on not replacing the given EC2 instance but rather aligning the configuration so that the replacement can be avoided. Eventually, you would achieve a state of 0 difference.

Observe the plan output and find all those attributes that cause the replacement. The plan output will highlight the same. In our example, the only attribute that causes replacement is the AMI ID. Closing this gap should avoid the replacement of the EC2 instance.

Change the value of ami from “unknown” to what is highlighted in the plan output, and run terraform plan again. Notice the output.

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

~ update in-place

Terraform will perform the following actions:

# aws\_instance.myvm will be updated in-place

~ resource "aws\_instance" "myvm" {

id = "i-0b9be609418aa0609"

~ instance\_type = "t2.micro" -> "unknown"

~ tags = {

- "Name" = "MyVM" -> null

}

~ tags\_all = {

- "Name" = "MyVM"

} -> (known after apply)

# (27 unchanged attributes hidden)

# (6 unchanged blocks hidden)

}

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

This time, the plan does not indicate the replacement of the EC2 instance. If you get the same output, you are **successful in partially importing our cloud resource**. You are currently in a state of lowered risk—if we apply the configuration now, the resource will not be replaced, but a few attributes would change.

**7. Improve config to avoid changes**

If we want to achieve a state of 0 difference, you need to align your resource block even more. The plan output highlights the attribute changes using ~ sign. It also indicates the difference in the values. For example, it highlights the change in the instance\_type value from “t2.micro” to “unknown”.

In other words, if the value of instance\_type had been “t2.micro”, Terraform would NOT have asked for a change. Similarly, you can see there are changes to the tags highlighted as well. Let’s change the configuration accordingly so that we can close these gaps. The final aws\_instance resource block should look as follows:

resource "aws\_instance" "myvm" {

ami = "ami-00f22f6155d6d92c5"

instance\_type = "t2.micro"

tags = {

"Name": "MyVM"

}

}

Run terraform plan again, and observe the output.

aws\_instance.myvm: Refreshing state... [id=i-0b9be609418aa0609]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.

If you have the same output, congratulations, as you have **successfully imported** a cloud resource into your Terraform config. It is now possible to manage this configuration directly via Terraform, without any surprises.

**How to import Terraform modules?**

Importing an existing resource into a module instance is not very different from importing an existing resource into a Terraform resource. The only difference is related to the address of the resource in Terraform, as it will be prefixed with the “module.module\_instance\_name”.

This means that an example import for a Terraform module will look like this:

terraform import module.iam\_roles.aws\_iam\_role.role\_name my\_role

**Terraform Import AWS VPC module example**

In this section, you will learn how to import resources into the modules. For reference, we use the AWS VPC module (version 3.2.0) that creates 29 resources. Move the terraform.tfstate file to another location so that Terraform becomes unaware of the existing resources.

This poses the challenge of **importing the AWS VPC module into your configuration**. Modules wrap multiple AWS resources into a single package that can be reused in various projects. You should expect a large codebase and many resources to be part of the module.

The process of importing resources created using a module is very similar to what we have discussed thus far, with a little difference in running the command. Go through the .tf files included in the module’s source, and identify the resources to be imported.

For example, the AWS VPC module creates a VPC resource. To import this resource, run the command as below.

This command imports the target VPC resource in AWS to our module’s configuration.

terraform import module.vpc.aws\_vpc.this <VPC ID>

Running the plan command indicates how many resources will be created. In other words, how many resources are yet to be imported.

+ "Name" = "my-vpc"

+ "Terraform" = "true"

}

+ tags\_all = {

+ "Environment" = "dev"

+ "Name" = "my-vpc"

+ "Terraform" = "true"

}

+ vpc\_id = "vpc-0127895db175d45ff"

}

Plan: 28 to add, 0 to change, 0 to destroy.

───────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply"

now.

The above output indicates 28 more resources to be imported to (re)build a perfectly consistent terraform.tfstate file. One less as we have successfully imported the VPC resource. Referring to the plan output, identify the resources in AWS and repeat the process for import.

