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

Command: plan

JUMP TO SECTION

> **Hands-on:** Try the Terraform: Get Started (https://learn.hashicorp.com/collections/terraform/aws-get-started?utm_source=WEBSITE&utm_medium=WEB_IO&utm_offer=ARTICLE_PAGE&utm_content=DOCS) collection on HashiCorp Learn.

The `terraform plan` command creates an execution plan. By default, creating a plan consists of:

- Reading the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date.
- Comparing the current configuration to the prior state and noting any differences.
- Proposing 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.

If you are using Terraform directly in an interactive terminal and you expect to apply the changes Terraform proposes, you can alternatively run `terraform apply` (/docs/cli/commands/apply.html) directly. By default, the "apply" command automatically generates a new plan and prompts for you to approve it.

You can use the optional `-out=FILE` option to save the generated plan to a file on disk, which you can later execute by passing the file to `terraform apply` (/docs/cli/commands/apply.html) as an extra argument. This two-step workflow is primarily intended for when 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).

If you run `terraform plan` without the `-out=FILE` option then it will create a *speculative plan*, which is a description of a the effect of the plan but without any intent to actually apply it.

In teams that use a version control and code review workflow for making changes to real infrastructure, developers can use speculative plans to verify the effect of their changes before submitting them for code review. However, it's important to consider that other changes made to the target system in the meantime might cause the final effect of a configuration change to be different than what an earlier speculative plan indicated, so you should always re-check the final non-speculative plan before applying to make sure that it still matches your intent.

Usage

Usage: `terraform plan [options]`

The `plan` subcommand looks in the current working directory for the root module configuration.

Because the plan command is one of the main commands of Terraform, it has a variety of different options, described in the following sections. However, most of the time you should not need to set any of these options, because a Terraform configuration should typically be designed to work with no special additional options for routine work.

The remaining sections on this page describe the various options:

- **Planning Modes**: There are some special alternative planning modes that you can use for some special situations where your goal is not just to change the remote system to match your configuration.
- **Planning Options**: Alongside the special planning modes, there are also some options you can set in order to customize the planning process for unusual needs.
  - **Resource Targeting** is one particular special planning option that has some important caveats associated with it.
- **Other Options**: These change the behavior of the planning command itself, rather than customizing the content of the generated plan.

Planning Modes

The section above described Terraform's default planning behavior, which is intended for changing the remote system to match with changes you've made to your configuration.

Terraform has two alternative planning modes, each of which creates a plan with a different intended outcome:

- **Destroy mode:** creates a plan whose goal is to destroy all remote objects that currently exist, leaving an empty Terraform state. This can be useful for situations like transient development environments, where the managed objects cease to be useful once the development task is complete.

  Activate destroy mode using the `-destroy` command line option.

- **Refresh-only mode:** creates a plan whose goal is only to update the Terraform state and any root module output values to match changes made to remote objects outside of Terraform. This can be useful if you've intentionally changed one or more remote objects outside of the usual workflow (e.g. while responding to an incident) and you now need to reconcile Terraform's records with those changes.

  Activate refresh-only mode using the `-refresh-only` command line option.

In situations where we need to discuss the default planning mode that Terraform uses when none of the alternative modes are selected, we refer to it as "Normal mode". Because these alternative modes are for specialized situations only, some other Terraform documentation only discusses the normal planning mode.

The planning modes are all mutually-exclusive, so activating any non-default planning mode disables the "normal" planning mode, and you can't use more than one alternative mode at the same time.

> **Note:** In Terraform v0.15 and earlier, the `-destroy` option is supported only by the `terraform plan` command, and not by the `terraform apply` command. To create and apply a plan in destroy mode in earlier versions you must run `terraform destroy` (/docs/cli/commands/destroy.html).

> **Note:** The `-refresh-only` option is available only in Terraform v0.15.4 and later.

Planning Options

In addition to the planning *modes* described above, there are also several additional options that can modify details of the behavior of the planning step.

When you use `terraform apply` without passing it a saved plan file, it incorporates the `terraform plan` command functionality and so the planning options in this section, along with the planning mode selection options in the previous section, are also available with the same meanings on `terraform apply`.

- `-refresh=false` - Disables the default behavior of synchronizing the Terraform state with remote objects before checking for configuration changes.

  This option can potentially make the planning operation faster by reducing the number of remote API requests, but it comes at the expense of having Terraform not take into account any changes that might've happened outside of Terraform, and thus the resulting plan may not be complete or correct.

  This option is not available in the "refresh only" planning mode, because it would effectively disable the entirety of the planning operation in that case.

