Best practices for using the Terraform AWS Provider

AWS Prescriptive Guidance

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Prescriptive Guidance: Best practices for using the Terraform AWS Provider

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Introduction **1**
Overview **2**
Objectives 2
Target audience 2
Security best practices **4**
Follow the principle of least privilege 4
Use IAM roles 5
Grant least privilege access by using IAM policies 5
Use temporary credentials for local runs 5
Use IAM roles for Amazon EC2 authentication 6
Use dynamic credentials for HCP Terraform workspaces 7
Use IAM roles in AWS CodeBuild 8
Run GitHub Actions remotely on HCP Terraform 8
Use GitHub Actions with OIDC and configure the AWS Credentials action 8
Use GitLab with OIDC and the AWS CLI 9
Use unique IAM users with legacy automation tools 9
Use the Jenkins AWS Credentials plugin 9
Continuously monitor, validate, and optimize least privilege 10
Continuously monitor access key usage 10
Continually validate IAM policies 10
Secure remote state storage 11
Enable encryption and access controls 11
Limit direct access to collaborative workflows 12
Keep secrets out of state 12
Continuously scan infrastructure and source code 14
Use AWS services for dynamic scanning 14
Detect drift 14
Perform static analysis 15
Ensure prompt remediation 15
Enforce policy as code 15
Guard against applying to the wrong account 17
Backend best practices **18**
Use Amazon S3 for remote storage 18
Enable remote state locking 19

Configure the state bucket ... 20
Enable versioning and automatic backups ... 20
Restore previous versions if needed ... 20
Use HCP Terraform ... 20
Facilitate team collaboration ... 21
Improve accountability by using AWS CloudTrail ... 21
Separate the backends for each environment ... 21
Reduce the scope of impact ... 22
Restrict production access ... 22
Simplify access controls ... 22
Avoid shared workspaces ... 22
Actively monitor remote state activity ... 22
Get alerts on suspicious unlocks ... 22
Monitor access attempts ... 23
Best practices for code base structure and organization ... 24
Implement a standard repository structure ... 24
Root module structure ... 26
Reusable module structure ... 27
Structure for modularity ... 28
Don't wrap single resources ... 28
Encapsulate logical relationships ... 29
Keep inheritance flat ... 29
Reference resources in outputs ... 29
Don't configure providers ... 29
Declare required providers ... 29
Configure providers for Regions and accounts ... 30
Follow naming conventions ... 31
Follow guidelines for resource naming ... 31
Follow guidelines for variable naming ... 32
Use attachment resources ... 32
Use default tags ... 33
Meet Terraform registry requirements ... 33
Use recommended module sources ... 34
Follow coding standards ... 35
Follow style guidelines ... 35
Configure pre-commit hooks ... 36

Refactor and import without re-creating resources ... 36
Best practices for testing ... 38
Validate inputs and assumptions in the configuration ... 38
Write unit tests with mock providers ... 38
Write integration tests ... 39
Best practices for AWS Provider version management ... 41
Commit the dependency lock file ... 41
Add automated version checks ... 41
Monitor new releases ... 42
Contribute to providers ... 42
Best practices for community modules ... 44
Discover community modules ... 44
Use variables for customization ... 44
Understand dependencies ... 44
Use trusted sources ... 45
Subscribe to notifications ... 45
Contribute to community modules ... 45
FAQ ... 47
Next steps ... 48
Resources ... 49
References ... 49
Tools ... 49
Document history ... 51

Best practices for using the Terraform AWS Provider

Michael Begin, Amazon Web Services

Managing infrastructure as code (IaC) with Terraform on AWS offers important benefits such as improved consistency, security, and agility. As your Terraform configuration grows in size and complexity, following best practices helps you avoid common pitfalls.

This guide is for organizations in situations such as these:

- A team that started with Terraform in one account now runs it across many accounts and Regions, and needs consistent credentials, state, and guardrails.
- A platform team publishes shared modules for application teams, and needs those modules to be safe to reuse and easy to upgrade.
- An organization is preparing for an audit or a regulatory review, and needs to show who changed infrastructure, how changes were reviewed, and that secrets aren't stored in plain text.
- Teams are upgrading to AWS Provider 6.x or a newer Terraform release, and want to adopt current features without breaking existing deployments.

This guide provides recommended best practices for using the Terraform AWS Provider from HashiCorp. It covers security controls, remote backends, code base structure, testing, version management, and community modules. The examples use Terraform 1.11 or later and AWS Provider 6.x. Each section goes into the specifics of applying these best practices:

- Security
- Backends
- Code base structure and organization
- Testing
- AWS Provider version management
- Community modules

Overview

Terraform providers are plugins that let Terraform interact with APIs. The Terraform AWS Provider is the official plugin for managing AWS infrastructure as code with Terraform. It handles authentication and translates Terraform configuration into AWS API calls that create, read, update, and delete AWS resources.

You configure the provider with a `provider` block in the root module. Each provider configuration works in one AWS account, the account of the credentials or role that it uses, and has a default AWS Region. For an example that manages resources in two Regions of one account, see **Configure providers for Regions and accounts** in Structure for modularity.

The practices in this guide apply the AWS Well-Architected Framework to Terraform. The security practices (temporary credentials, least privilege, encrypted state, secrets kept out of state) follow the Security Pillar. The backend, testing, and version management practices follow the Operational Excellence Pillar and the infrastructure as code capabilities in AWS Well-Architected DevOps Guidance: changes reviewed before they run, repeatable deployments, and detecting drift.

Objectives

This guide helps you gain operational knowledge on the Terraform AWS Provider and addresses the following business goals that you can achieve by following Infrastructure as Code (IaC) best practices around security, reliability, compliance, and developer productivity.

- Improve infrastructure code quality and consistency across Terraform projects.
- Accelerate developer onboarding and ability to contribute to infrastructure code.
- Increase business agility through faster infrastructure changes.
- Reduce errors and downtime related to infrastructure changes.
- Optimize infrastructure costs by following IaC best practices.
- Strengthen your overall security posture by implementing best practices.

Target audience

The target audience for this guide includes technical leads and managers who oversee teams that use Terraform for infrastructure as code (IaC) on AWS.

Other potential readers include infrastructure engineers, DevOps engineers, solutions architects, and developers who actively use Terraform to manage AWS infrastructure.

These best practices help you save time and realize the benefits of IaC in your role.

Security best practices

Properly managing authentication, access controls, and security is critical for secure use of the Terraform AWS Provider. This section outlines best practices around:

- IAM roles and permissions for least-privilege access
- Temporary credentials for people and pipelines, to help prevent unauthorized access to AWS accounts and resources
- Encryption and access controls for remote state
- Keeping secrets out of the Terraform state file
- Infrastructure and source code scanning to identify misconfigurations and drift
- Policy as code to enforce governance guardrails

Following these best practices helps strengthen your security posture when you use Terraform to manage AWS infrastructure.

Follow the principle of least privilege

Least privilege is a fundamental security principle that refers to granting only the minimum permissions required for a user, process, or system to perform its intended functions. It's a core concept in access control and a preventative measure against unauthorized access and potential data breaches.

The principle of least privilege is central to how Terraform authenticates and performs actions against AWS.

When you use Terraform to provision and manage AWS resources, it acts on behalf of an entity (user or role) that requires only the minimum permissions necessary to make API calls. Not following least privilege introduces the following security risks:

- If Terraform has excessive permissions beyond what's needed, an unintended misconfiguration could make undesired changes or deletions.
- Overly permissive access grants increase the scope of impact if Terraform state files or credentials are compromised.
- Failing to follow least privilege violates security best practices and regulatory compliance requirements for granting minimal required access.

Use IAM roles

Use IAM roles instead of IAM users wherever possible to enhance security with the Terraform AWS Provider. IAM roles provide temporary security credentials that expire automatically, which eliminates the need to manage long-term access keys. Roles also offer precise access controls through IAM policies.

Grant least privilege access by using IAM policies

Carefully construct IAM policies to ensure that roles and users have only the minimum set of permissions that are required for their workload. Start with an empty policy and iteratively add allowed services and actions. To accomplish this:

- Use IAM Access Analyzer policy generation to create a policy from the actions that a role called, as recorded in AWS CloudTrail.
- Manually review policies to remove any capabilities that aren't essential for the role's intended responsibility.
- Use IAM policy variables and tags to simplify permission management.

Well-constructed policies grant just enough access to accomplish the workload's responsibilities and nothing more. Define actions at the operation level, and allow calls only to required APIs on specific resources.

Following this best practice reduces the scope of impact and follows the fundamental security principles of separation of duties and least privilege access. Start strict and open access gradually as needed, instead of starting open and trying to restrict access later.

Use temporary credentials for local runs

When you run Terraform from a workstation, don't configure long-term access keys. Sign in with a named AWS Command Line Interface (AWS CLI) profile that supplies temporary credentials, and let the AWS Provider assume the role that Terraform runs as.

