Best practices for using the Terraform AWS Provider

AWS Prescriptive Guidance

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

Table of Contents

Best practices for using the Terraform AWS Provider

Michael Begin, Senior DevOps Consultant, Amazon Web Services (AWS)

May 2024 (document history)

Managing infrastructure as code (IaC) with Terraform on AWS offers important benefits such as improved consistency, security, and agility. However, as your Terraform configuration grows in size and complexity, it becomes critical to follow best practices to avoid pitfalls.

This guide provides recommended best practices for using the Terraform AWS Provider from HashiCorp. It walks you through proper versioning, security controls, remote backends, codebase structure, and community providers to optimize Terraform on AWS. Each section dives into more details on the specifics of applying these best practices:

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

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 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 through best practice implementation.

Target audience

The target audience for this guide includes technical leads and managers who oversee teams that use Terraform for IaC on AWS. Other potential readers include infrastructure engineers, DevOps engineers, solutions architects, and developers who actively use Terraform to manage AWS infrastructure.

Following these best practices will save time and help unlock the benefits of IaC for these roles.

Overview

Terraform providers are plugins that allow Terraform to interact with different APIs. The Terraform AWS Provider is the official plugin for managing AWS infrastructure as code (IaC) with Terraform. It translates Terraform syntax into AWS API calls to create, read, update, and delete AWS resources.

The AWS Provider handles authentication, translating Terraform syntax to AWS API calls, and provisioning resources in AWS. You use a Terraform `provider` code block to configure the provider plugin that Terraform uses to interact with the AWS API. You can configure multiple AWS Provider blocks to manage resources across different AWS accounts and Regions.

Here's an example Terraform configuration that uses multiple AWS Provider blocks with aliases to manage an Amazon Relational Database Service (Amazon RDS) database that has a replica in a different Region and account. The primary and secondary providers assume different AWS Identity and Access Management (IAM) roles:

```
# Configure the primary AWS Provider
provider "aws" {
  region = "us-west-1"
  alias  = "primary"
}

# Configure a secondary AWS Provider for the replica Region and account
provider "aws" {
  region      = "us-east-1"
  alias       = "replica"
  assume_role {
    role_arn     = "arn:aws:iam::<replica-account-id>:role/<role-name>"
    session_name = "terraform-session"
  }
}

# Primary Amazon RDS database
resource "aws_db_instance" "primary" {
  provider = aws.primary

  # ... RDS instance configuration
}

# Read replica in a different Region and account
resource "aws_db_instance" "read_replica" {
```

```
  provider = aws.replica

  # ... RDS read replica configuration
  replicate_source_db = aws_db_instance.primary.id
}
```

In this example:

- The first `provider` block configures the primary AWS Provider in the `us-west-1` Region with the alias `primary`.
- The second `provider` block configures a secondary AWS Provider in the `us-east-1` Region with the alias `replica`. This provider is used to create a read replica of the primary database in a different Region and account. The `assume_role` block is used to assume an IAM role in the replica account. The `role_arn` specifies the Amazon Resource Name (ARN) of the IAM role to assume, and `session_name` is a unique identifier for the Terraform session.
- The `aws_db_instance.primary` resource creates the primary Amazon RDS database by using the `primary` provider in the `us-west-1` Region.
- The `aws_db_instance.read_replica` resource creates a read replica of the primary database in the `us-east-1` Region by using the `replica` provider. The `replicate_source_db` attribute references the ID of the `primary` database.

Security best practices

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

- IAM roles and permissions for least-privilege access
- Securing credentials to help prevent unauthorized access to AWS accounts and resources
- Remote state encryption to help protect sensitive data
- Infrastructure and source code scanning to identify misconfigurations
- Access controls for remote state storage
- Sentinel policy enforcement to implement 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 emphasized multiple times in this section because it directly relates to how Terraform authenticates and runs actions against cloud providers such as AWS.

When you use Terraform to provision and manage AWS resources, it acts on behalf of an entity (user or role) that requires appropriate permissions to make API calls. Not following least privilege opens up major 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.
- Not following least privilege goes against 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 automatically rotate, 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:

- Enable IAM Access Analyzer to evaluate policies and highlight unused permissions that can be removed.
- 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.

Assume IAM roles for local authentication

When you run Terraform locally, avoid configuring static access keys. Instead, use IAM roles to grant privileged access temporarily without exposing long-term credentials.

First, create an IAM role with the necessary minimum permissions and add a trust relationship that allows the IAM role to be assumed by your user account or federated identity. This authorizes temporary usage of the role.

Trust relationship policy example:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:role/terraform-execution"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Then, run the AWS CLI command **aws sts assume-role** to retrieve short-lived credentials for the role. These credentials are typically valid for one hour.

AWS CLI command example:

```
aws sts assume-role --role-arn arn:aws:iam::111122223333:role/terraform-execution --
role-session-name terraform-session-example
```

The output of the command contains an access key, secret key, and session token that you can use to authenticate to AWS:

```
{
    "AssumedRoleUser": {
        "AssumedRoleId": "AROA3XFRBF535PLBIFPI4:terraform-session-example",
        "Arn": "arn:aws:sts::111122223333:assumed-role/terraform-execution/terraform-
session-example"
    },
    "Credentials": {
        "SecretAccessKey": " wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
        "SessionToken": " AQoEXAMPLEH4aoAH0gNCAPyJxz4BlCFFxWNE1OPTgk5TthT
+FvwqnKwRcOIfrRh3c/LTo6UDdyJwOOvEVPvLXCrrrUtdnniCEXAMPLE/
IvU1dYUg2RVAJBanLiHb4IgRmpRV3zrkuWJOgQs8IZZaIv2BXIa2R4OlgkBN9bkUDNCJiBeb/
AXlzBBko7b15fjrBs2+cTQtpZ3CYWFXG8C5zqx37wnOE49mRl/+OtkIKGO7fAE",
        "Expiration": "2024-03-15T00:05:07Z",
        "AccessKeyId": "ASIAIOSFODNN7EXAMPLE"
    }
}
```

The AWS Provider can also automatically handle assuming the role.

Provider configuration example for assuming an IAM role:

```
provider "aws" {
  assume_role {
    role_arn     = "arn:aws:iam::111122223333:role/terraform-execution"
    session_name = "terraform-session-example"
  }
}
```

This grants elevated privilege strictly for the Terraform session's duration. The temporary keys cannot be leaked because they expire automatically after the maximum duration of the session.

The key benefits of this best practice include improved security compared with long-lived access keys, fine-grained access controls on the role for least privileges, and the ability to easily revoke access by modifying the role's permissions. By using IAM roles, you also avoid having to directly store secrets locally in scripts or on disk, which helps you share Terraform configuration securely across a team.

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 EC2 instances to inherit the permissions defined in the role. Then, launch instances by specifying that instance profile. The instance will authenticate through the attached role.