**Terraform import example - importing a file**

Suppose you want to manage the lifecycle of a file using Terraform. The problem is that you decided to do this after the file was created. To solve this issue, we will need to import it.

We have defined a file called **my\_file** with this content:

Hey, this is a file!

Now, let’s define the Terraform configuration for it:

resource "local\_file" "my\_file"{

content = "different content"

filename = "./my\_file"

}

Next, import the file:

terraform import local\_file.my\_file ./my\_file

This resource does not support import. Please contact the provider developer for additional information.

As you can see, running import doesn’t help us because this resource does not support import. What we can do, however, is read the content directly from our file and recreate it like this:

resource "local\_file" "my\_file"{

content = file("./my\_file")

filename = "./my\_file\_terraform"

}

terraform apply

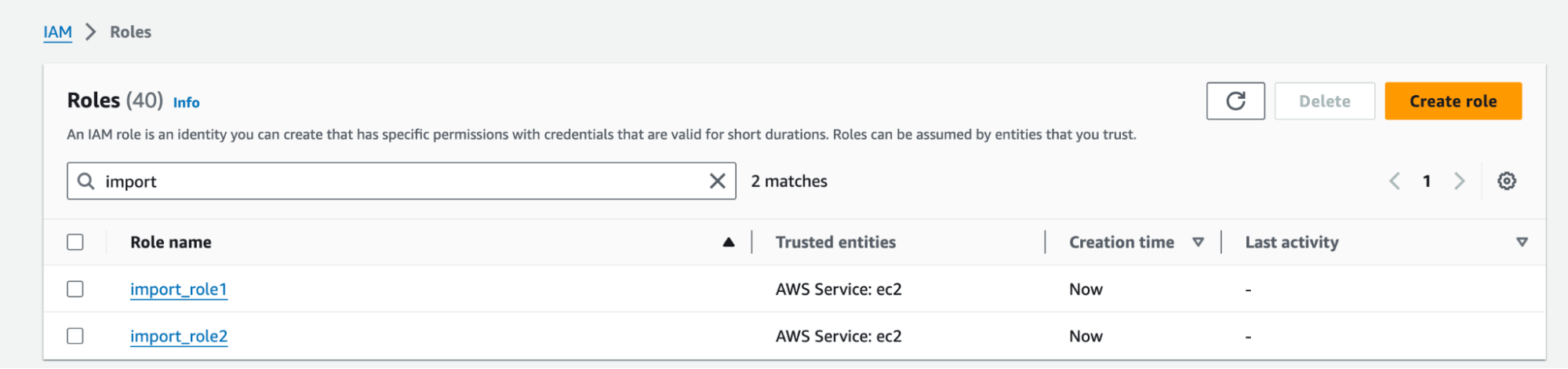
local\_file.my\_file: Creating...

local\_file.my\_file: Creation complete after 0s [id=6d2d3b974083c41cf0f19e9ecf86435092167de9]

The only issue with this is the fact that if you are using the file function, the old file must exist for reading the content. This can be overcome by adding the content directly in the content parameter.

**Importing IAM roles using Terraform for\_each**

Suppose you want to manage multiple [IAM roles with Terraform](https://spacelift.io/blog/terraform-iam-role) that already exist. In this example, I created two roles in my AWS account that I want to import using for\_each.



Now, let’s define the Terraform configuration for the roles:

provider "aws" {

region = "eu-west-1"

}

locals {

roles = ["import\_role1", "import\_role2"]

}

resource "aws\_iam\_role" "import\_roles" {

for\_each = toset(local.roles)

name = each.value

assume\_role\_policy = jsonencode(

{

Statement = [

{

Action = "sts:AssumeRole"

Principal = {

Service = "ec2.amazonaws.com"

}

Effect = "Allow"

},

]

Version = "2012-10-17"

})

description = "Allows EC2 instances to call AWS services on your behalf."

}

We have to ensure we use the same role policy we have used inside our AWS account and the same description.