- If your organization uses AWS IAM Identity Center, create a profile with aws configure sso, and sign in with `aws sso login --profile <profile-name>`.
- If you sign in to the AWS Management Console as an IAM user, or through federation directly with IAM, sign in with aws login (AWS CLI version 2.32.0 or later). The identity needs the permissions in the `SignInLocalDevelopmentAccess` AWS managed policy. Terraform 1.15

and later (for the S3 backend) and AWS Provider 6.23.0 and later accept `aws login` credentials, and the CLI refreshes them for up to 12 hours.

Next, create an IAM role for Terraform that has the minimum permissions, and a trust policy that allows only the identities that run Terraform. Don't make the role its own principal. For IAM Identity Center users, set the principal to the account and add an `ArnLike` condition on `aws:PrincipalArn` that matches `arn:aws:iam::<account-id>:role/aws-reserved/sso.amazonaws.com/*AWSReservedSSO_<permission-set-name>_*`. IAM Identity Center adds a random suffix to the role names that it creates, and puts the roles under a path that can include a Region, so the condition needs both wildcards.

Then add an assume_role block with the role's ARN and a `session_name` to the provider configuration. AWS CloudTrail records the session name, so you can tell which runs came from which people or pipelines.

Choose between the two sign-in methods based on how your people already access AWS:

- **IAM Identity Center** gives you one sign-in for every account and central permission sets. It requires an organization instance of IAM Identity Center, and sessions last as long as the permission set's session duration.
- `aws login` needs no new identity infrastructure if your users already sign in to the console as IAM users or through IAM federation. Each user still needs an identity in each account that they work in, which is harder to manage at scale.

Select the profile when you run Terraform, for example `AWS_PROFILE=terraform-dev terraform plan`. Don't set `profile` in the `provider` block of shared code, because pipelines authenticate differently.

The key benefits of this best practice include temporary credentials that expire on their own, nothing to copy into scripts or files, fine-grained access controls on the role, and the ability to revoke access by changing the role or the permission set. The same configuration works for everyone on the team because no one's credentials are in it.

Use IAM roles for Amazon EC2 authentication

When you run Terraform from Amazon Elastic Compute Cloud (Amazon EC2) instances, avoid storing long-term credentials locally. Instead, use IAM roles and instance profiles to grant least-privilege permissions automatically.

First, create an IAM role with the minimum permissions and assign the role to the instance profile. The instance profile allows Amazon EC2 instances to inherit the permissions defined in the role. Then, launch instances by specifying that instance profile. The instance authenticates through the attached role.

Before you run any Terraform operations, run `aws sts get-caller-identity` on the instance. The output should show the ARN of the instance role's session (`arn:aws:sts::<account-id>:assumed-role/<role-name>/<instance-id>`), which confirms that the instance profile supplies the credentials.

This approach avoids hardcoding permanent AWS keys into scripts or Terraform configuration within the instance. The instance role and profile supply temporary credentials to Terraform transparently.

The key benefits of this best practice include improved security over long-term credentials, reduced credential management overhead, and consistency between development, test, and production environments. IAM role authentication simplifies Terraform runs from Amazon EC2 instances while enforcing least-privilege access.

Use dynamic credentials for HCP Terraform workspaces

HCP Terraform is a managed service provided by HashiCorp that helps teams use Terraform to provision and manage infrastructure across multiple projects and environments. When you run Terraform in HCP Terraform, use dynamic provider credentials instead of stored access keys. For each run, HCP Terraform issues a workload identity token, which is an OpenID Connect (OIDC) token. The AWS Provider exchanges this token for temporary credentials by calling `AssumeRoleWithWebIdentity` on an IAM role that trusts HCP Terraform.

To set this up, add an IAM OIDC identity provider for `app.terraform.io` to your account, and create a role whose trust policy checks the token's `aud` and `sub` claims. The `sub` claim names the organization, project, workspace, and run phase. You can therefore give plan runs a read-only role and apply runs a role that can make changes, by setting the `TFC_AWS_PLAN_ROLE_ARN` and `TFC_AWS_APPLY_ROLE_ARN` workspace variables. In the apply role's trust policy, require `app.terraform.io:aud` to equal `aws.workload.identity`, and require `app.terraform.io:sub` to equal the exact workspace and phase, for example `organization:example-org:project:networking:workspace:network-prod:run_phase:apply`.

Benefits include no long-term keys to store or rotate, credentials that last only for the run, and separate permissions for each workspace and run phase.

Use IAM roles in AWS CodeBuild

In AWS CodeBuild, run your builds by using an IAM role that's assigned to the CodeBuild project. This allows each build to automatically inherit temporary credentials from the role instead of using long-term keys.

Run GitHub Actions remotely on HCP Terraform

Configure GitHub Actions workflows to run Terraform remotely on HCP Terraform workspaces. Rely on dynamic credentials and remote state locking instead of GitHub secrets management.

Use GitHub Actions with OIDC and configure the AWS Credentials action

Use the OpenID Connect (OIDC) standard to federate GitHub Actions identity through IAM. Use the Configure AWS Credentials action to exchange the GitHub token for temporary AWS credentials without needing long-term access keys.

In the role's trust policy, check the token's `sub` claim so that only the intended repository, branch, or environment can assume the role. IAM rejects a GitHub OIDC trust policy that doesn't restrict `token.actions.githubusercontent.com:sub` (or `job_workflow_ref`). When `sub` names a GitHub environment, also restrict the branch with the `ref` claim, as IAM Access Analyzer recommends. The following condition block lets only jobs in the `production` environment of one repository, on the `main` branch, assume the role:

```
"Condition": {
  "StringEquals": {
    "token.actions.githubusercontent.com:aud": "sts.amazonaws.com",
    "token.actions.githubusercontent.com:sub": "repo:example-org/
infrastructure:environment:production",
    "token.actions.githubusercontent.com:ref": "refs/heads/main"
  }
}
```

The role's principal is the account's GitHub OIDC identity provider, `arn:aws:iam::<account-id>:oidc-provider/token.actions.githubusercontent.com`, and the action is `sts:AssumeRoleWithWebIdentity`.

Choose HCP Terraform dynamic credentials when HCP Terraform runs your plans and applies, and GitHub or GitLab OIDC when the CI runner runs Terraform. With HCP Terraform, the run phase in the token separates plan and apply permissions for you, but you depend on HashiCorp's service for every run. With CI runner OIDC, you keep the runs in your own pipeline, and you separate plan and apply permissions yourself, for example with a GitHub environment that requires approval before the apply job runs.

Use GitLab with OIDC and the AWS CLI

Use the OIDC standard to federate GitLab identities through IAM for temporary access. By relying on OIDC, you avoid managing long-term AWS access keys in GitLab. Credentials are issued just in time, which improves security. Restrict the role's trust policy on the `sub` claim, which names the project and branch (for example, `project_path:example-group/infrastructure:ref_type:branch:ref:main`), so that only that pipeline can assume the role with the permissions you grant it.

Use unique IAM users with legacy automation tools

If you have automation tools and scripts that lack native support for using IAM roles, you can create individual IAM users to grant programmatic access. The principle of least privilege still applies. Minimize policy permissions and rely on separate roles for each pipeline or script. If the workload runs outside AWS and can hold an X.509 certificate, use IAM Roles Anywhere to obtain temporary credentials instead of access keys. As you migrate to more modern tools or scripts, begin supporting roles natively and gradually transition to them.

> **Warning**
>
> IAM users have long-term credentials, which present a security risk. To help mitigate this risk, provide these users with only the permissions they require to perform the task, and remove these users when they are no longer needed.

Use the Jenkins AWS Credentials plugin

Use the AWS Credentials plugin in Jenkins to centrally configure and inject AWS credentials into builds dynamically. This avoids checking secrets into source control.

Continuously monitor, validate, and optimize least privilege

Over time, you might grant additional permissions that exceed the minimum required policies. Continuously analyze access to identify and remove any unnecessary entitlements.

Continuously monitor access key usage

If you cannot avoid using access keys, use IAM credential reports to find access keys that are older than 90 days, and revoke inactive keys across both user accounts and machine roles. Alert administrators to manually confirm the removal of keys for active employees and systems.

Monitoring key usage helps you optimize permissions because you can identify and remove unused entitlements. When you follow this best practice with access key rotation, it limits credential lifespan and enforces least privilege access.

AWS provides several services and features that you can use to find unused credentials and notify administrators:

- **IAM Access Analyzer:** Unused access findings report roles, access keys, passwords, and permissions that haven't been used for the number of days you choose, across an account or an organization.
- **AWS Security Hub CSPM:** IAM controls such as IAM.3 (access keys rotated every 90 days or less) and IAM.22 (credentials unused for 45 days removed) check every account and send findings that you can route to administrators through Amazon EventBridge and Amazon Simple Notification Service (Amazon SNS).
- **AWS Config:** Managed rules such as `access-keys-rotated` and `iam-user-unused-credentials-check` evaluate IAM users continuously, and can start remediation or send notifications to an Amazon SNS topic.

Continually validate IAM policies

Use IAM Access Analyzer to keep policies close to what each role needs:

- Validate policies before deployment. In your pipeline, run `aws accessanalyzer validate-policy` on each policy document that Terraform manages, and fail the build on security warnings and errors. Validation flags issues such as overly broad wildcards and GitHub OIDC trust policies without a repository condition.