Before you run any Terraform operations, verify that the role is present in the instance metadata to confirm that the credentials were successfully inherited.

```
TOKEN=$(curl -s -X PUT "http://169.254.169.254/latest/api/token" -H "X-aws-ec2-
metadata-token-ttl-seconds: 21600")

curl -H "X-aws-ec2-metadata-token: $TOKEN" -s http://169.254.169.254/latest/meta-data/
iam/security-credentials/
```

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

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 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 credentials to simplify and secure AWS authentication. Terraform automatically exchanges temporary credentials on each run without needing IAM role assumption.

Benefits include easier secret rotation, centralized credential management across workspaces, least-privilege permissions, and eliminating hardcoded keys. Relying on hashed ephemeral keys enhances security compared with long-lived access keys.

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.

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 having to directly manage long-term AWS access keys within GitLab.

Credentials are exchanged just-in-time, which improves security. Users also gain least privilege access according to the permissions in the IAM role.

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. As you migrate to more modern tools or tools, 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, we recommend that you provide these users with only the permissions they require to perform the task and that you 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, additional permissions might get granted that can exceed the minimum policies required. 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 unused 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 set up alerts and notifications for administrators. Here are some options:

- AWS Config: You can use AWS Config rules to evaluate the configuration settings of your AWS resources, including IAM access keys. You can create custom rules to check for specific conditions, such as unused access keys that are older than a specific number of days. When a rule is violated, AWS Config can start an evaluation for remediation or send notifications to an Amazon Simple Notification Service (Amazon SNS) topic.
- AWS Security Hub: Security Hub provides a comprehensive view of your AWS account's security posture and can help detect and notify you about potential security issues, including unused or inactive IAM access keys. Security Hub can integrate with Amazon EventBridge and Amazon SNS or AWS Chatbot to send notifications to administrators.
- AWS Lambda: Lambda functions can be called by various events, including Amazon CloudWatch Events or AWS Config rules. You can write custom Lambda functions to evaluate IAM access key usage, perform additional checks, and send notifications by using services such as Amazon SNS or AWS Chatbot.

Continually validate IAM policies

Use IAM Access Analyzer to evaluate policies that are attached to roles and identify any unused services or excess actions that were granted. Implement periodic access reviews to manually verify that policies match current requirements.

Compare the existing policy with the policy generated by IAM Access Analyzer and remove any unnecessary permissions. You should also provide reports to users and automatically revoke unused permissions after a grace period. This helps ensure that minimal policies remain in effect.

Proactively and frequently revoking obsolete access minimizes the credentials that might be at risk during a breach. 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.

Failure to secure remote state can lead to serious issues such as loss of state data, inability to manage infrastructure, inadvertent resource deletion, and exposure of sensitive information that might be present in the state file. For this reason, securing remote state storage is crucial for production-grade Terraform usage.

Enable encryption and access controls

Use Amazon Simple Storage Service (Amazon S3) server-side encryption (SSE) to encrypt remote state at rest.

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 protections help reduce the attack surface, and collaboration workflows enable productivity.

Use AWS Secrets Manager

There are many resources and data sources in Terraform that store secret values in plaintext in the state file. Avoid storing secrets in state—use AWS Secrets Manager instead.

Instead of attempting to manually encrypt sensitive values, rely on Terraform's built-in support for sensitive state management. When exporting sensitive values to output, make sure that the values are marked as sensitive.

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 such as Amazon Inspector, AWS Security Hub, Amazon Detective, and Amazon GuardDuty to monitor provisioned infrastructure across accounts and Regions. Schedule recurring scans in Security Hub to track deployment and configuration drift. Scan EC2 instances, Lambda functions, containers, S3 buckets, and other resources.

Perform static analysis

Embed static analyzers such as Checkov directly into CI/CD pipelines to scan Terraform configuration code (HCL) and identify risks preemptively before deployment. This moves security checks to an earlier point in the development process (referred to as *shifting left*) and prevents misconfigured infrastructure.

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 preventative, detective, and reactive controls while embedding security earlier into the software development lifecycle (SDLC).

Enforce policy checks

Use code frameworks such as HashiCorp Sentinel policies to provide governance guardrails and standardized templates for infrastructure provisioning with Terraform.

Sentinel policies can define requirements or restrictions on Terraform configuration to align with organizational standards and best practices. For example, you can use Sentinel policies to:

- Require tags on all resources.
- Restrict instance types to an approved list.
- Enforce mandatory variables.
- Prevent the destruction of production resources.

Embedding policy checks into Terraform configuration lifecycles enables proactive enforcement of standards and architecture guidelines. Sentinel provides shared policy logic that helps accelerate development while preventing unapproved practices.

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 an optimal 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 low-cost backend for managing Terraform state on AWS. The global footprint and resilience of Amazon S3 exceeds what most teams can achieve by 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 that are fully in the AWS Cloud, Amazon S3 is typically the ideal choice over Kubernetes or HashiCorp Consul clusters. The simplicity, resilience, and tight AWS integration of Amazon S3 state storage provides an optimal 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 will lead to more collaborative Terraform code bases while limiting the impact of errors or unauthorized modifications. By implementing a well-architected remote backend, teams can optimize Terraform workflows.

Best practices:

- Use Amazon S3 for remote storage
- Facilitate team collaboration
- Separate the backends for each environment
- Actively monitor remote state activity

Use Amazon S3 for remote storage

Storing Terraform state remotely in Amazon S3 and implementing state locking and consistency checking by using Amazon DynamoDB provide major benefits over local file storage. Remote state enables team collaboration, change tracking, backup protections, and remote locking for increased safety.

Using Amazon S3 with the S3 Standard storage class (default) instead of ephemeral local storage or self-managed solutions provides 99.999999999% durability and 99.99% availability protections to prevent accidental state data loss. AWS managed services such as Amazon S3 and DynamoDB provide service-level agreements (SLAs) that exceed what most organizations can achieve when they self-manage storage. Rely on these protections to keep remote backends accessible.

Enable remote state locking

DynamoDB locking restricts state access to prevent concurrent write operations. This prevents simultaneous modifications from multiple users and reduces errors.

Example backend configuration with state locking:

```
terraform {
  backend "s3" {
    bucket         = "myorg-terraform-states"
    key            = "myapp/production/tfstate"
    region         = "us-east-1"
    dynamodb_table = "TerraformStateLocking"
  }
}
```

Enable versioning and automatic backups

For additional safeguarding, enable automatic versioning and backups by using AWS Backup on Amazon S3 backends. Versioning preserves all previous versions of the state whenever changes are made. It also lets you restore previous working state snapshots if needed to roll back unwanted changes or recover from accidents.

Restore previous versions if needed

Versioned Amazon S3 state buckets make it easy to revert changes by restoring a previous known good 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 also use Sentinel policy as code to implement governance controls.

Although HCP Terraform requires using a software as a service (SaaS) platform, for many teams the benefits around security, access controls, automated policy checks, and collaboration features make it an optimal choice over self-managing state storage with Amazon S3 or DynamoDB.

Easy integration with services such as GitHub and GitLab with minor configuration also appeals to users who fully embrace cloud and SaaS tools for better 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 increases business agility across the enterprise.