Now that we have the configuration in place, let’s run a plan first to see that resources are not yet in the state:

terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

# aws\_iam\_role.import\_roles["import\_role1"] will be created

…

}

# aws\_iam\_role.import\_roles["import\_role2"] will be created

+ resource "aws\_iam\_role" "import\_roles" {

…

}

Plan: 2 to add, 0 to change, 0 to destroy.

Now we can import the resources:

terraform import "aws\_iam\_role.import\_roles[\"import\_role1\"]" import\_role1

aws\_iam\_role.import\_roles["import\_role1"]: Importing from ID "import\_role1"...

aws\_iam\_role.import\_roles["import\_role1"]: Import prepared!

Prepared aws\_iam\_role for import

aws\_iam\_role.import\_roles["import\_role1"]: Refreshing state... [id=import\_role1]

Import successful!

The resources that were imported are shown above. These resources are now in

your Terraform state and will henceforth be managed by Terraform.

terraform import "aws\_iam\_role.import\_roles[\"import\_role2\"]" import\_role2

aws\_iam\_role.import\_roles["import\_role2"]: Importing from ID "import\_role2"...

aws\_iam\_role.import\_roles["import\_role2"]: Import prepared!

Prepared aws\_iam\_role for import

aws\_iam\_role.import\_roles["import\_role2"]: Refreshing state... [id=import\_role2]

Import successful!

The resources that were imported are shown above. These resources are now in

your Terraform state and will henceforth be managed by Terraform.

Ensure you escape the quotation marks when you specify the role key while doing the import.

Now, let’s run another plan:

terraform plan

aws\_iam\_role.import\_roles["import\_role1"]: Refreshing state... [id=import\_role1]

aws\_iam\_role.import\_roles["import\_role2"]: Refreshing state... [id=import\_role2]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.

As you can see, the resources have been imported successfully.

**How to use the Terraform import block [Terraform 1.5 import]**

In Terraform 1.5, a new import mechanism is available. A new top-level import block can be defined in your code to allow import operations. As this Terraform import block is added in the code, importing will not be a state operation. From now on, as for every other resource, it becomes a plannable operation.

**Example: Terraform import S3 bucket**

Let’s use a concrete example to see the new import in action.

**1. Create the import block**

For this, we have created two s3 buckets manually, and we want to import them in our state:

* import-bucket-tf15
* import-bucket-tf15-2

provider "aws" {

region = "eu-west-1"

}

import {

# ID of the cloud resource

# Check provider documentation for importable resources and format

id = "import-bucket-tf15"

# Resource address

to = aws\_s3\_bucket.this

}

import {

# ID of the cloud resource

# Check provider documentation for importable resources and format

id = "import-bucket-tf15-2"

# Resource address

to = aws\_s3\_bucket.this2

}

The Terraform import block, as you can see above, takes two parameters:

* id → The id of the resource used in your cloud provider
* to → The resource address that will be used in Terraform

**2. Generate the configuration**

Next, if you want to generate the configuration automatically, you can run the following command:

terraform plan -generate-config-out=generated\_resources.tf

This will result in:

aws\_s3\_bucket.this: Preparing import... [id=import-bucket-tf15]

aws\_s3\_bucket.this2: Preparing import... [id=import-bucket-tf15-2]

aws\_s3\_bucket.this2: Refreshing state... [id=import-bucket-tf15-2]

aws\_s3\_bucket.this: Refreshing state... [id=import-bucket-tf15]

Terraform will perform the following actions:

# aws\_s3\_bucket.this will be imported

# (config will be generated)

resource "aws\_s3\_bucket" "this" {

ommited

}

}

# aws\_s3\_bucket.this2 will be imported

# (config will be generated)

resource "aws\_s3\_bucket" "this2" {

ommited

}

Plan: 2 to import, 0 to add, 0 to change, 0 to destroy.

╷

│ Warning: Config generation is experimental

│

│ Generating configuration during import is currently experimental, and the generated configuration format may change in future versions.

╵

─────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Terraform has generated configuration and written it to generated\_resources.tf. Please review the configuration and edit it as necessary before adding it to version

control.

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.

As you can see, config generation is still experimental at the moment of writing this article, so you should review it carefully before making any changes to your Terraform configuration.