- Run custom policy checks such as `check-no-new-access`, which fails when a changed policy grants more access than the deployed version, and `check-access-not-granted` for actions that must never be allowed.
- Review unused access findings, compare existing policies with policies generated from CloudTrail activity, and remove permissions that aren't used. Provide reports to role owners, and revoke unused permissions after a grace period.

Proactively and frequently revoking obsolete access minimizes the credentials that might be at risk during a security event. Automation provides sustainable, long-term credential hygiene and permissions optimization. Following this best practice limits the scope of impact by proactively enforcing least privilege across AWS identities and resources.

Secure remote state storage

Remote state storage refers to storing the Terraform state file remotely instead of locally on the machine where Terraform is running. The state file is crucial because it keeps track of the resources that are provisioned by Terraform and their metadata.

Unsecured remote state can result in loss of state data, inability to manage infrastructure, inadvertent resource deletion, and exposure of sensitive information stored in the state file. Securing remote state storage is therefore essential for production-grade Terraform usage.

Enable encryption and access controls

Amazon Simple Storage Service (Amazon S3) encrypts every new object with server-side encryption with Amazon S3 managed keys (SSE-S3) by default. For a state bucket, add the following controls:

- Encrypt state with an AWS Key Management Service (AWS KMS) customer managed key (SSE-KMS), so that reading state also requires permission to use the key. Set `kms_key_id` in the backend configuration, and enable an S3 Bucket Key to reduce AWS KMS requests.
- Add a bucket policy that denies requests that don't use TLS (`aws:SecureTransport` is `false`).
- Turn on S3 Block Public Access, and set S3 Object Ownership to **Bucket owner enforced** so that access control lists (ACLs) are disabled.
- Give the role that Terraform runs as access only to its own state object, its `.tflock` lock object, and the AWS KMS key.

For one state file with S3 native locking, the role needs these permissions and no others on the state bucket and key:

- `s3:ListBucket` on the bucket, limited with an `s3:prefix` condition to the state key and its lock file
- `s3:GetObject` and `s3:PutObject` on the state key
- `s3:GetObject`, `s3:PutObject`, and `s3:DeleteObject` on the lock file (`<key>.tflock`)
- `kms:Decrypt`, `kms:Encrypt`, and `kms:GenerateDataKey` on the AWS KMS key

A role that only plans needs the same lock file permissions, because `terraform plan` takes the state lock. It doesn't need `s3:PutObject` on the state key. For details, see the S3 backend permissions in the Terraform documentation.

For the backend configuration and the bucket policy, see Backend best practices.

Limit direct access to collaborative workflows

- Structure collaboration workflows in HCP Terraform or in a CI/CD pipeline within your Git repository to limit direct state access.
- Rely on pull requests, run approvals, policy checks, and notifications to coordinate changes.

Following these guidelines helps secure sensitive resource attributes and avoids conflicts with team members' changes. Encryption and strict access controls help reduce the scope of exposure, and collaboration workflows support consistent, coordinated change management.

Keep secrets out of state

Terraform stores resource attributes and data source results in state. A secret that a data source reads, or that you pass to a regular argument, is written to the state file in plain text. Marking a value as `sensitive` hides it from CLI output, but it's still stored in state. Keep secrets in AWS Secrets Manager, and use the following features so that secret values never reach the plan or state:

- **Ephemeral resources** (Terraform 1.10 and later) read a value during a run and never store it. For example, the ephemeral `aws_secretsmanager_secret_version` resource reads a secret. The `aws_secretsmanager_secret_version` data source reads the same secret but stores its value in state.

- **Write-only arguments** (Terraform 1.11 and later) accept ephemeral values and never store them, for example `password_wo` on `aws_db_instance`, `secret_string_wo` on `aws_secretsmanager_secret_version`, and `value_wo` on `aws_ssm_parameter`. Terraform sends a write-only value only when the matching version argument, such as `password_wo_version`, changes. Increment the version to rotate the value.
- **Ephemeral variables and outputs** (Terraform 1.10 and later) pass secrets between modules without storing them.
- **Service-managed passwords:** For Amazon RDS, `manage_master_user_password = true` makes Amazon RDS generate the master user password and store it in Secrets Manager, so Terraform never handles it. Amazon RDS doesn't support this for a DB instance that is the source of a read replica (except for RDS for SQL Server and RDS for Db2). For those, use a write-only password, as in the following example.

The following example reads a database password from Secrets Manager and passes it to Amazon RDS. The password doesn't appear in the plan, the state, or the logs. The other `aws_db_instance` arguments are omitted:

```
ephemeral "aws_secretsmanager_secret_version" "db" {
  secret_id = "orders/db-master"
}

resource "aws_db_instance" "primary" {
  password_wo         = ephemeral.aws_secretsmanager_secret_version.db.secret_string
  password_wo_version = 1
}
```

To create the secret from Terraform as well, generate the value with the ephemeral `random_password` resource and store it with `secret_string_wo` on `aws_secretsmanager_secret_version`. Then read it back with the ephemeral resource, so that the database and the secret remain in sync.

Choose service-managed passwords where the service supports them, because Terraform never handles the secret and the service rotates it. Choose write-only arguments when the service can't manage the secret, or when other systems need the same value. In that case, you are responsible for rotation: increment the version argument, and account for the brief window when the new value is in AWS Secrets Manager but not yet in the database.

For more information, see Securing sensitive data by using AWS Secrets Manager and HashiCorp Terraform and HashiCorp's documentation on ephemeral resources and write-only arguments.

Continuously scan infrastructure and source code

Proactively scan both infrastructure and source code continuously for risks such as exposed credentials or misconfigurations to harden your security posture. Address findings promptly by reconfiguring or patching resources.

Use AWS services for dynamic scanning

Use AWS native tools to monitor provisioned infrastructure across accounts and Regions. AWS Config records configuration changes and evaluates them against rules. AWS Security Hub CSPM runs security controls and aggregates findings. Amazon Inspector continuously scans Amazon Elastic Compute Cloud (Amazon EC2) instances, AWS Lambda functions, and container images for software vulnerabilities. Amazon GuardDuty detects threats in your accounts, workloads, and Amazon S3 buckets.

Detect drift

AWS Config shows changes to resources, but not whether they still match your Terraform configuration. To detect Terraform drift, run `terraform plan -detailed-exitcode` on a schedule in your pipeline. Exit code 0 means that the infrastructure matches the configuration, 1 means that the plan failed, and 2 means that there's drift or pending changes. Save the plan with `-out`, and on exit code 2, publish `terraform show` output for the saved plan so that someone can review the drift before deciding whether to apply or update the configuration.

Keep these points in mind:

- Don't use the exit code of `terraform plan -refresh-only -detailed-exitcode` as the drift signal. With Terraform 1.17.0 and AWS Provider 6.70.0, it returns 2 even when nothing has drifted. If you want a refresh-only check, count the `resource_drift` entries in the saved plan instead: `terraform show -json refresh.tfplan | jq '.resource_drift // [] | length'`.
- Don't add `-minimal-refresh` to drift jobs. That option refreshes only resources that already have proposed changes, so the plan misses changes made outside Terraform.
- In HCP Terraform, health assessments check workspaces for drift on a schedule.

Perform static analysis

Embed static analyzers directly into CI/CD pipelines to scan Terraform configuration code (HCL) and identify risks before deployment. This moves security checks to an earlier point in the development process (referred to as *shifting left*) and prevents misconfigured infrastructure. Options include Checkov, Trivy (`trivy config`), and TFLint with the AWS ruleset.

Ensure prompt remediation

For all scan findings, ensure prompt remediation by either updating Terraform configuration, applying patches, or reconfiguring resources manually as appropriate. Lower risk levels by addressing the root causes.

Using both infrastructure scanning and code scanning provides layered insight across Terraform configurations, the provisioned resources, and application code. This maximizes the coverage of risk and compliance through preventive, detective, and reactive controls while embedding security earlier into the software development lifecycle (SDLC).

Enforce policy as code

Use policy-as-code frameworks to provide governance guardrails for infrastructure provisioning with Terraform. HCP Terraform and Terraform Enterprise evaluate policies on each run, by using one of the following frameworks:

- Sentinel, HashiCorp's policy language
- Open Policy Agent (OPA), with policies written in Rego
- Terraform policy, with policies written in HCL. The Terraform CLI can also evaluate these policies, starting with Terraform 1.17. HashiCorp labels the framework as beta in its documentation.

For example, you can use policies to:

- Require tags on all resources.
- Restrict instance types to an approved list.
- Require encryption at rest, for example `storage_encrypted` on Amazon RDS instances.
- Require `allowed_account_ids` on every AWS provider configuration.
- Allow modules only from approved sources and versions.

- Prevent the destruction of production resources.

A Terraform policy file contains `provider_policy`, `resource_policy`, and `module_policy` blocks. Each has an `enforce` block with a condition and an error message, like a Terraform validation rule. A `resource_policy` for `aws_db_instance` can require `storage_encrypted = true`, and a `module_policy` can require a minimum module version before Terraform downloads the module. For the syntax, see the Terraform policy documentation.