Improve accountability by using AWS CloudTrail

Integrate AWS CloudTrail with the Amazon S3 bucket to capture API calls made to the state bucket. Filter CloudTrail events to track `PutObject`, `DeleteObject,` and other relevant calls.

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

Combine CloudTrail logs with Amazon S3 state 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 preventative improvements.

We also recommend that you enforce IAM permissions to limit state bucket access. Overall, S3 Versioning and CloudTrail monitoring supports auditing across infrastructure changes. Teams gain improved accountability, transparency, and audit capabilities into 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.

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

Lock down 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 breach. 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

Continuously monitoring remote state activity is critical for detecting potential issues early. Look for anomalous unlocks, changes, or access attempts.

Get alerts on suspicious unlocks

Most state changes should run through CI/CD pipelines. Generate alerts if state unlocks occur directly through developer workstations, which could signal unauthorized or untested changes.

Monitor access attempts

Authentication failures on state buckets might indicate reconnaissance activity. Notice if multiple accounts are trying to access state, or unusual IP addresses appear, which signals compromised credentials.

Best practices for code base structure and organization

Proper code base structure and organization is critical as Terraform usage grows across large teams and enterprises. A well-architected code base enables collaboration at scale while enhancing maintainability.

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.

By implementing code base structure guidelines around modularity, documentation, standards, and logical organization, you can support broad collaboration across teams while keeping Terraform maintainable as usage spreads across an organization. By enforcing conventions and standards, you can avoid the complexity of a fragmented code base.

Best practices:

- Implement a standard repository structure
- Structure for modularity
- Follow naming conventions
- Use attachment resources
- Use default tags
- Meet Terraform registry requirements
- Use recommended module sources
- Follow coding standards

Implement a standard repository structure

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

- **Root module or directory**: This should be the primary entry point for both Terraform root and re-usable modules and is expected to be unique. If you have a more complex architecture, you can use nested modules to create lightweight abstractions. This helps you describe infrastructure in terms of its architecture instead of directly, in terms of physical objects.
- **README**: The root module and any nested modules should have README files. This file must be named `README.md`. It should contain a description of the module and what it should be used for. If you want to include an example of using this module with other resources, put it in an `examples` directory. Consider including a diagram that depicts the infrastructure resources the module might create and their relationships. Use terraform-docs to automatically generate inputs or outputs of the module.
- **main.tf**: This is the primary entry point. For a simple module, all resources might be created in this file. For a complex module, resource creation might be spread across multiple files, but any nested module calls should be in the `main.tf` file.
- **variables.tf** and **outputs.tf**: These files contain the declarations for variables and outputs. All variables and outputs should have one-sentence or two-sentence descriptions that explain their purpose. These descriptions are used for documentation. For more information, see the HashiCorp documentation for variable configuration and output configuration.
  - All variables must have a defined type.
  - The variable declaration can also include a default argument. If the declaration includes a default argument, the variable is considered to be optional, and the default value is used if you don't set a value when you call the module or run Terraform. The default argument requires a literal value and cannot reference other objects in the configuration. To make a variable required, omit a default in the variable declaration and consider whether setting `nullable = false` makes sense.
  - For variables that have environment-independent values (such as `disk_size`), provide default values.
  - For variables that have environment-specific values (such as `project_id`), don't provide default values. In this case, the calling module must provide meaningful values.

  - Use empty defaults for variables such as empty strings or lists only when leaving the variable empty is a valid preference that the underlying APIs don't reject.
  - Be judicious in your use of variables. Parameterize values only if they must vary for each instance or environment. When you decide whether to expose a variable, ensure that you have a concrete use case for changing that variable. If there's only a small chance that a variable might be needed, don't expose it.
    - Adding a variable with a default value is backward compatible.
    - Removing a variable is backward incompatible.
    - In cases where a literal is reused in multiple places, you should use a local value without exposing it as a variable.
  - Don't pass outputs directly through input variables, because doing so prevents them from being properly added to the dependency graph. To ensure that implicit dependencies are created, make sure that outputs reference attributes from resources. Instead of referencing an input variable for an instance directly, pass the attribute.
- **locals.tf**: This file contains local values that assign a name to an expression, so a name can be used multiple times within a module instead of repeating the expression. Local values are like a function's temporary local variables. The expressions in local values aren't limited to literal constants; they can also reference other values in the module, including variables, resource attributes, or other local values, in order to combine them.
- **providers.tf**: This file contains the terraform block and provider blocks. `provider` blocks must be declared only in root modules by consumers of modules.

  If you're using HCP Terraform, also add an empty cloud block. The `cloud` block should be configured entirely through environment variables and environment variable credentials as part of a CI/CD pipeline.
- **versions.tf**: This file contains the required_providers block. All Terraform modules must declare which providers it requires so that Terraform can install and use these providers.
- **data.tf**: For simple configuration, put data sources next to the resources that reference them. For example, if you are fetching an image to be used in launching an instance, place it alongside the instance instead of collecting data resources in their own file. If the number of data sources becomes too large, consider moving them to a dedicated `data.tf` file.
- **.tfvars files**: For root modules, you can provide non-sensitive variables by using a `.tfvars` file. For consistency, name the variable files `terraform.tfvars`. Place common values at the root of the repository, and environment-specific values within the `envs/` folder.

- **Nested modules**: Nested modules should exist under the `modules/` subdirectory. Any nested module that has a `README.md` is considered usable by an external user. If a `README.md` doesn't exist, the module is considered for internal use only. Nested modules should be used to split complex behavior into multiple small modules that users can carefully pick and choose.

  If the root module includes calls to nested modules, these calls should use relative paths such as `./modules/sample-module` so that Terraform will consider them to be part of the same repository or package instead of downloading them again separately.

  If a repository or package contains multiple nested modules, they should ideally be composable by the caller instead of directly calling each other and creating a deeply nested tree of modules.
- **Examples**: Examples of using a reusable module should exist under the `examples/` subdirectory at the root of the repository. For each example, you can add a README to explain the goal and usage of the example. Examples for submodules should also be placed in the root `examples/` directory.

  Because examples are often copied into other repositories for customization, module blocks should have their source set to the address an external caller would use, not to a relative path.
- **Service named files**: Users often want to separate Terraform resources by service in multiple files. This practice should be discouraged as much as possible, and resources should be defined in `main.tf` instead. However, if a collection of resources (for example, IAM roles and policies) exceeds 150 lines, it's reasonable to break it into its own files, such as `iam.tf`. Otherwise, all resource code should be defined in the `main.tf`.
- **Custom scripts**: Use scripts only when necessary. Terraform doesn't account for, or manage, the state of resources that are created through scripts. Use custom scripts only when Terraform resources don't support the desired behavior. Place custom scripts called by Terraform in a `scripts/` directory.
- **Helper scripts**: Organize helper scripts that aren't called by Terraform in a `helpers/` directory. Document helper scripts in the `README.md` file with explanations and example invocations. If helper scripts accept arguments, provide argument checking and `--help` output.
- **Static files**: Static files that Terraform references but doesn't run (such as startup scripts loaded onto EC2 instances) must be organized into a `files/` directory. Place lengthy documents in external files, separate from their HCL. Reference them with the file() function.
- **Templates**: For files that the Terraform templatefile function reads in, use the file extension `.tftpl`. Templates must be placed in a `templates/` directory.