- `-replace=ADDRESS` - Instructs Terraform to plan to replace the single resource instance with the given address. If the given instance would normally have caused only an "update" action, or no action at all, then Terraform will choose a "replace" action instead.

  You can use this option if you have learned that a particular remote object has become degraded in some way. If you are using immutable infrastructure patterns then you may wish to respond to that by replacing the malfunctioning object with a new object that has the same configuration.

  This option is allowed only in the normal planning mode, so this option is incompatible with the `-destroy` option.

  The `-replace=...` option is available only from Terraform v0.15.2 onwards. For earlier versions, you can achieve a similar effect (with some caveats) using `terraform taint` (/docs/cli/commands/taint.html).

- `-target=ADDRESS` - Instructs Terraform to focus its planning efforts only on resource instances which match the given address and on any objects that those instances depend on.

  This command is for exceptional use only. See Resource Targeting below for more information.

- `-var 'NAME=VALUE'` - Sets a value for a single input variable (/docs/language/values/variables.html) declared in the root module of the configuration. Use this option multiple times to set more than one variable. For more information see Input Variables on the Command Line, below.

- `-var-file=FILENAME` - Sets values for potentially many input variables (/docs/language/values/variables.html) declared in the root module of the configuration, using definitions from a "tfvars" file (/docs/language/values/variables.html#variable-definitions-tfvars-files). Use this option multiple times to include values from more than one file.

There are several other ways to set values for input variables in the root module, aside from the `-var` and `-var-file` options. For more information, see Assigning Values to Root Module Variables (/docs/language/values/variables.html#assigning-values-to-root-module-variables).

Input Variables on the Command Line

You can use the `-var` command line option to specify values for input variables (/docs/language/values/variables.html) declared in your root module.

However, to do so will require writing a command line that is parsable both by your chosen command line shell *and* Terraform, which can be complicated for expressions involving lots of quotes and escape sequences. In most cases we recommend using the `-var-file` option instead, and write your actual values in a separate file so that Terraform can parse them directly, rather than interpreting the result of your shell's parsing.

To use `-var` on a Unix-style shell on a system like Linux or macOS we recommend writing the option argument in single quotes `'` to ensure the shell will interpret the value literally:

```
terraform plan -var 'name=value'
```

If your intended value also includes a single quote then you'll still need to escape that for correct interpretation by your shell, which also requires temporarily ending the quoted sequence so that the backslash escape character will be significant:

```
terraform plan -var 'name=va'\''lue'
```

When using Terraform on Windows, we recommend using the Windows Command Prompt (`cmd.exe`). When you pass a variable value to Terraform from the Windows Command Prompt, use double quotes `"` around the argument:

```
terraform plan -var "name=value"
```

If your intended value includes literal double quotes then you'll need to escape those with a backslash:

```
terraform plan -var "name=va\"lue"
```

PowerShell on Windows cannot correctly pass literal quotes to external programs, so we do not recommend using Terraform with PowerShell when you are on Windows. Use Windows Command Prompt instead.

The appropriate syntax for writing the variable value is different depending on the variable's type constraint (/docs/language/expressions/type-constraints.html). The primitive types `string`, `number`, and `bool` all expect a direct string value with no special punctuation except that required by your shell, as shown in the above examples. For all other type constraints, including list, map, and set types and the special `any` keyword, you must write a valid Terraform language expression representing the value, and write any necessary quoting or escape characters to ensure it will pass through your shell literally to Terraform. For example, for a `list(string)` type constraint:

```
# Unix-style shell
terraform plan -var 'name=["a", "b", "c"]'

# Windows Command Prompt (do not use PowerShell on Windows)
terraform plan -var "name=[\"a\", \"b\", \"c\"]"
```

Similar constraints apply when setting input variables using environment variables. For more information on the various methods for setting root module input variables, see Assigning Values to Root Module Variables (/docs/language/values/variables.html#assigning-values-to-root-module-variables).

Resource Targeting

> **Hands-on:** Try the Target resources (https://learn.hashicorp.com/tutorials/terraform/resource-targeting?in=terraform/state&utm_source=WEBSITE&utm_medium=WEB_IO&utm_offer=ARTICLE_PAGE&utm_content=DOCS) tutorial on HashiCorp Learn.

You can use the `-target` option to focus Terraform's attention on only a subset of resources. You can use resource address syntax (/docs/cli/state/resource-addressing.html) to specify the constraint. Terraform interprets the resource address as follows:

- If the given address identifies one specific resource instance, Terraform will select that instance alone. For resources with either `count` or `for_each` set, a resource instance address must include the instance index part, like `aws_instance.example[0]`.
- If the given address identifies a resource as a whole, Terraform will select all of the instances of that resource. For resources with either `count` or `for_each` set, this means selecting *all* instance indexes currently associated with that resource. For single-instance resources (without either `count` or `for_each`), the resource address and the resource instance address are identical, so this possibility does not apply.
- If the given address identifies an entire module instance, Terraform will select all instances of all resources that belong to that module instance and all of its child module instances.