Choose the framework based on where your runs happen and what your team already knows:

- **Sentinel** is a well-established option in HCP Terraform and Terraform Enterprise, with policy libraries from HashiCorp. Its language is specific to HashiCorp products, and it doesn't run outside them.
- **OPA** uses an open standard that you can also apply to Kubernetes and other tools, and open source tools can evaluate the JSON plan in any pipeline. Your team must learn Rego, and you build and maintain the policy integration in each pipeline yourself.
- **Terraform policy** uses HCL, so Terraform authors can write policies without learning another language, and it can check module sources before download. It is the most recently introduced framework, its CLI enforcement needs an HCP Terraform organization, and HashiCorp still labels it beta.

When you adopt Terraform policy, keep the following in mind:

- Write tests for each policy and run them with `tfpolicy test`. Authoring and testing don't need an HCP Terraform account.
- Label registry module policies with the full address, including `registry.terraform.io`. When Terraform evaluates module policies during `init`, a module's source includes the hostname, so a policy labeled `terraform-aws-modules/lambda/aws` matches no module. Use the full address in test mocks too, or the tests pass while the policy never runs.
- Prefer attributes that are known at plan time. A policy on a computed attribute, such as `tags_all` on a resource that has no tags, can't be evaluated until after the apply. Check `default_tags` on the provider configuration instead.
- In HCP Terraform, Terraform policy sets come from a version control repository or are uploaded through the API, which requires the Standard or Premium edition. A Terraform policy set that has no policies fails every run that it applies to. Attach a new policy set to a test workspace until its first version has uploaded, before you apply it to projects or the whole organization.

- To evaluate policies from the CLI, run `terraform init`, `plan`, or `apply` with `-policies=<directory>`. The CLI needs the separate `tfpolicy-plugin` binary, and a `cloud` block that selects a workspace in an HCP Terraform organization, which supplies the policy entitlement. With Terraform 1.17.0, `init -policies` stops when a module policy fails, but `plan -policies` and `apply -policies` report failed policies and still exit 0, and `apply` still makes the changes. In a pipeline, fail the job based on the policy results in the JSON output instead of the exit code. Run `terraform plan -json -policies=<directory>`, and fail when any `policy_result` message has the result `DenyResult` and an enforcement level other than `advisory`.

Policy checks complement IAM permissions. IAM limits what Terraform can do, and policies limit what your configuration is allowed to ask for.

Guard against applying to the wrong account

Set `allowed_account_ids` in every `provider "aws"` block. If the credentials in use belong to any other account, the provider stops the run with an error before it changes anything. This protects against a stale `AWS_PROFILE` or a pipeline that picks up the wrong role. The provider example in **Configure providers for Regions and accounts**, under Structure for modularity, shows the setting. If one configuration must work in several accounts, use `forbidden_account_ids` to exclude production accounts instead.

Backend best practices

Using a proper remote backend to store your state file is critical for enabling collaboration, ensuring state file integrity through locking, providing reliable backup and recovery, integrating with CI/CD workflows, and taking advantage of advanced security, governance, and management features offered by managed services such as HCP Terraform.

Terraform supports various backend types such as Kubernetes, HashiCorp Consul, and HTTP. However, this guide focuses on Amazon S3, which is a recommended backend solution for most AWS users.

As a fully managed object storage service that offers high durability and availability, Amazon S3 provides a secure, scalable, and cost-effective backend for managing Terraform state on AWS. The global footprint and resilience of Amazon S3 can be difficult to replicate when self-managing state storage. Additionally, being natively integrated with AWS access controls, encryption options, versioning capabilities, and other services makes Amazon S3 a convenient backend choice.

This guide doesn't provide backend guidance for other solutions such as Kubernetes or Consul because the primary target audience is AWS customers. For teams working primarily in the AWS Cloud, Amazon S3 is typically a recommended choice over Kubernetes or HashiCorp Consul clusters. The simplicity, resilience, and tight AWS integration of Amazon S3 state storage provides a strong foundation for most users who follow AWS best practices. Teams can take advantage of the durability, backup protections, and availability of AWS services to keep remote Terraform state highly resilient.

Following the backend recommendations in this section can help teams build more collaborative Terraform code bases while reducing the impact of errors or unauthorized modifications. By implementing a well-architected remote backend, teams can improve Terraform workflows.

Use Amazon S3 for remote storage

Storing Terraform state remotely in Amazon S3, with S3 native state locking, provides key benefits over local file storage. Remote state enables team collaboration, change tracking, backup protections, and locking to help prevent concurrent write conflicts.

The S3 Standard storage class (the default) is designed for 99.999999999% durability and 99.99% availability, which helps protect state from accidental loss compared to local storage or self-

managed solutions. For details, see Comparing the Amazon S3 storage classes in the Amazon S3 User Guide.

Enable remote state locking

State locking prevents concurrent write operations and reduces errors from simultaneous changes by multiple users. Use S3 native state locking by setting `use_lockfile = true`. It's generally available in Terraform 1.11 and later, so set `required_version = ">= 1.11"`. Locking through Amazon DynamoDB (the `dynamodb_table` argument) is deprecated, and HashiCorp plans to remove it in a future version.

While a run holds the lock, Terraform keeps a lock object named `<key>.tflock` next to the state file. Another run that tries to take the lock fails with `Error acquiring the state lock` and shows who holds it. To wait for a lock instead of failing, set `-lock-timeout`, for example `-lock-timeout=5m`.

Example backend configuration with S3 native locking and SSE-KMS encryption:

```
terraform {
  backend "s3" {
    bucket       = "DOC-EXAMPLE-BUCKET"
    key          = "network/prod/terraform.tfstate"
    region       = "us-west-2"
    encrypt      = true
    kms_key_id   = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
    use_lockfile = true
  }
}
```

With `encrypt = true`, `kms_key_id` selects the key that encrypts the state. Without `encrypt`, the backend ignores `kms_key_id`. If the role that runs Terraform doesn't have access to the state, add an `assume_role` block to the backend configuration with a role that does. The backend's role is separate from the provider's, so you can keep state access out of the role that changes resources.

To migrate from DynamoDB locking, set `use_lockfile = true` alongside `dynamodb_table` and run `terraform init -reconfigure`. Terraform then takes both locks. After every user and pipeline that uses the state runs Terraform 1.11 or later with the new configuration, remove `dynamodb_table`.

Configure the state bucket

Apply the controls from Enable encryption and access controls to the state bucket. Add a bucket policy statement that denies `s3:*` to every principal on the bucket and its objects when `aws:SecureTransport` is `false`. For an example, see Enforce encryption of data in transit in the Amazon S3 documentation.

Enable versioning and automatic backups

Enable S3 Versioning and backups by using AWS Backup on Amazon S3 backends to add additional data protection. Versioning preserves all previous versions of the state each time Terraform writes a change. It also lets you restore previous working state snapshots to roll back unwanted changes or recover from accidents.

Restore previous versions if needed

Versioned Amazon S3 state buckets let you revert changes by restoring a previous valid state snapshot. This helps protect against accidental changes and provides additional backup capabilities.

Use HCP Terraform

HCP Terraform provides a fully managed backend alternative to configuring your own state storage. HCP Terraform automatically handles the secure storage of state and encryption while unlocking additional features.

When you use HCP Terraform, state is stored remotely by default, which enables state sharing and locking across your organization. Detailed policy controls help you restrict state access and changes.

Additional capabilities include version control integrations, policy guardrails, workflow automation, variables management, and single sign-on integrations with SAML. You can enforce policy as code with Sentinel, OPA, or Terraform policy to implement governance controls.

Although HCP Terraform requires using a software as a service (SaaS) offering, for many teams the benefits around security, access controls, automated policy checks, and collaboration features make it a strong alternative to self-managing state storage in Amazon S3.

If your team uses cloud and SaaS tools such as GitHub and GitLab, you can integrate them with HCP Terraform to support your team workflows.

Facilitate team collaboration

Use remote backends to share state data across all the members of your Terraform team. This facilitates collaboration because it gives the entire team visibility into infrastructure changes. Shared backend protocols combined with state history transparency simplify internal change management. All infrastructure changes go through the established pipeline, which helps teams deliver infrastructure updates consistently and at scale.

Improve accountability by using AWS CloudTrail

Use CloudTrail to record the API calls made to the state bucket. Object-level operations such as `PutObject`, `GetObject`, and `DeleteObject` are data events. Trails don't log data events by default, and CloudTrail charges for them, so limit the trail to the state bucket with an advanced event selector. Select `eventCategory` equals `Data`, `resources.type` equals `AWS::S3::Object`, and `resources.ARN` starts with the state bucket's ARN followed by `/`. In Terraform, these are `field_selector` blocks in the `advanced_event_selector` block of `aws_cloudtrail`. Also set `enable_log_file_validation = true`.

CloudTrail logs show the IAM identity of the principal that made each API call for state change. You can match the user's identity to a machine account or to members of the team who interact with the backend storage.

Combine CloudTrail logs with S3 Versioning to tie infrastructure changes to the principal who applied them. By analyzing multiple revisions, you can attribute any updates to the machine account or responsible team member.

If an unintended or disruptive change occurs, state versioning provides rollback capabilities. CloudTrail traces the change to the user so you can discuss preventive improvements.