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 are called by the root module, any modules called by those modules, and so on).

Terraform root module layout basic example:

```
.
### data.tf
### envs
#   ### dev
#   #   ### terraform.tfvars
#   ### prod
#   #   ### terraform.tfvars
#   ### test
#       ### terraform.tfvars
### locals.tf
### main.tf
### outputs.tf
### providers.tf
### README.md
### terraform.tfvars
### 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. It's important to keep the module tree relatively flat to make it easier to reuse the 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
### variables.tf
### versions.tf
```

Structure for modularity

In principle, you can combine any resources and other constructs into a module, but overusing nested and reusable modules can make your overall Terraform configuration harder to understand and maintain, so use these modules in moderation.

When it makes sense, break your configuration into reusable modules that raise the level of abstraction by describing a new concept in your architecture that is constructed from resource types.

When you modularize your infrastructure into reusable definitions, aim for logical sets of resources instead of individual components or overly complex collections.

Don't wrap single resources

You shouldn't create modules that are thin wrappers around other single resource types. If you have trouble finding a name for your module that's different from the name of the main resource type inside it, your module probably isn't creating a new abstraction—it's adding unnecessary complexity. Instead, use the resource type directly in the calling module.

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 build tunnels through them.

By focusing modules on logical resource groupings that represent architecture patterns, teams can quickly configure reliable infrastructure foundations. Balance abstraction without over-engineering or over-simplification.

Reference resources in outputs

For every resource that's defined in a reusable module, 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.

For shared modules, define the minimum required provider versions in a required_providers block in `versions.tf`.

To declare that a module requires a particular version of the AWS provider, use a `required_providers` block inside a `terraform` block:

```
terraform {
  required_version = ">= 1.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 4.0.0"
    }
  }
}
```

If a shared module supports only a specific version of the AWS provider, use the *pessimistic constraint operator* (~>), which allows only the rightmost version component to increment:

```
terraform {
  required_version = ">= 1.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
```

```
  }
}
```

In this example, `~> 4.0` allows the installation of `4.57.1` and `4.67.0` but not `5.0.0`. For more information, see Version Constraint Syntax in the HashiCorp documentation.

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 have pseudo-resources embedded as attributes in them. Where possible, you should avoid using these embedded resource attributes and use the unique resource to attach that pseudo-resource instead. These resource relationships can cause cause-and-effect issues that are unique for each resource.

Using an embedded attribute (avoid this pattern):

```
resource "aws_security_group" "allow_tls" {
  ...
  ingress {
    description      = "TLS from VPC"
    from_port        = 443
    to_port          = 443
    protocol         = "tcp"
    cidr_blocks      = [aws_vpc.main.cidr_block]
    ipv6_cidr_blocks = [aws_vpc.main.ipv6_cidr_block]
  }

  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
}
```

Using attachment resources (preferred):

```
resource "aws_security_group" "allow_tls" {
  ...
}

resource "aws_security_group_rule" "example" {
  type              = "ingress"
  description      = "TLS from VPC"
  from_port        = 443
  to_port          = 443
  protocol         = "tcp"
  cidr_blocks      = [aws_vpc.main.cidr_block]
```

```
  ipv6_cidr_blocks = [aws_vpc.main.ipv6_cidr_block]
  security_group_id = aws_security_group.allow_tls.id
}
```

Use default tags

Assign tags to all resources that can accept tags. The Terraform AWS Provider has an aws_default_tags data source that you should use inside the root module.

Consider adding necessary tags to all resources that are created by a Terraform module. Here's a list of possible tags to attach:

- **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

A module repository must meet all of the following requirements so it can be published to a Terraform registry.

You should always follow these requirements even if you aren't planning to publish the module to a registry in the short term. By doing so, you can publish the module to a registry later without having to change the configuration and structure of the repository.

- Repository name: For a module repository, use the three-part name `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 must adhere to the standard repository structure. This allows the registry to inspect your module and generate documentation, track resource usage, and more.

  - After you create the Git repository, copy the module files to the root of the repository. We recommend that you place each module that is intended to be reusable in the root of its own repository, but you can also reference modules from subdirectories.
  - If you're using HCP Terraform, publish the modules that are intended to be shared to your organization registry. The registry handles downloads and controls access with HCP Terraform API tokens, so consumers do not need access to the module's source repository even when they run Terraform from the command line.
- Location and permissions: The repository must be in one of your configured version control system (VCS) providers, and the HCP Terraform VCS user account must have administrator access to the repository. The registry needs administrator access to create the webhooks to import new module versions.
- x.y.z tags for releases: At least one release tag must be present for you to publish a module. The registry uses release tags to identify module versions. Release tag names must use semantic versioning, which you can optionally prefix with a `v` (for example, `v1.1.0` and `1.1.0`). The registry ignores tags that do not look like version numbers. For more information about publishing modules, see the Terraform documentation.

For more information, see Preparing a Module Repository in the Terraform documentation.

Use recommended module sources

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

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

The following examples illustrate the most common and recommended source types for sharing modules. Registry modules support versioning. You should always provide a specific version, as shown in the following examples.

Registry

Terraform registry:

```
module "lambda" {
```

```
  source = "github.com/terraform-aws-modules/terraform-aws-lambda.git?
ref=e78cdf1f82944897ca6e30d6489f43cf24539374" #--> v4.18.0

  ...

}
```

By pinning commit hashes, you can avoid drift from public registries that are vulnerable to supply chain attacks.

HCP Terraform:

```
module "eks_karpenter" {
  source = "app.terraform.io/my-org/eks/aws"
  version = "1.1.0"

  ...

  enable_karpenter = true
}
```

Terraform Enterprise:

```
module "eks_karpenter" {
  source = "terraform.mydomain.com/my-org/eks/aws"
  version = "1.1.0"

  ...

  enable_karpenter = true
}
```

VCS providers

VCS providers support the `ref` argument for selecting a specific revision, as shown in the following examples.

GitHub (HTTPS):

```
module "eks_karpenter" {
```

```
  source = "github.com/my-org/terraform-aws-eks.git?ref=v1.1.0"

  ...

  enable_karpenter = true
}
```

Generic Git repository (HTTPS):

```
module "eks_karpenter" {
  source = "git::https://example.com/terraform-aws-eks.git?ref=v1.1.0"

  ...

  enable_karpenter = true
}
```

Generic Git repository (SSH):

> **Warning**
>
> You need to configure credentials to access private repositories.

```
module "eks_karpenter" {
  source = "git::ssh://username@example.com/terraform-aws-eks.git?ref=v1.1.0"

  ...

  enable_karpenter = true
}
```

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 as usage spreads widely across an 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`, `tflint`, `checkov`, and other code scans and style checks 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 changes are introduced. Standards become part of the coding workflow.

