HashiCorp
(https://www.hashicorp.com)

Provisioners

JUMP TO SECTION

Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.

Provisioners are a Last Resort

Hands-on: To learn about more declarative ways to handle provisioning actions, try the Provision Infrastructure Deployed with Terraform (https://learn.hashicorp.com/collections/terraform/provision?utm_source=WEBSITE&utm_medium=WEB_IO&utm_offer=ARTICLE_PAGE&utm_content=DOCS) collection on HashiCorp Learn.

Terraform includes the concept of provisioners as a measure of pragmatism, knowing that there will always be certain behaviors that can't be directly represented in Terraform's declarative model.

However, they also add a considerable amount of complexity and uncertainty to Terraform usage. Firstly, Terraform cannot model the actions of provisioners as part of a plan because they can in principle take any action. Secondly, successful use of provisioners requires coordinating many more details than Terraform usage usually requires: direct network access to your servers, issuing Terraform credentials to log in, making sure that all of the necessary external software is installed, etc.

The following sections describe some situations which can be solved with provisioners in principle, but where better solutions are also available. We do not recommend using provisioners for any of the use-cases described in the following sections.

Even if your specific use-case is not described in the following sections, we still recommend attempting to solve it using other techniques first, and use provisioners only if there is no other option.

Passing data into virtual machines and other compute resources

When deploying virtual machines or other similar compute resources, we often need to pass in data about other related infrastructure that the software on that server will need to do its job.

The various provisioners that interact with remote servers over SSH or WinRM can potentially be used to pass such data by logging in to the server and providing it directly, but most cloud computing platforms provide mechanisms to pass data to instances at the time of their creation such that the data is immediately available on system boot. For example:

- Alibaba Cloud: `user_data` on `alicloud_instance` (https://registry.terraform.io/providers/aliyun/alicloud/latest/docs/resources/instance) or `alicloud_launch_template` (https://registry.terraform.io/providers/aliyun/alicloud/latest/docs/resources/launch_template).
- Amazon EC2: `user_data` or `user_data_base64` on `aws_instance` (https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance), `aws_launch_template` (https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/launch_template), and `aws_launch_configuration` (https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/launch_configuration).

- Amazon Lightsail: `user_data` on `aws_lightsail_instance` (https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/lightsail_instance).
- Microsoft Azure: `custom_data` on `azurerm_virtual_machine` (https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/virtual_machine) or `azurerm_virtual_machine_scale_set` (https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/virtual_machine_scale_set).
- Google Cloud Platform: `metadata` on `google_compute_instance` (https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/compute_instance) or `google_compute_instance_group` (https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/compute_instance_group).
- Oracle Cloud Infrastructure: `metadata` or `extended_metadata` on `oci_core_instance` (https://registry.terraform.io/providers/hashicorp/oci/latest/docs/resources/core_instance) or `oci_core_instance_configuration` (https://registry.terraform.io/providers/hashicorp/oci/latest/docs/resources/core_instance_configuration).
- VMware vSphere: Attach a virtual CDROM to `vsphere_virtual_machine` (https://registry.terraform.io/providers/hashicorp/vsphere/latest/docs/resources/virtual_machine) using the `cdrom` block, containing a file called `user-data.txt`.

Many official Linux distribution disk images include software called cloud-init (https://cloudinit.readthedocs.io/en/latest/) that can automatically process in various ways data passed via the means described above, allowing you to run arbitrary scripts and do basic system configuration immediately during the boot process and without the need to access the machine over SSH.

> **Hands-on:** Try the Provision Infrastructure with Cloud-Init (https://learn.hashicorp.com/tutorials/terraform/cloud-init?in=terraform/provision&utm_source=WEBSITE&utm_medium=WEB_IO&utm_offer=ARTICLE_PAGE&utm_content=DOCS) tutorial on HashiCorp Learn.

If you are building custom machine images, you can make use of the "user data" or "metadata" passed by the above means in whatever way makes sense to your application, by referring to your vendor's documentation on how to access the data at runtime.

This approach is *required* if you intend to use any mechanism in your cloud provider for automatically launching and destroying servers in a group, because in that case individual servers will launch unattended while Terraform is not around to provision them.

Even if you're deploying individual servers directly with Terraform, passing data this way will allow faster boot times and simplify deployment by avoiding the need for direct network access from Terraform to the new server and for remote access credentials to be provided.

Running configuration management software

As a convenience to users who are forced to use generic operating system distribution images, Terraform includes a number of specialized provisioners for launching specific configuration management products.

We strongly recommend not using these, and instead running system configuration steps during a custom image build process. For example, HashiCorp Packer (https://packer.io/) offers a similar complement of configuration management provisioners and can run their installation steps during a separate build process, before creating a system disk image that you can deploy many times.

> **Hands-on:** Try the Provision Infrastructure with Packer (https://learn.hashicorp.com/tutorials/terraform/packer?in=terraform/provision&utm_source=WEBSITE&utm_medium=WEB_IO&utm_offer=ARTICLE_PAGE&utm_content=DOCS) tutorial on HashiCorp Learn.

If you are using configuration management software that has a centralized server component, you will need to delay the *registration* step until the final system is booted from your custom image. To achieve that, use one of the mechanisms described above to pass the necessary information into each instance so that it can register itself with the configuration management server immediately on boot, without the need to accept commands from Terraform over SSH or WinRM.