Also enforce IAM permissions to limit state bucket access. Overall, S3 Versioning and CloudTrail monitoring support auditing across infrastructure changes. Teams gain improved accountability, transparency, and audit capabilities for the Terraform state history.

Separate the backends for each environment

Use distinct Terraform backends for each application environment. Separate backends isolate state between development, test, and production. For the strongest isolation, put each environment in its own AWS account, in line with Organizing your AWS environment using multiple accounts, and keep each environment's state bucket in that account.

Reduce the scope of impact

Isolating state helps ensure that changes in lower environments don't impact production infrastructure. Accidents or experiments in development and test environments have limited impact.

Restrict production access

Restrict permissions for the production state backend to read-only access for most users. Limit who can modify the production infrastructure to the CI/CD pipeline and break glass roles.

Simplify access controls

Managing permissions at the backend level simplifies access control between environments. Using distinct S3 buckets for each application and environment means that broad read or write permissions can be granted on entire backend buckets.

Avoid shared workspaces

Although you can use Terraform workspaces to separate state between environments, distinct backends provide stronger isolation. If you have shared workspaces, accidents can still impact multiple environments.

Keeping environment backends fully isolated minimizes the impact of any single failure or unauthorized access event. Separate backends also align access controls to the environment's sensitivity level. For example, you can provide write protection for the production environment and broader access for development and test environments.

Actively monitor remote state activity

Monitor remote state activity continuously to detect anomalies early. Look for unexpected unlocks, unauthorized changes, or suspicious access attempts.

Get alerts on suspicious unlocks

With S3 native locking, the lock is the `<key>.tflock` object, and releasing a lock deletes that object. Your pipeline releases its lock at the end of every run, and `terraform force-unlock` or a manual delete releases another principal's lock. Most state changes should run through CI/CD pipelines, so alert when a principal other than the pipeline role deletes a lock object.

Create an Amazon EventBridge rule whose event pattern matches `source` `aws.s3`, `detail-type` `AWS API Call via CloudTrail`, `detail.eventName` `DeleteObject`, a `detail.requestParameters.key` that ends with `.tflock`, and a `detail.userIdentity.arn` that doesn't start with the pipeline role's assumed-role ARN. EventBridge receives these events only from a trail that logs data events for the state bucket, as described in the previous section.

Send the matches to an Amazon Simple Notification Service (Amazon SNS) topic or Amazon Simple Queue Service (Amazon SQS) queue that your team monitors.

Monitor access attempts

Authentication failures on state buckets might indicate unauthorized access attempts. Monitor for access from multiple unexpected accounts or unusual IP addresses, which may indicate unauthorized credential use.

Best practices for code base structure and organization

Proper code base structure and organization becomes increasingly important as Terraform usage grows across large teams and organizations. A structured, maintainable code base enables collaboration at scale. This section provides recommendations on Terraform modularity, naming conventions, documentation, and coding standards that support quality and consistency.

Guidance includes breaking configuration into reusable modules by environment and components, establishing naming conventions by using prefixes and suffixes, documenting modules and clearly explaining inputs and outputs, and applying consistent formatting rules by using automated style checks.

Additional best practices cover logically organizing modules and resources in a structured hierarchy, cataloging public and private modules in documentation, and abstracting unnecessary implementation details in modules to simplify usage.

Applying these guidelines helps teams scale Terraform usage consistently and reduces the long-term cost of maintaining a growing infrastructure code base.

Implement a standard repository structure

We recommend that you implement the following repository layout. Standardizing on these practices across modules improves discoverability, transparency, organization, and reliability while enabling reuse across many Terraform configurations.

- **Root directory:** In a root module, this is where you run `terraform plan` and `terraform apply`. In a reusable module, it's what callers point their `source` at. Split a large design into nested modules that match parts of the architecture, such as the network or the data tier, instead of one module for each resource type.
- **README.md:** Every module needs one, including nested modules that other teams use. Say what the module is for, what it creates, and what it expects from the caller. A diagram of the resources helps reviewers. terraform-docs can generate the inputs and outputs tables from the code, so they don't drift from it.
- **main.tf:** Resources and calls to nested modules. A small module needs no other resource files.
- **variables.tf** and **outputs.tf:** Inputs and outputs, each with a short `description`, because the description becomes the module's documentation. For details, see variable blocks and output blocks in the HashiCorp documentation.

- Give every variable a `type`.
- A variable with a `default` is optional. The default must be a literal value; it can't refer to other objects in the configuration. To make a variable required, leave out the default, and consider setting `nullable = false`.
- Set a default when one value suits every environment, such as `log_retention_days = 30`. Leave out the default when the right value depends on the environment, such as `vpc_id` or `account_id`, so that each caller has to set it.
- Use an empty string or an empty list as a default only when the AWS API accepts an empty value for that setting.
- Expose a variable only when callers have a concrete reason to change the value. You can add an optional variable later without breaking callers, but removing a variable breaks every caller that sets it. For a literal that you repeat inside the module, use a local value instead of a variable.
- Build outputs from resource attributes, not from input variables. An output such as `value = aws_s3_bucket.this.id` makes the caller's resources depend on the bucket, so Terraform creates the bucket first. An output that returns `var.bucket_name` carries no dependency, so a caller's resources can be created before the bucket exists. For more information, see Create resource dependencies.
- **locals.tf:** Local values give a name to an expression that the module uses more than once, such as a common name prefix built from `var.environment`. Unlike variables, callers can't set them.
- **providers.tf:** The terraform block and provider blocks. Only root modules declare `provider` blocks. If you're using HCP Terraform, also add an empty cloud block, and let your CI/CD pipeline set the organization, workspace, and token through environment variables and environment variable credentials.
- **versions.tf:** The `required_version` setting and the required_providers block. Every module declares the providers that it uses.
- **.terraform.lock.hcl:** The dependency lock file records the exact provider versions and checksums that `terraform init` selected. Commit it in root modules. Terraform reads the lock file only from the root module, so reusable modules don't need one. For more information, see Version management.
- **Data sources:** Declare each data source in the file that uses its result, for example the `aws_ami` lookup next to the `aws_instance` that launches from it. Move data sources to a `data.tf` file only when there are enough of them to make the main file hard to read.
- **.tfvars files:** Keep non-sensitive inputs for a root module in `.tfvars` files: values that every environment shares in `terraform.tfvars` at the repository root, and the rest in `envs/`

`<environment>/terraform.tfvars`. Don't put secrets in `.tfvars` files that you commit. Pass them from your pipeline's secret store as ephemeral variables.

- **tests/:** Put Terraform test files (`*.tftest.hcl`) in a `tests/` directory. For more information, see Best practices for testing.
- **modules/:** Put nested modules in `modules/<name>/`. A nested module with a `README.md` is one that other teams can call, and one without is internal to the repository. Call nested modules with relative paths, such as `./modules/network`, so that Terraform loads them from the same repository instead of downloading them. Have the root module wire nested modules together, instead of having nested modules call each other several levels deep.
- **examples/:** Put usage examples for a reusable module in `examples/<name>/`, each with a short README. People copy examples into their own repositories, so set each example's module `source` to the registry or Git address that a caller would use, not to a relative path.
- **Splitting files:** Keep resources in `main.tf` until the file gets hard to review, and then split by purpose, such as `iam.tf` for the module's roles and policies. Avoid a file for every AWS service by default, because a reader then has to open several files to follow one feature.
- **scripts/:** Terraform doesn't track what a script creates or changes, so prefer resources and data sources. When no resource can do what you need, put the script that Terraform runs, for example through the `external` data source or a `local-exec` provisioner, in `scripts/`.
- **helpers/:** Put scripts that people run by hand and Terraform never calls, such as a one-time bootstrap or a cleanup script, in `helpers/`. Document each one in `README.md` with an example command.

Root module structure

Terraform always runs in the context of a single root module. A complete Terraform configuration consists of a root module and the tree of child modules (which includes the modules that the root module calls, any modules that those modules call, and so on).

Terraform root module layout basic example:

```
.
### .terraform.lock.hcl
### data.tf
### envs
#   ### dev
#   #   ### terraform.tfvars
#   ### prod
```

```
#   #   ### terraform.tfvars
#   ### test
#       ### terraform.tfvars
### locals.tf
### main.tf
### outputs.tf
### providers.tf
### README.md
### terraform.tfvars
### tests
#   ### main.tftest.hcl
### variables.tf
### versions.tf
```

Reusable module structure

Reusable modules follow the same concepts as root modules. To define a module, create a new directory for it and place the `.tf` files inside, just as you would define a root module. Terraform can load modules either from local relative paths or from remote repositories. If you expect a module to be reused by many configurations, place it in its own version control repository. Keep the module tree relatively flat to make it easier to reuse modules in different combinations.