Best practices for AWS Provider version management

Carefully managing versions of the AWS Provider and associated Terraform modules is critical for stability. This section outlines best practices around version constraints and upgrades.

Best practices:

- Add automated version checks
- Monitor new releases
- Contribute to providers

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 to scan for provider versions that don't have pinned major/minor version constraints defined. Use the TFLint ruleset plugin for Terraform AWS Provider, which provides rules for detecting possible errors and checks for best practices about AWS resources.
- Fail CI runs that detect unpinned provider versions to prevent implicit upgrades from reaching 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 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.
- Reference issued pull requests when you contribute proposed fixes for provider defects or enhancements. Link to related issues.
- Follow the contribution guidelines in the repository for coding conventions, testing standards, and documentation.

By giving back 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 is key to managing complex Terraform configurations and promoting reuse. This section provides best practices around community modules, dependencies, sources, abstraction, and contributions.

Best practices:

- Discover community modules
- Understand dependencies
- Use trusted sources
- Contribute to community modules

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 popular options 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.

Forking should be limited to contributing fixes or features 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 avoid surprises when you use the module.

Use trusted sources

Sourcing Terraform modules from unverified or unknown publishers introduces significant risk. Use modules only from trusted sources.

- Favor certified modules from the Terraform Registry that are published by verified creators such as AWS or HashiCorp partners.
- For custom modules, review publisher history, support levels, and usage reputation, even if the module is from your own organization.

By not allowing modules from unknown or unvetted sources, you can reduce the risk of injecting 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, highly rated sources instead of implicitly pulling in updates.

Consuming modules only from trusted sources and monitoring changes provide stability and security. Vetted modules enhance productivity while minimizing supply chain risk.

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 one of the most widely used and complex providers for provisioning infrastructure in Terraform. Following these best practices help users optimize their usage of the provider for 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 level up 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, incorporating 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

Here are some potential next steps after reading this guide:

- If you have an existing Terraform code base, review your configuration and identify areas that could be improved based on the recommendations that are provided in this guide. For example, review best practices for implementing remote backends, separating code into modules, using version pinning, and so on, 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. Follow the advice around state management, authentication, code structure, and so on from the beginning.
- Try using some of the HashiCorp community modules referenced in this guide to see if they simplify your architecture patterns. The modules allow higher levels of abstraction, so you don't have to rewrite common resources.
- Enable linting, security scans, policy checks, and automated testing tools to reinforce some of the best practices around security, compliance, and code quality. Tools such as TFLint, tfsec, and Checkov can help.
- Review the latest AWS Provider documentation 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)
- Terraform modules for AWS services (Terraform Registry)
- The AWS and HashiCorp Partnership (HashiCorp blog post)
- Dynamic Credentials with the AWS Provider (HCP Terraform documentation)
- DynamoDB State Locking (Terraform documentation)
- Enforce Policy with Sentinel (Terraform documentation)

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.
- 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.
- CDK for Terraform: A framework that lets you use familiar languages such as TypeScript, Python, Java, C#, and Go instead of HashiCorp Configuration Language (HCL) to define, provision, and test your Terraform infrastructure as code.

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Initial publication	—	May 28, 2024

AWS Prescriptive Guidance glossary

The following are commonly used terms in strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the **Provide feedback** link at the end of the glossary.

Numbers

7 Rs

Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:

- Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
- Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
- Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
- Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
- Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. You migrate servers from an on-premises platform to a cloud service for the same platform. Example: Migrate a Microsoft Hyper-V application to AWS.
- Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later time, and legacy applications that you want to retain, because there's no business justification for migrating them.

- Retire – Decommission or remove applications that are no longer needed in your source environment.

A

ABAC

See attribute-based access control.

abstracted services

See managed services.

ACID

See atomicity, consistency, isolation, durability.

active-active migration

A database migration method in which the source and target databases are kept in sync (by using a bidirectional replication tool or dual write operations), and both databases handle transactions from connecting applications during migration. This method supports migration in small, controlled batches instead of requiring a one-time cutover. It's more flexible but requires more work than active-passive migration.

active-passive migration

A database migration method in which in which the source and target databases are kept in sync, but only the source database handles transactions from connecting applications while data is replicated to the target database. The target database doesn't accept any transactions during migration.

aggregate function

A SQL function that operates on a group of rows and calculates a single return value for the group. Examples of aggregate functions include `SUM` and `MAX`.

AI

See artificial intelligence.

AIOps

See artificial intelligence operations.

anonymization

The process of permanently deleting personal information in a dataset. Anonymization can help protect personal privacy. Anonymized data is no longer considered to be personal data.

anti-pattern

A frequently used solution for a recurring issue where the solution is counter-productive, ineffective, or less effective than an alternative.

application control

A security approach that allows the use of only approved applications in order to help protect a system from malware.

application portfolio

A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

artificial intelligence (AI)

The field of computer science that is dedicated to using computing technologies to perform cognitive functions that are typically associated with humans, such as learning, solving problems, and recognizing patterns. For more information, see What is Artificial Intelligence?

artificial intelligence operations (AIOps)

The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

asymmetric encryption

An encryption algorithm that uses a pair of keys, a public key for encryption and a private key for decryption. You can share the public key because it isn't used for decryption, but access to the private key should be highly restricted.

atomicity, consistency, isolation, durability (ACID)

A set of software properties that guarantee the data validity and operational reliability of a database, even in the case of errors, power failures, or other problems.

attribute-based access control (ABAC)

The practice of creating fine-grained permissions based on user attributes, such as department, job role, and team name. For more information, see ABAC for AWS in the AWS Identity and Access Management (IAM) documentation.

authoritative data source

A location where you store the primary version of data, which is considered to be the most reliable source of information. You can copy data from the authoritative data source to other locations for the purposes of processing or modifying the data, such as anonymizing, redacting, or pseudonymizing it.

Availability Zone

A distinct location within an AWS Region that is insulated from failures in other Availability Zones and provides inexpensive, low-latency network connectivity to other Availability Zones in the same Region.

AWS Cloud Adoption Framework (AWS CAF)

A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the AWS CAF website and the AWS CAF whitepaper.

AWS Workload Qualification Framework (AWS WQF)

A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.

B

bad bot

A bot that is intended to disrupt or cause harm to individuals or organizations.