Once Terraform has selected one or more resource instances that you've directly targeted, it will also then extend the selection to include all other objects that those selections depend on either directly or indirectly.

This targeting capability is provided for exceptional circumstances, such as recovering from mistakes or working around Terraform limitations. It is *not recommended* to use `-target` for routine operations, since this can lead to undetected configuration drift and confusion about how the true state of resources relates to configuration.

Instead of using `-target` as a means to operate on isolated portions of very large configurations, prefer instead to break large configurations into several smaller configurations that can each be independently applied. Data sources (/docs/language/data-sources/index.html) can be used to access information about resources created in other configurations, allowing a complex system architecture to be broken down into more manageable parts that can be updated independently.

Other Options

The `terraform plan` command also has some other options that are related to the input and output of the planning command, rather than customizing what sort of plan Terraform will create. These commands are not necessarily also available on `terraform apply`, unless otherwise stated in the documentation for that command.

The available options are:

- `-compact-warnings` - Shows any warning messages in a compact form which includes only the summary messages, unless the warnings are accompanied by at least one error and thus the warning text might be useful context for the errors.
- `-detailed-exitcode` - Returns a detailed exit code when the command exits. When provided, this argument changes the exit codes and their meanings to provide more granular information about what the resulting plan contains:
  - 0 = Succeeded with empty diff (no changes)
  - 1 = Error
  - 2 = Succeeded with non-empty diff (changes present)

- `-input=false` - Disables Terraform's default behavior of prompting for input for root module input variables that have not otherwise been assigned a value. This option is particular useful when running Terraform in non-interactive automation systems.

- `-json` - Enables the machine readable JSON UI (/docs/internals/machine-readable-ui.html) output. This implies `-input=false`, so the configuration must have no unassigned variable values to continue.

- `-lock=false` - Don't hold a state lock during the operation. This is dangerous if others might concurrently run commands against the same workspace.

- `-lock-timeout=DURATION` - Unless locking is disabled with `-lock=false`, instructs Terraform to retry acquiring a lock for a period of time before returning an error. The duration syntax is a number followed by a time unit letter, such as "3s" for three seconds.

- `-no-color` - Disables terminal formatting sequences in the output. Use this if you are running Terraform in a context where its output will be rendered by a system that cannot interpret terminal formatting.

- `-out=FILENAME` - Writes the generated plan to the given filename in an opaque file format that you can later pass to `terraform apply` to execute the planned changes, and to some other Terraform commands that can work with saved plan files.

  Terraform will allow any filename for the plan file, but a typical convention is to name it `tfplan`. **Do not** name the file with a suffix that Terraform recognizes as another file format; if you use a `.tf` suffix then Terraform will try to interpret the file as a configuration source file, which will then cause syntax errors for subsequent commands.

  The generated file is not in any standard format intended for consumption by other software, but the file *does* contain your full configuration, all of the values associated with planned changes, and all of the plan options including the input variables. If your plan includes any sort of sensitive data, even if obscured in Terraform's terminal output, it will be saved in cleartext in the plan file. You should therefore treat any saved plan files as potentially-sensitive artifacts.

- `-parallelism=n` - Limit the number of concurrent operation as Terraform walks the graph (/docs/internals/graph.html#walking-the-graph). Defaults to 10.

For configurations using the `local` backend (/docs/language/settings/backends/local.html) only, `terraform plan` accepts the legacy command line option `-state` (/docs/language/settings/backends/local.html#command-line-arguments).

Passing a Different Configuration Directory

Terraform v0.13 and earlier accepted an additional positional argument giving a directory path, in which case Terraform would use that directory as the root module instead of the current working directory.

That usage was deprecated in Terraform v0.14 and removed in Terraform v0.15. If your workflow relies on overriding the root module directory, use the `-chdir` global option (./#switching-working-directory-with-chdir) instead, which works across all commands and makes Terraform consistently look in the given directory for all files it would normaly read or write in the current working directory.

If your previous use of this legacy pattern was also relying on Terraform writing the `.terraform` subdirectory into the current working directory even though the root module directory was overridden, use the `TF_DATA_DIR` environment variable (/docs/cli/config/environment-variables.html#tf_data_dir) to direct Terraform to write the `.terraform` directory to a location other than the current working directory.