Terraform reusable module layout basic example:

```
.
### data.tf
### examples
#   ### multi-az-new-vpc
#   #   ### data.tf
#   #   ### locals.tf
#   #   ### main.tf
#   #   ### outputs.tf
#   #   ### providers.tf
#   #   ### README.md
#   #   ### terraform.tfvars
#   #   ### variables.tf
#   #   ### versions.tf
#   #   ### vpc.tf
#   ### single-az-existing-vpc
#       ### data.tf
#       ### locals.tf
#       ### main.tf
```

```
#       ### outputs.tf
#       ### providers.tf
#       ### README.md
#       ### terraform.tfvars
#       ### variables.tf
#       ### versions.tf
### iam.tf
### locals.tf
### main.tf
### outputs.tf
### README.md
### tests
#   ### integration.tftest.hcl
#   ### unit.tftest.hcl
### variables.tf
### versions.tf
```

Structure for modularity

A module is worth its cost when it gives a name to something that your architecture talks about, such as a VPC network baseline, a data tier, or an application service. Each module adds an interface to maintain, versions to manage, and one more place to look during an incident. Split a configuration into reusable modules when several teams or environments need the same set of resources, not for every group of resources.

Size each module around one capability. A module for each resource type leaves callers to wire everything together, and a module that builds a whole application stack is hard to reuse and slow to plan.

Don't wrap single resources

A module that contains one resource, and passes every argument of that resource through as a variable, adds a layer without adding a concept. A useful test is the module's name: if the best name you can find is the resource type, such as `s3-bucket`, use the resource directly, or design a module around what the bucket is for, such as `log-archive`.

Encapsulate logical relationships

Group sets of related resources such as networking foundations, data tiers, security controls, and applications. A reusable module should encapsulate infrastructure pieces that work together to enable a capability.

Keep inheritance flat

When you nest modules in subdirectories, avoid going more than one or two levels deep. Deeply nested inheritance structures complicate configurations and troubleshooting. Modules should build on other modules, not pass through them as intermediaries.

By focusing modules on logical resource groupings that represent architecture patterns, teams can configure reliable infrastructure foundations. Aim for the right level of abstraction: avoid both excessive complexity and oversimplification.

Reference resources in outputs

For every resource that a reusable module defines, include at least one output that references the resource. Variables and outputs let you infer dependencies between modules and resources. Without any outputs, users cannot properly order your module in relation to their Terraform configurations.

Well-structured modules that provide environment consistency, purpose-driven groupings, and exported resource references enable organization-wide Terraform collaboration at scale. Teams can assemble infrastructure from reusable building blocks.

Don't configure providers

Although shared modules inherit providers from calling modules, modules should not configure provider settings themselves. Avoid specifying provider configuration blocks in modules. This configuration should only be declared once globally.

Declare required providers

Although provider configurations are shared between modules, shared modules must also declare their own provider requirements. This practice enables Terraform to ensure that there is a single version of the provider that's compatible with all modules in the configuration, and to specify the

source address that serves as the global (module-agnostic) identifier for the provider. However, module-specific provider requirements don't specify any of the configuration settings that determine what remote endpoints the provider will access, such as an AWS Region.

By declaring version requirements and avoiding hardcoded provider configuration, modules provide portability and reusability across Terraform configurations using shared providers.

Declare version requirements differently in reusable modules and in root modules:

- **Reusable modules** declare the oldest Terraform and provider versions that they work with, by using >=. A lower bound lets the root module choose any newer version.
- **Root modules** also set an upper bound, usually with the *pessimistic constraint operator* (~>), so that a new major version isn't installed without review. The dependency lock file then pins the exact version.

For example, both set `required_version = ">= 1.11"` in the `terraform` block. In `required_providers`, a reusable module declares the AWS Provider with `version = ">= 6.0"`, and a root module declares it with `version = "~> 6.0"`.

`~> 6.0` allows any 6.x version, such as `6.70.0`, but not `7.0.0`. To allow only patch releases of one minor version, use three components, such as `~> 6.70.0`. For more information, see Version constraints and Provider requirements in the HashiCorp documentation.

Configure providers for Regions and accounts

Configure providers in the root module. Each provider configuration works in one AWS account, the account of the credentials or role that it uses, and has a default AWS Region. In AWS Provider 6.x, most resources also accept a top-level `region` argument, so one provider configuration can manage resources in several Regions of the same account. To manage resources in another account, add a provider configuration with an alias that assumes a role in that account.

The following example sets the default Region and limits the configuration to one account. It then creates an Amazon Relational Database Service (Amazon RDS) read replica in a second Region of the same account, from a primary DB instance that is defined elsewhere in the configuration:

```
provider "aws" {
  region              = "us-west-2"
  allowed_account_ids = ["111122223333"]
}
```

```
resource "aws_db_instance" "read_replica" {
  region              = "us-east-1"
  replicate_source_db = aws_db_instance.primary.arn
  instance_class      = "db.t4g.medium"
}
```

In this example:

- The `provider` block manages resources in account 111122223333, with `us-west-2` as the default Region. `allowed_account_ids` stops any run whose credentials belong to a different account, before Terraform changes anything.
- The replica sets `region = "us-east-1"`, so the same provider configuration creates it in that Region. A cross-Region replica refers to its source by Amazon Resource Name (ARN) in `replicate_source_db`. The source needs automated backups (`backup_retention_period` greater than 0), and an encrypted replica needs a key in its own Region in `kms_key_id`. Changing a resource's `region` value replaces the resource.
- To manage resources in another account, such as a DNS record in a shared networking account, add a second `provider` block with an `alias` and an assume_role block, and set `provider = aws.<alias>` on those resources.

Amazon RDS creates a read replica in the same account as its source DB instance. To copy data to a database in another account, use AWS Database Migration Service (AWS DMS) or the database engine's own replication.

Follow naming conventions

Clear, descriptive names simplify your understanding of relationships between resources in the module and the purpose of configuration values. Consistency with style guidelines enhances readability for both module users and maintainers.

Follow guidelines for resource naming

- Use *snake_case* (where lowercase terms are separated by underscores) for all resource names to match Terraform style standards. This practice ensures consistency with the naming convention for resource types, data source types, and other predefined values. This convention doesn't apply to name arguments.

- To simplify references to a resource that is the only one of its type (for example, a single load balancer for an entire module), name the resource `main` or `this` for clarity.
- Use meaningful names that describe the purpose and context of the resource, and that help differentiate between similar resources (for example, `primary` for the main database and `read_replica` for a read replica of the database).
- Use singular, not plural names.
- Don't repeat the resource type in the resource name.

Follow guidelines for variable naming

- Add units to the names of inputs, local variables, and outputs that represent numeric values such as disk size or RAM size (for example, `ram_size_gb` for RAM size in gigabytes). This practice makes the expected input unit clear for configuration maintainers.
- Use binary units such as MiB and GiB for storage sizes, and decimal units such as MB or GB for other metrics.
- Give Boolean variables positive names such as `enable_external_access`.

Use attachment resources

Some resources can manage related objects in two ways: as nested blocks inside the parent resource, or as separate resources that attach to it. Prefer the separate resources. When the same objects are managed in both places, or by two resource types, each one overwrites the other's changes, and Terraform shows a difference on every plan. Separate resources also give each object its own ID and tags.

For security groups, use one `aws_vpc_security_group_ingress_rule` or `aws_vpc_security_group_egress_rule` resource for each rule. The AWS Provider documentation recommends these resources over the `ingress` and `egress` blocks of `aws_security_group`, and over the `aws_security_group_rule` resource. Each rule resource takes one CIDR block, prefix list, or referenced security group.

For example, a web server security group that allows HTTPS from the load balancer's security group becomes an `aws_security_group` with no `ingress` or `egress` blocks, plus an `aws_vpc_security_group_ingress_rule` with `security_group_id`, `referenced_security_group_id`, `ip_protocol = "tcp"`, and `from_port` and `to_port` set to `443`.

Scope egress rules to the destinations that the workload needs, such as the security group of the VPC endpoints, instead of `0.0.0.0/0`. Scanners such as Trivy rate unrestricted egress as critical.

Other resources follow the same rule. For example, use `aws_iam_role_policy_attachment` instead of `managed_policy_arns` on `aws_iam_role`, and `aws_s3_bucket_versioning` and the other `aws_s3_bucket_*` resources instead of the deprecated arguments of `aws_s3_bucket`.

Use default tags

Assign tags to all resources that can accept tags. Set the tags that every resource in a configuration shares in the provider's `default_tags` block. The provider adds them to every resource that it creates, and a resource's own `tags` argument adds to them or overrides them. The `tags_all` attribute of each resource shows the combined result. For example, set `Environment`, `Owner`, and `CostCenter` in `default_tags`, and set only `Name` on each resource.

`aws_autoscaling_group` doesn't receive default tags. Read them with the aws_default_tags data source, and pass them to the group's `tag` blocks with a `dynamic "tag"` block that sets `propagate_at_launch = true`, so that they're also applied to the instances that it launches.

Add the following tags to all resources that a Terraform module creates:

- **Name**: Human-readable resource name
- **AppId**: The ID for the application that uses the resource
- **AppRole**: The resource's technical function; for example, "webserver" or "database"
- **AppPurpose**: The resource's business purpose; for example, "frontend ui" or "payment processor"
- **Environment**: The software environment, such as dev, test, or prod
- **Project**: The projects that use the resource
- **CostCenter**: Who to bill for resource usage

Meet Terraform registry requirements

Structure each reusable module so that you can publish it to a registry, even if you don't plan to publish it soon. You can then publish it later without changing its configuration or repository layout.