BCP

See business continuity planning.

behavior graph

A unified, interactive view of resource behavior and interactions over time. You can use a behavior graph with Amazon Detective to examine failed logon attempts, suspicious API calls, and similar actions. For more information, see Data in a behavior graph in the Detective documentation.

big-endian system

A system that stores the most significant byte first. See also endianness.

binary classification

A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as “Is this email spam or not spam?" or "Is this product a book or a car?"

bloom filter

A probabilistic, memory-efficient data structure that is used to test whether an element is a member of a set.

blue/green deployment

A deployment strategy where you create two separate but identical environments. You run the current application version in one environment (blue) and the new application version in the other environment (green). This strategy helps you quickly roll back with minimal impact.

bot

A software application that runs automated tasks over the internet and simulates human activity or interaction. Some bots are useful or beneficial, such as web crawlers that index information on the internet. Some other bots, known as *bad bots*, are intended to disrupt or cause harm to individuals or organizations.

botnet

Networks of bots that are infected by malware and are under the control of a single party, known as a *bot herder* or *bot operator*. Botnets are the best-known mechanism to scale bots and their impact.

branch

A contained area of a code repository. The first branch created in a repository is the *main branch*. You can create a new branch from an existing branch, and you can then develop features or fix bugs in the new branch. A branch you create to build a feature is commonly referred to as a *feature branch*. When the feature is ready for release, you merge the feature branch back into the main branch. For more information, see About branches (GitHub documentation).

break-glass access

In exceptional circumstances and through an approved process, a quick means for a user to gain access to an AWS account that they don't typically have permissions to access. For more information, see the Implement break-glass procedures indicator in the AWS Well-Architected guidance.

brownfield strategy

The existing infrastructure in your environment. When adopting a brownfield strategy for a system architecture, you design the architecture around the constraints of the current systems and infrastructure. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

buffer cache

The memory area where the most frequently accessed data is stored.

business capability

What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

business continuity planning (BCP)

A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.

C

CAF

See AWS Cloud Adoption Framework.

canary deployment

The slow and incremental release of a version to end users. When you are confident, you deploy the new version and replace the current version in its entirety.

CCoE

See Cloud Center of Excellence.

CDC

See change data capture.

change data capture (CDC)

The process of tracking changes to a data source, such as a database table, and recording metadata about the change. You can use CDC for various purposes, such as auditing or replicating changes in a target system to maintain synchronization.

chaos engineering

Intentionally introducing failures or disruptive events to test a system's resilience. You can use AWS Fault Injection Service (AWS FIS) to perform experiments that stress your AWS workloads and evaluate their response.

CI/CD

See continuous integration and continuous delivery.

classification

A categorization process that helps generate predictions. ML models for classification problems predict a discrete value. Discrete values are always distinct from one another. For example, a model might need to evaluate whether or not there is a car in an image.

client-side encryption

Encryption of data locally, before the target AWS service receives it.

Cloud Center of Excellence (CCoE)

A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.

cloud computing

The cloud technology that is typically used for remote data storage and IoT device management. Cloud computing is commonly connected to edge computing technology.

cloud operating model

In an IT organization, the operating model that is used to build, mature, and optimize one or more cloud environments. For more information, see Building your Cloud Operating Model.

cloud stages of adoption

The four phases that organizations typically go through when they migrate to the AWS Cloud:

- Project – Running a few cloud-related projects for proof of concept and learning purposes
- Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)
- Migration – Migrating individual applications
- Re-invention – Optimizing products and services, and innovating in the cloud

These stages were defined by Stephen Orban in the blog post The Journey Toward Cloud-First & the Stages of Adoption on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the migration readiness guide.

CMDB

See configuration management database.

code repository

A location where source code and other assets, such as documentation, samples, and scripts, are stored and updated through version control processes. Common cloud repositories include GitHub or AWS CodeCommit. Each version of the code is called a *branch*. In a microservice structure, each repository is devoted to a single piece of functionality. A single CI/CD pipeline can use multiple repositories.

cold cache

A buffer cache that is empty, not well populated, or contains stale or irrelevant data. This affects performance because the database instance must read from the main memory or disk, which is slower than reading from the buffer cache.

cold data

Data that is rarely accessed and is typically historical. When querying this kind of data, slow queries are typically acceptable. Moving this data to lower-performing and less expensive storage tiers or classes can reduce costs.

computer vision (CV)

A field of AI that uses machine learning to analyze and extract information from visual formats such as digital images and videos. For example, AWS Panorama offers devices that add CV to on-premises camera networks, and Amazon SageMaker provides image processing algorithms for CV.

configuration drift

For a workload, a configuration change from the expected state. It might cause the workload to become noncompliant, and it's typically gradual and unintentional.

configuration management database (CMDB)

A repository that stores and manages information about a database and its IT environment, including both hardware and software components and their configurations. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration.

conformance pack

A collection of AWS Config rules and remediation actions that you can assemble to customize your compliance and security checks. You can deploy a conformance pack as a single entity in an AWS account and Region, or across an organization, by using a YAML template. For more information, see Conformance packs in the AWS Config documentation.

continuous integration and continuous delivery (CI/CD)

The process of automating the source, build, test, staging, and production stages of the software release process. CI/CD is commonly described as a pipeline. CI/CD can help you automate processes, improve productivity, improve code quality, and deliver faster. For more information, see Benefits of continuous delivery. CD can also stand for *continuous deployment*. For more information, see Continuous Delivery vs. Continuous Deployment.

CV

See computer vision.

D

data at rest

Data that is stationary in your network, such as data that is in storage.

data classification

A process for identifying and categorizing the data in your network based on its criticality and sensitivity. It is a critical component of any cybersecurity risk management strategy because it helps you determine the appropriate protection and retention controls for the data. Data classification is a component of the security pillar in the AWS Well-Architected Framework. For more information, see Data classification.

data drift

A meaningful variation between the production data and the data that was used to train an ML model, or a meaningful change in the input data over time. Data drift can reduce the overall quality, accuracy, and fairness in ML model predictions.

data in transit

Data that is actively moving through your network, such as between network resources.

data mesh

An architectural framework that provides distributed, decentralized data ownership with centralized management and governance.

data minimization

The principle of collecting and processing only the data that is strictly necessary. Practicing data minimization in the AWS Cloud can reduce privacy risks, costs, and your analytics carbon footprint.

data perimeter

A set of preventive guardrails in your AWS environment that help make sure that only trusted identities are accessing trusted resources from expected networks. For more information, see Building a data perimeter on AWS.

data preprocessing

To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

data provenance

The process of tracking the origin and history of data throughout its lifecycle, such as how the data was generated, transmitted, and stored.

data subject

An individual whose data is being collected and processed.

data warehouse

A data management system that supports business intelligence, such as analytics. Data warehouses commonly contain large amounts of historical data, and they are typically used for queries and analysis.

database definition language (DDL)

Statements or commands for creating or modifying the structure of tables and objects in a database.

database manipulation language (DML)

Statements or commands for modifying (inserting, updating, and deleting) information in a database.