**3. Review the output**

The resulting file will be similar to this:

# \_\_generated\_\_ by Terraform

# Please review these resources and move them into your main configuration files.

# \_\_generated\_\_ by Terraform from "import-bucket-tf15"

resource "aws\_s3\_bucket" "this" {

bucket = "import-bucket-tf15"

bucket\_prefix = null

force\_destroy = null

object\_lock\_enabled = false

tags = {}

tags\_all = {}

}

# \_\_generated\_\_ by Terraform from "import-bucket-tf15-2"

resource "aws\_s3\_bucket" "this2" {

bucket = "import-bucket-tf15-2"

bucket\_prefix = null

force\_destroy = null

object\_lock\_enabled = false

tags = {}

tags\_all = {}

}

**4. Run terraform plan**

Now, if you run the terraform plan, this is the output you are going to see:

Plan: 2 to import, 0 to add, 0 to change, 0 to destroy.

**5. Run terraform apply**

After you are done with all the changes, you can simply run the terraform apply to add these resources to the state:

Apply complete! Resources: 2 imported, 0 added, 0 changed, 0 destroyed.

**How to import multiple resources in Terraform?**

As you can see, the process of importing simple resources is pretty straightforward. The key here is to understand how Terraform state works. However, things can get quite tedious when importing complex deployments.

In the case of complex deployments, the team should clearly identify the resources they want to manage using Terraform. A diagram representing the entire landscape helps plan and get the parts of the deployment under Terraform’s management.

Planning output is key. If it is well-formatted, you will build a perfectly consistent state file. Prioritize and mitigate configurations that replace target resources, followed by configurations that cause changes. Import the resources as they are to satisfy the state file until there are no gaps. Once you achieve 0 gaps, any enhancement or code restructuring can be focused on and implemented.

**The difference between Terraform import and Terraform state mv**

With terraform import you are importing existing infrastructure resources inside of your state to manage them with Terraform. However, if you want to keep an object created with Terraform and manage it as a different resource (having a different name or moving it under a different module inside your configuration), you will need to use terraform state mv.

**Common questions about importing Terraform resources**

Now, we will answer some of your questions about this command.

**Does running terraform import change anything?**

Running terraform import won’t make any actual changes to your infrastructure; rather, it will just import that infrastructure resource to your Terraform state. Of course, if you then run an apply without defining the configuration for the imported resource, Terraform will have to destroy that resource because its configuration is not present.

**Does terraform import generate code?**

No, terraform import does not generate code. This falls under the engineer’s responsibility.

**What are the disadvantages of running Terraform import?**

Terraform import is a powerful tool that solves many issues related to manually created infrastructure. There are, however, some slight disadvantages when you are using it:

* You need to write the resource code after you import it.
* Without defining the exact configuration of the resource as the one you have done when creating it manually will result in [infrastructure drift](https://spacelift.io/blog/drift-detection).
* In older versions of Terraform, you had to import resources one by one (now this can be done using the import block).

**What is the difference between using Terraform data sources and importing a resource?**

Terraform data sources are used to list details about existing resources. You can’t make changes to an existing resource with a data source. By importing it to Terraform, you can handle its entire lifecycle.

**What is the difference between using terraform import and Terraformer?**

Terraform import will import only one resource that you will have to specify explicitly. Terraformer can import multiple types of resources with a single command and also generate code for these resources.

**Can only pre-existing objects be imported into Terraform?**

Yes, only pre-existing objects can be imported into Terraform.

**Best practices for Terraform import**

There are a couple of things you should consider before using terraform import:

* **Understand your existing infrastructure** and see how it would fit your Terraform configuration.
* **Create your Terraform configuration first** – before importing your resources, it will be really helpful to create your HCL code and ensure it actually reflects the settings and properties that have been set up for your IaC resources.
* **Use version control** – before doing imports, ensure your state file is managed remotely and also that you have versioning enabled for it. This also applies to your code configuration as well.
* **Run a plan after import** – it is very important to understand if the import worked properly and if the code configuration you’ve built really reflects the resource you have imported.