Requirements for the public Terraform Registry:

- **Repository:** A public GitHub repository named `terraform-aws-<NAME>`, where `<NAME>` reflects the type of infrastructure the module manages. The `<NAME>` segment can contain additional hyphens (for example, `terraform-aws-iam-terraform-roles`).
- **Standard module structure:** The module files are at the root of the repository, following the standard structure. The registry uses this structure to generate documentation, track resource usage, and more.
- **Release tags:** At least one tag that uses semantic versioning, optionally prefixed with `v` (for example, `v1.1.0` or `1.1.0`). The registry ignores tags that don't look like version numbers.

Requirements for an HCP Terraform private registry:

- **Repository:** The repository is in one of your organization's configured version control system (VCS) providers, and the HCP Terraform VCS user account has administrator access to it. The registry needs administrator access to create the webhooks that import new module versions.
- **Release tags:** Semantic version tags, as for the public registry, unless you publish with branch-based publishing.
- **Access:** The registry handles downloads and controls access with HCP Terraform API tokens, so consumers don't need access to the module's source repository, even when they run Terraform from the command line.

Place each module that is intended to be reusable in the root of its own repository, but you can also reference modules from subdirectories.

Use recommended module sources

Terraform uses the `source` argument in a module block to find and download the source code for a child module.

Use local paths for closely related modules that have the primary purpose of factoring out repeated code elements, and a module registry or a VCS provider for modules that are intended to be shared by multiple configurations.

The most common source types for sharing modules are registries and version control systems. Choose based on how much control you need over the code that runs:

- **Public Terraform Registry:** set `source` to the module address, such as `terraform-aws-modules/lambda/aws`, and set `version` to an exact version, such as `8.12.0`. Change the

version deliberately, after you review the release. This is the simplest option, and tools such as Dependabot and Renovate can propose upgrades. The trade-off is that the publisher controls what a version number points to, and the dependency lock file covers providers, not modules.

- **Git repository pinned to a commit:** set `source` to the module's repository with `?ref=` and a full commit SHA, and record the matching release in a comment. The code that runs is always the code that you reviewed. The trade-off is that upgrades mean finding and reviewing a new SHA, and Terraform downloads the whole repository.
- **Private registry in HCP Terraform or Terraform Enterprise:** set `source` to `<hostname>/<organization>/<name>/<provider>`, such as `app.terraform.io/example-org/eks/aws`, and set `version`. Your organization controls the published versions, and teams can discover approved modules in one place. The trade-off is that you depend on HCP Terraform or Terraform Enterprise for every `terraform init`.
- **Other VCS providers:** for repositories in other Git services, use a `git::` source with the repository URL and a `?ref=` tag or commit SHA. Prefer SSH or HTTPS URLs that use the credentials your CI/CD pipeline already has.

Some modules have inputs that change their behavior in ways that matter at scale. For example, set `trigger_on_package_timestamp = false` in `terraform-aws-modules/lambda/aws`. Its default, `true`, plans a new deployment package on every run, even when the source hasn't changed.

Follow coding standards

Apply consistent Terraform formatting rules and styles across all configuration files. Enforce standards by using automated style checks in CI/CD pipelines. When you embed coding best practices into team workflows, configurations remain readable, maintainable, and collaborative across your organization.

Follow style guidelines

- Format all Terraform files (`.tf` files) with the terraform fmt command to match HashiCorp style standards.
- Use the terraform validate command to verify the syntax and structure of your configuration.
- Statically analyze code quality by using TFLint. This linter checks for Terraform best practices beyond just formatting and fails builds when it encounters errors.

Configure pre-commit hooks

Configure client-side pre-commit hooks that run `terraform fmt`, `terraform validate`, `tflint`, a security scanner such as `trivy config` or `checkov`, and `terraform-docs` before you allow commits. This practice helps you validate standards conformance earlier in developer workflows.

Use pre-commit frameworks such as pre-commit to add Terraform linting, formatting, and code scanning as hooks on your local machine. Hooks run on each Git commit and fail the commit if checks don't pass.

Moving style and quality checks to local pre-commit hooks provides rapid feedback to developers before they introduce changes. Standards become part of the coding workflow.

Refactor and import without re-creating resources

Use configuration blocks instead of `terraform state` commands to change what Terraform manages. The blocks are reviewed in a pull request like any other change, and `terraform plan` shows their effect before anything happens.

- **moved** (Terraform 1.1 and later): Rename a resource, or move it into or out of a module, without destroying and re-creating it.
- **removed** (Terraform 1.7 and later): Stop managing a resource. With `destroy = false`, Terraform removes the resource from state and leaves it running.
- **import** (Terraform 1.5 and later): Bring an existing resource under management. Add `-generate-config-out=<file>` to `terraform plan` to have Terraform write the resource configuration for you.

For example, a `moved` block with `from = aws_ssm_parameter.old_name` and `to = aws_ssm_parameter.feature_flag` renames the resource in state when you rename it in the configuration. Run `terraform plan` to confirm that the plan shows a move and no destroy.

To find existing resources to import, use `terraform query` (Terraform 1.14 and later) with `list` blocks in a `.tfquery.hcl` file. `terraform query -generate-config-out=<file>` writes an `import` block and a `resource` block for each result. A `list` block covers the whole account and Region for its resource type, so set a `limit`, and review and filter the results before you import them.

For more information, see Refactoring and Import in the HashiCorp documentation.

Best practices for testing

Testing catches mistakes before they reach an environment. Terraform provides two layers of checks. Validation in the configuration (variable validation, preconditions and postconditions, and `check` blocks) runs during every plan. The `terraform test` command (Terraform 1.6 and later) runs test files against a module, either with a mocked provider or by creating real resources. Run both in your pipeline, together with the formatting, linting, and scanning steps described in Follow coding standards.

Validate inputs and assumptions in the configuration

- **Variable validation** rejects a bad input before Terraform plans anything. Use it for rules about a single input, such as a name format.
- **Preconditions and postconditions** in a resource's `lifecycle` block check assumptions about the resource or its environment, such as the Region that the provider uses. A failed condition stops the run.
- **check blocks** (Terraform 1.5 and later) report a warning without stopping the run. Use them for ongoing checks, such as whether an endpoint responds.

In a `check` block, refer to input values rather than to resources that the same run creates. For example, a `check` block that reads a bucket with a data source should look it up by `var.name`, not by `aws_s3_bucket.this.id`. If it refers to the bucket resource, Terraform can't read it until after the apply, and plan-mode tests fail with `Check block assertion known after apply`.

Write unit tests with mock providers

A `mock_provider` block (Terraform 1.7 and later) replaces the AWS Provider with generated values, so tests run in seconds and need no AWS credentials. Use mocks to test variable validation, conditions, and the values that the module computes. Inside `mock_provider`, `mock_data` and `mock_resource` blocks set the values that data sources and resources return.

The following test file checks that a module rejects a bucket name that breaks its `validation` rule for `var.name`. The run succeeds only if the validation fails:

```
mock_provider "aws" {}
```

```
run "uppercase_name_rejected" {
  command = plan

  variables {
    name = "Not_Valid"
  }

  expect_failures = [var.name]
}
```

To test a failed precondition or `check` block, list the resource or `check.<name>` in `expect_failures`. A run that fails more than one condition must list all of them.

In Terraform 1.17 and later, `mock_provider` also accepts `mock_ephemeral` blocks. With them, unit tests can run configurations that read secrets through ephemeral resources, such as the example in Keep secrets out of state, without calling the AWS provider. Set `secret_string` in a `mock_ephemeral "aws_secretsmanager_secret_version"` block.

Write integration tests

Integration tests create real resources, check them, and then destroy them. Run them in a dedicated test account. In the test file, configure the provider with `allowed_account_ids` set to that account, and with default tags that identify test resources, so that you can find anything that a failed run leaves behind. Use `command = apply` in the `run` blocks, and assert on attributes that only AWS can set, such as ARNs.

`terraform test` destroys the resources that each test file creates, even when an assertion fails. Keep integration tests in their own file, such as `tests/integration.tftest.hcl`, so that you can run the fast unit tests on every commit and the integration tests on a schedule or before a release. To run one file, use `terraform test -filter=tests/unit.tftest.hcl`.

Use both kinds of test, because each catches what the other misses:

- **Unit tests with mocks** are fast, low-cost, and safe to run on every commit, and they need no credentials. They can't tell you whether AWS accepts the configuration, because the mock returns whatever you tell it to.
- **Integration tests** prove that the configuration works against real APIs, including quotas, permissions, and service behavior. They take minutes, cost money while the resources exist, need a test account and credentials, and can leave resources behind if a run is interrupted.

For more information, see Tests in the HashiCorp documentation and the AWS Prescriptive Guidance pattern Test AWS infrastructure by using LocalStack and Terraform Tests.

Best practices for AWS Provider version management

Managing versions of the AWS Provider and associated Terraform modules carefully helps maintain stability. This section outlines best practices around version constraints, locking, and upgrades.