DDL

See database definition language.

deep ensemble

To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

deep learning

An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

defense-in-depth

An information security approach in which a series of security mechanisms and controls are thoughtfully layered throughout a computer network to protect the confidentiality, integrity, and availability of the network and the data within. When you adopt this strategy on AWS, you add multiple controls at different layers of the AWS Organizations structure to help secure resources. For example, a defense-in-depth approach might combine multi-factor authentication, network segmentation, and encryption.

delegated administrator

In AWS Organizations, a compatible service can register an AWS member account to administer the organization's accounts and manage permissions for that service. This account is called the *delegated administrator* for that service. For more information and a list of compatible services, see Services that work with AWS Organizations in the AWS Organizations documentation.

deployment

The process of making an application, new features, or code fixes available in the target environment. Deployment involves implementing changes in a code base and then building and running that code base in the application's environments.

development environment

See environment.

detective control

A security control that is designed to detect, log, and alert after an event has occurred. These controls are a second line of defense, alerting you to security events that bypassed the preventative controls in place. For more information, see Detective controls in *Implementing security controls on AWS*.

development value stream mapping (DVSM)

A process used to identify and prioritize constraints that adversely affect speed and quality in a software development lifecycle. DVSM extends the value stream mapping process originally designed for lean manufacturing practices. It focuses on the steps and teams required to create and move value through the software development process.

digital twin

A virtual representation of a real-world system, such as a building, factory, industrial equipment, or production line. Digital twins support predictive maintenance, remote monitoring, and production optimization.

dimension table

In a star schema, a smaller table that contains data attributes about quantitative data in a fact table. Dimension table attributes are typically text fields or discrete numbers that behave like text. These attributes are commonly used for query constraining, filtering, and result set labeling.

disaster

An event that prevents a workload or system from fulfilling its business objectives in its primary deployed location. These events can be natural disasters, technical failures, or the result of human actions, such as unintentional misconfiguration or a malware attack.

disaster recovery (DR)

The strategy and process you use to minimize downtime and data loss caused by a disaster. For more information, see Disaster Recovery of Workloads on AWS: Recovery in the Cloud in the AWS Well-Architected Framework.

DML

See database manipulation language.

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

DR

See disaster recovery.

drift detection

Tracking deviations from a baselined configuration. For example, you can use AWS CloudFormation to detect drift in system resources, or you can use AWS Control Tower to detect changes in your landing zone that might affect compliance with governance requirements.

DVSM

See development value stream mapping.

E

EDA

See exploratory data analysis.

edge computing

The technology that increases the computing power for smart devices at the edges of an IoT network. When compared with cloud computing, edge computing can reduce communication latency and improve response time.

encryption

A computing process that transforms plaintext data, which is human-readable, into ciphertext.

encryption key

A cryptographic string of randomized bits that is generated by an encryption algorithm. Keys can vary in length, and each key is designed to be unpredictable and unique.

endianness

The order in which bytes are stored in computer memory. Big-endian systems store the most significant byte first. Little-endian systems store the least significant byte first.

endpoint

See service endpoint.

endpoint service

A service that you can host in a virtual private cloud (VPC) to share with other users. You can create an endpoint service with AWS PrivateLink and grant permissions to other AWS accounts or to AWS Identity and Access Management (IAM) principals. These accounts or principals can connect to your endpoint service privately by creating interface VPC endpoints. For more information, see Create an endpoint service in the Amazon Virtual Private Cloud (Amazon VPC) documentation.

enterprise resource planning (ERP)

A system that automates and manages key business processes (such as accounting, MES, and project management) for an enterprise.

envelope encryption

The process of encrypting an encryption key with another encryption key. For more information, see Envelope encryption in the AWS Key Management Service (AWS KMS) documentation.

environment

An instance of a running application. The following are common types of environments in cloud computing:

- development environment – An instance of a running application that is available only to the core team responsible for maintaining the application. Development environments are used to test changes before promoting them to upper environments. This type of environment is sometimes referred to as a *test environment*.
- lower environments – All development environments for an application, such as those used for initial builds and tests.
- production environment – An instance of a running application that end users can access. In a CI/CD pipeline, the production environment is the last deployment environment.
- upper environments – All environments that can be accessed by users other than the core development team. This can include a production environment, preproduction environments, and environments for user acceptance testing.

epic

In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the program implementation guide.

ERP

See enterprise resource planning.

exploratory data analysis (EDA)

The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies, and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

F

fact table

The central table in a star schema. It stores quantitative data about business operations. Typically, a fact table contains two types of columns: those that contain measures and those that contain a foreign key to a dimension table.

fail fast

A philosophy that uses frequent and incremental testing to reduce the development lifecycle. It is a critical part of an agile approach.

fault isolation boundary

In the AWS Cloud, a boundary such as an Availability Zone, AWS Region, control plane, or data plane that limits the effect of a failure and helps improve the resilience of workloads. For more information, see AWS Fault Isolation Boundaries.

feature branch

See branch.

features

The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

feature importance

How significant a feature is for a model's predictions. This is usually expressed as a numerical score that can be calculated through various techniques, such as Shapley Additive Explanations (SHAP) and integrated gradients. For more information, see Machine learning model interpretability with :AWS.

feature transformation

To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML model to benefit from the data. For example, if you break down the "2021-05-27 00:15:37" date into "2021", "May", "Thu", and "15", you can help the learning algorithm learn nuanced patterns associated with different data components.

FGAC

See fine-grained access control.

fine-grained access control (FGAC)

The use of multiple conditions to allow or deny an access request.

flash-cut migration

A database migration method that uses continuous data replication through change data capture to migrate data in the shortest time possible, instead of using a phased approach. The objective is to keep downtime to a minimum.

G

geo blocking

See geographic restrictions.

geographic restrictions (geo blocking)

In Amazon CloudFront, an option to prevent users in specific countries from accessing content distributions. You can use an allow list or block list to specify approved and banned countries. For more information, see Restricting the geographic distribution of your content in the CloudFront documentation.

Gitflow workflow

An approach in which lower and upper environments use different branches in a source code repository. The Gitflow workflow is considered legacy, and the trunk-based workflow is the modern, preferred approach.

greenfield strategy

The absence of existing infrastructure in a new environment. When adopting a greenfield strategy for a system architecture, you can select all new technologies without the restriction of compatibility with existing infrastructure, also known as brownfield. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

guardrail

A high-level rule that helps govern resources, policies, and compliance across organizational units (OUs). *Preventive guardrails* enforce policies to ensure alignment to compliance standards. They are implemented by using service control policies and IAM permissions boundaries. *Detective guardrails* detect policy violations and compliance issues, and generate alerts

for remediation. They are implemented by using AWS Config, AWS Security Hub, Amazon GuardDuty, AWS Trusted Advisor, Amazon Inspector, and custom AWS Lambda checks.

H

HA

See high availability.

heterogeneous database migration

Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. AWS provides AWS SCT that helps with schema conversions.

high availability (HA)

The ability of a workload to operate continuously, without intervention, in the event of challenges or disasters. HA systems are designed to automatically fail over, consistently deliver high-quality performance, and handle different loads and failures with minimal performance impact.

historian modernization

An approach used to modernize and upgrade operational technology (OT) systems to better serve the needs of the manufacturing industry. A *historian* is a type of database that is used to collect and store data from various sources in a factory.

homogeneous database migration

Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema.

hot data

Data that is frequently accessed, such as real-time data or recent translational data. This data typically requires a high-performance storage tier or class to provide fast query responses.

hotfix

An urgent fix for a critical issue in a production environment. Due to its urgency, a hotfix is usually made outside of the typical DevOps release workflow.

hypercare period

Immediately following cutover, the period of time when a migration team manages and monitors the migrated applications in the cloud in order to address any issues. Typically, this period is 1–4 days in length. At the end of the hypercare period, the migration team typically transfers responsibility for the applications to the cloud operations team.