First-class Terraform provider functionality may be available

It is technically possible to use the `local-exec` provisioner to run the CLI for your target system in order to create, update, or otherwise interact with remote objects in that system.

If you are trying to use a new feature of the remote system that isn't yet supported in its Terraform provider, that might be the only option. However, if there *is* provider support for the feature you intend to use, prefer to use that provider functionality rather than a provisioner so that Terraform can be fully aware of the object and properly manage ongoing changes to it.

Even if the functionality you need is not available in a provider today, we suggest to consider `local-exec` usage a temporary workaround and to also open an issue in the relevant provider's repository to discuss adding first-class provider support. Provider development teams often prioritize features based on interest, so opening an issue is a way to record your interest in the feature.

Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction. Provisioners can be used to bootstrap a resource, cleanup before destroy, run configuration management, etc.

How to use Provisioners

> **Note:** Provisioners should only be used as a last resort. For most common situations there are better alternatives. For more information, see the sections above.

If you are certain that provisioners are the best way to solve your problem after considering the advice in the sections above, you can add a `provisioner` block inside the `resource` block of a compute instance.

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo The server's IP address is ${self.private_ip}"
  }
}
```

The `local-exec` provisioner requires no other configuration, but most other provisioners must connect to the remote system using SSH or WinRM. You must include a `connection` block (/docs/language/resources/provisioners/connection.html) so that Terraform will know how to communicate with the server.

Terraform includes several built-in provisioners; use the navigation sidebar to view their documentation.

It's also possible to use third-party provisioners as plugins, by placing them in `%APPDATA%\terraform.d\plugins`, `~/.terraform.d/plugins`, or the same directory where the Terraform binary is installed. However, we do not recommend using any provisioners except the built-in `file`, `local-exec`, and `remote-exec` provisioners.

All provisioners support the `when` and `on_failure` meta-arguments, which are described below (see Destroy-Time Provisioners and Failure Behavior).

The `self` Object

Expressions in `provisioner` blocks cannot refer to their parent resource by name. Instead, they can use the special `self` object.

The `self` object represents the provisioner's parent resource, and has all of that resource's attributes. For example, use `self.public_ip` to reference an `aws_instance`'s `public_ip` attribute.

> **Technical note:** Resource references are restricted here because references create dependencies. Referring to a resource by name within its own block would create a dependency cycle.

Suppressing Provisioner Logs in CLI Output

The configuration for a `provisioner` block may use sensitive values, such as `sensitive` variables (/docs/language/values/variables.html#suppressing-values-in-cli-output) or `sensitive` output values (/docs/language/values/outputs.html#sensitive-suppressing-values-in-cli-output). In this case, all log output from the provisioner is automatically suppressed to prevent the sensitive values from being displayed.

Creation-Time Provisioners

By default, provisioners run when the resource they are defined within is created. Creation-time provisioners are only run during *creation*, not during updating or any other lifecycle. They are meant as a means to perform bootstrapping of a system.

If a creation-time provisioner fails, the resource is marked as **tainted**. A tainted resource will be planned for destruction and recreation upon the next `terraform apply`. Terraform does this because a failed provisioner can leave a resource in a semi-configured state. Because Terraform cannot reason about what the provisioner does, the only way to ensure proper creation of a resource is to recreate it. This is tainting.

You can change this behavior by setting the `on_failure` attribute, which is covered in detail below.

Destroy-Time Provisioners

If `when = destroy` is specified, the provisioner will run when the resource it is defined within is *destroyed*.

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    when    = destroy
    command = "echo 'Destroy-time provisioner'"
  }
}
```

Destroy provisioners are run before the resource is destroyed. If they fail, Terraform will error and rerun the provisioners again on the next `terraform apply`. Due to this behavior, care should be taken for destroy provisioners to be safe to run multiple times.

Destroy-time provisioners can only run if they remain in the configuration at the time a resource is destroyed. If a resource block with a destroy-time provisioner is removed entirely from the configuration, its provisioner configurations are removed along with it and thus the destroy provisioner won't run. To work around this, a multi-step process can be used to safely remove a resource with a destroy-time provisioner:

- Update the resource configuration to include `count = 0`.
- Apply the configuration to destroy any existing instances of the resource, including running the destroy provisioner.
- Remove the resource block entirely from configuration, along with its `provisioner` blocks.
- Apply again, at which point no further action should be taken since the resources were already destroyed.

This limitation may be addressed in future versions of Terraform. For now, destroy-time provisioners must be used sparingly and with care.

NOTE: A destroy-time provisioner within a resource that is tainted *will not* run. This includes resources that are marked tainted from a failed creation-time provisioner or tainted manually using `terraform taint`.

Multiple Provisioners

Multiple provisioners can be specified within a resource. Multiple provisioners are executed in the order they're defined in the configuration file.

You may also mix and match creation and destruction provisioners. Only the provisioners that are valid for a given operation will be run. Those valid provisioners will be run in the order they're defined in the configuration file.

Example of multiple provisioners:

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo first"
  }

  provisioner "local-exec" {
    command = "echo second"
  }
}
```

Failure Behavior

By default, provisioners that fail will also cause the Terraform apply itself to fail. The `on_failure` setting can be used to change this. The allowed values are:

- `continue` - Ignore the error and continue with creation or destruction.
- `fail` - Raise an error and stop applying (the default behavior). If this is a creation provisioner, taint the resource.

Example:

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command    = "echo The server's IP address is ${self.private_ip}"
    on_failure = continue
  }
}
```