Commit the dependency lock file

`terraform init` writes the exact provider versions it selects, and their checksums, to the dependency lock file, `.terraform.lock.hcl`. Commit this file in every root module. Later runs, on any machine, install the same versions until you deliberately upgrade:

- To move to a newer version that the constraints allow, run `terraform init -upgrade`, review the change to the lock file, and commit it.
- If you change a version constraint so that it excludes the locked version, `terraform init` fails until you run it with `-upgrade`.
- In CI, run `terraform init -lockfile=readonly` so that a job fails instead of changing the lock file.
- The lock file records checksums for the platforms where `init` ran. If developers and CI use different operating systems, add the other platforms' checksums: `terraform providers lock -platform=linux_amd64 -platform=linux_arm64 -platform=darwin_arm64`.

Add automated version checks

Add version checks for Terraform providers in your CI/CD pipelines to validate version pinning, and fail builds if the version is undefined.

- Add TFLint checks in CI/CD pipelines. The `terraform_required_version` and `terraform_required_providers` rules report modules that don't declare their Terraform version or provider versions. The TFLint ruleset for the AWS Provider adds rules that detect possible errors and check for best practices about AWS resources.
- Fail CI runs that detect unpinned provider versions to prevent implicit upgrades from reaching production.

Automate upgrade proposals

Use a dependency update tool such as Dependabot or Renovate to open pull requests when new provider or module versions are released. Each pull request then runs your normal checks and tests. Before you merge one, check that the updated lock file has checksums for every platform that runs Terraform. If it doesn't, add them with `terraform providers lock`.

Upgrade major versions in steps

Major versions of the AWS Provider remove deprecated arguments and resources. For each major upgrade, follow the provider's upgrade guide, such as the version 6 upgrade guide, in this order:

1. Upgrade to the latest release of your current major version.
2. Run `terraform plan`, and fix every deprecation warning until the plan is clean.
3. Change the version constraint, for example from `~> 5.0` to `~> 6.0`, and run `terraform init -upgrade`.
4. Run `terraform plan` again. Fix any differences, and any deprecation warnings that only the new version reports.
5. Apply in non-production environments first, and then in production.

Monitor new releases

- Monitor provider release notes and changelog feeds. Get notifications on new major/minor releases.
- Assess release notes for potentially breaking changes and evaluate their impact on your existing infrastructure.
- Upgrade minor versions in non-production environments first to validate them before updating the production environment.

By automating version checks in pipelines and monitoring new releases, you can catch unsupported upgrades early and give your teams time to evaluate the impact of new major/minor releases before you update production environments.

Contribute to providers

Actively contribute to the HashiCorp AWS Provider by reporting defects or requesting features in GitHub issues:

- Open well-documented issues on the AWS Provider repository to detail any bugs you encountered or functionality that is missing. Provide reproducible steps.
- Request and vote on enhancements to expand the capabilities of the AWS Provider for managing new services.
- When you contribute a fix or an enhancement, link your pull request to the issue that it addresses.
- Follow the contribution guidelines in the repository for coding conventions, testing standards, and documentation.

By contributing to the providers you use, you can provide direct input into their roadmap and help improve their quality and capabilities for all users.

Best practices for community modules

Using modules effectively helps you manage complex Terraform configurations and promote reuse. This section covers recommended practices for community modules, dependencies, sources, abstraction, and contributions.

Discover community modules

Search the Terraform Registry, GitHub, and other sources for existing AWS modules that might solve your use case before you build a new module. Look for modules that have recent updates and are actively maintained.

Use variables for customization

When you use community modules, pass inputs through variables instead of forking or directly modifying the source code. Override defaults where required instead of changing the internals of the module.

Limit forking to contributing fixes or features back to the original module to benefit the broader community.

Understand dependencies

Before you use the module, review its source code and documentation to identify dependencies:

- **Required providers**: Note the versions of AWS, Kubernetes, or other providers the module requires.
- **Nested modules**: Check for other modules used internally that introduce cascading dependencies.
- **External data sources**: Note the APIs, custom plugins, or infrastructure dependencies that the module relies on.

By mapping out the full tree of direct and indirect dependencies, you can prevent unexpected issues when you use the module.

Use trusted sources

Sourcing Terraform modules from unverified or unknown publishers introduces significant risk. Use modules only from sources that you trust, and review them like any other third-party code.

- On the Terraform Registry, modules with the verified badge are published by HashiCorp partners. AWS publishes verified modules in the aws-ia namespace.
- Widely used community namespaces, such as terraform-aws-modules, aren't verified. Review their maintainers, release history, and open issues before you adopt them.
- For custom modules, review publisher history, support levels, and usage reputation, even if the module is from your own organization.
- Pin each module to an exact version, and review the changes before you upgrade. For tighter supply chain control, pin a commit SHA, as described in Use recommended module sources.

Restricting modules to trusted, vetted sources reduces the risk of introducing vulnerabilities or maintenance issues into your code.

Subscribe to notifications

Subscribe to notifications for new module releases from trusted publishers:

- Watch GitHub module repositories to get alerts on new versions of the module.
- Monitor publisher blogs and changelogs for updates.
- Get proactive notifications for new versions from verified, well-maintained sources instead of automatically receiving updates.

Consuming modules only from trusted sources and monitoring changes helps maintain stability and reduce security risk across your supply chain.

Contribute to community modules

Submit fixes and enhancements for community modules that are hosted in GitHub:

- Open pull requests on modules to address defects or limitations that you encounter in your usage.

- Request new best practice configurations to be added to existing OSS modules by creating issues.

Contributing to community modules enhances reusable, codified patterns for all Terraform practitioners.

FAQ

Q. Why focus on the AWS Provider?

A. The AWS Provider is a widely used and complex provider for provisioning infrastructure in Terraform. Following these best practices helps you optimize your use of the provider in the AWS environment.

Q. I'm new to Terraform. Can I use this guide?

A. The guide is for people who are new to Terraform as well as more advanced practitioners who want to expand their skills. The practices improve workflows for users at any stage of learning.

Q. What are some key best practices covered?

A. Key best practices include using IAM roles over access keys, pinning versions, automated testing, remote state locking, credential rotation, contributing back to providers, and logically organizing code bases.

Q. Where can I learn more about Terraform?

A. The Resources section includes links to the official HashiCorp Terraform documentation and community forums. Use the links to learn more about advanced Terraform workflows.

Next steps

Consider the following next steps after reading this guide:

- If you have an existing Terraform code base, review your configuration and identify areas for improvement based on the recommendations in this guide. For example, review best practices for implementing remote backends, separating code into modules, and using version pinning, and validate these in your configuration.
- If you don't have an existing Terraform code base, use these best practices when you structure your new configuration. Apply the guidance on state management, authentication, and code structure from the beginning.
- Explore the HashiCorp community modules referenced in this guide to determine whether they simplify your architecture patterns. The modules provide higher levels of abstraction, so you don't have to rewrite common resources.
- Enable formatting, linting, security scans, policy checks, and automated tests to reinforce the best practices around security, compliance, and code quality. Tools such as TFLint, Trivy, Checkov, and `terraform test` can help.
- Review the latest AWS Provider documentation and changelog to see if there are any new resources or functionality that could help optimize your Terraform usage. Stay up to date on new versions of the AWS Provider.
- For additional guidance, see the Terraform documentation, best practices guide, and style guide on the HashiCorp website.

Resources

References

The following links provide additional reading material for the Terraform AWS Provider and using Terraform for IaC on AWS.

- Terraform AWS Provider (HashiCorp documentation)
- AWS Provider version 6 upgrade guide (HashiCorp documentation)
- Terraform modules for AWS services (Terraform Registry)
- Dynamic credentials with the AWS Provider (HCP Terraform documentation)
- S3 backend (Terraform documentation)
- Policy enforcement (HCP Terraform documentation)
- Getting started with Terraform: Guidance for AWS CDK and AWS CloudFormation experts (AWS Prescriptive Guidance)
- Securing sensitive data by using AWS Secrets Manager and HashiCorp Terraform (AWS Prescriptive Guidance)
- Working around Terraform data source limitations on AWS (AWS Prescriptive Guidance)

Tools

The following tools help improve code quality and automation of Terraform configurations on AWS, as recommended in this best practices guide.

Code quality:

- Checkov: Scans Terraform code to identify misconfigurations before deployment.
- Trivy: Scans Terraform code for misconfigurations with `trivy config`.
- TFLint: Identifies possible errors, deprecated syntax, and unused declarations. This linter can also enforce AWS best practices and naming conventions.
- terraform-docs: Generates documentation from Terraform modules in various output formats.

Automation tools:

- HCP Terraform: Helps teams version, collaborate, and build Terraform workflows with policy checks and approval gates.
- Atlantis: An open-source Terraform pull request automation tool for validating code changes.

Document history

The following table describes significant changes to this guide.

Change	Description	Date
Updated for Terraform 1.11 and later and AWS Provider 6.x	Updated authentication, secrets handling, state bucket controls, version managemen t, policy as code, and code examples. Added the Best practices for testing section.	October 29, 2026
Remote state locking update	Updated the Backend best practices section to reflect Amazon S3 native state locking, which is now the recommended approach.	August 26, 2025
Initial publication	—	May 28, 2024