I

IaC

See infrastructure as code.

identity-based policy

A policy attached to one or more IAM principals that defines their permissions within the AWS Cloud environment.

idle application

An application that has an average CPU and memory usage between 5 and 20 percent over a period of 90 days. In a migration project, it is common to retire these applications or retain them on premises.

IIoT

See industrial Internet of Things.

immutable infrastructure

A model that deploys new infrastructure for production workloads instead of updating, patching, or modifying the existing infrastructure. Immutable infrastructures are inherently more consistent, reliable, and predictable than mutable infrastructure. For more information, see the Deploy using immutable infrastructure best practice in the AWS Well-Architected Framework.

inbound (ingress) VPC

In an AWS multi-account architecture, a VPC that accepts, inspects, and routes network connections from outside an application. The AWS Security Reference Architecture recommends

setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

incremental migration

A cutover strategy in which you migrate your application in small parts instead of performing a single, full cutover. For example, you might move only a few microservices or users to the new system initially. After you verify that everything is working properly, you can incrementally move additional microservices or users until you can decommission your legacy system. This strategy reduces the risks associated with large migrations.

Industry 4.0

A term that was introduced by Klaus Schwab in 2016 to refer to the modernization of manufacturing processes through advances in connectivity, real-time data, automation, analytics, and AI/ML.

infrastructure

All of the resources and assets contained within an application's environment.

infrastructure as code (IaC)

The process of provisioning and managing an application's infrastructure through a set of configuration files. IaC is designed to help you centralize infrastructure management, standardize resources, and scale quickly so that new environments are repeatable, reliable, and consistent.

industrial Internet of Things (IIoT)

The use of internet-connected sensors and devices in the industrial sectors, such as manufacturing, energy, automotive, healthcare, life sciences, and agriculture. For more information, see Building an industrial Internet of Things (IIoT) digital transformation strategy.

inspection VPC

In an AWS multi-account architecture, a centralized VPC that manages inspections of network traffic between VPCs (in the same or different AWS Regions), the internet, and on-premises networks. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

Internet of Things (IoT)

: The network of connected physical objects with embedded sensors or processors that communicate with other devices and systems through the internet or over a local communication network. For more information, see What is IoT?

interpretability

: A characteristic of a machine learning model that describes the degree to which a human can understand how the model's predictions depend on its inputs. For more information, see Machine learning model interpretability with AWS.

IoT

: See Internet of Things.

IT information library (ITIL)

: A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM.

IT service management (ITSM)

: Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the operations integration guide.

ITIL

: See IT information library.

ITSM

: See IT service management.

L

label-based access control (LBAC)

: An implementation of mandatory access control (MAC) where the users and the data itself are each explicitly assigned a security label value. The intersection between the user security label and data security label determines which rows and columns can be seen by the user.

landing zone

A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

large migration

A migration of 300 or more servers.

LBAC

See label-based access control.

least privilege

The security best practice of granting the minimum permissions required to perform a task. For more information, see Apply least-privilege permissions in the IAM documentation.

lift and shift

See 7 Rs.

little-endian system

A system that stores the least significant byte first. See also endianness.

lower environments

See environment.

M

machine learning (ML)

A type of artificial intelligence that uses algorithms and techniques for pattern recognition and learning. ML analyzes and learns from recorded data, such as Internet of Things (IoT) data, to generate a statistical model based on patterns. For more information, see Machine Learning.

main branch

See branch.

malware

Software that is designed to compromise computer security or privacy. Malware might disrupt computer systems, leak sensitive information, or gain unauthorized access. Examples of malware include viruses, worms, ransomware, Trojan horses, spyware, and keyloggers.

managed services

AWS services for which AWS operates the infrastructure layer, the operating system, and platforms, and you access the endpoints to store and retrieve data. Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB are examples of managed services. These are also known as *abstracted services*.

manufacturing execution system (MES)

A software system for tracking, monitoring, documenting, and controlling production processes that convert raw materials to finished products on the shop floor.

MAP

See Migration Acceleration Program.

mechanism

A complete process in which you create a tool, drive adoption of the tool, and then inspect the results in order to make adjustments. A mechanism is a cycle that reinforces and improves itself as it operates. For more information, see Building mechanisms in the AWS Well-Architected Framework.

member account

All AWS accounts other than the management account that are part of an organization in AWS Organizations. An account can be a member of only one organization at a time.

MES

See manufacturing execution system.

Message Queuing Telemetry Transport (MQTT)

A lightweight, machine-to-machine (M2M) communication protocol, based on the publish/subscribe pattern, for resource-constrained IoT devices.

microservice

A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include

microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.

microservices architecture

An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed, and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.

Migration Acceleration Program (MAP)

An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios.

migration at scale

The process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a *migration factory* of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the AWS migration strategy.

migration factory

Cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners, migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the Cloud Migration Factory guide in this content set.

migration metadata

The information about the application and server that is needed to complete the migration. Each migration pattern requires a different set of migration metadata. Examples of migration metadata include the target subnet, security group, and AWS account.

migration pattern

A repeatable migration task that details the migration strategy, the migration destination, and the migration application or service used. Example: Rehost migration to Amazon EC2 with AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The MPA tool (requires login) is available free of charge to all AWS consultants and APN Partner consultants.

Migration Readiness Assessment (MRA)

The process of gaining insights about an organization's cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.

migration strategy

The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs entry in this glossary and see Mobilize your organization to accelerate large-scale migrations.

ML

See machine learning.

modernization

Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment

An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and

milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.

monolithic applications (monoliths)

Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can use a microservices architecture. For more information, see Decomposing monoliths into microservices.

MPA

See Migration Portfolio Assessment.

MQTT

See Message Queuing Telemetry Transport.

multiclass classification

A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask "Is this product a book, car, or phone?" or "Which product category is most interesting to this customer?"

mutable infrastructure

A model that updates and modifies the existing infrastructure for production workloads. For improved consistency, reliability, and predictability, the AWS Well-Architected Framework recommends the use of immutable infrastructure as a best practice.

O

OAC

See origin access control.

OAI

See origin access identity.

OCM

See organizational change management.

offline migration

A migration method in which the source workload is taken down during the migration process. This method involves extended downtime and is typically used for small, non-critical workloads.

OI

See operations integration.

OLA

See operational-level agreement.

online migration

A migration method in which the source workload is copied to the target system without being taken offline. Applications that are connected to the workload can continue to function during the migration. This method involves zero to minimal downtime and is typically used for critical production workloads.

OPC-UA

See Open Process Communications - Unified Architecture.

Open Process Communications - Unified Architecture (OPC-UA)

A machine-to-machine (M2M) communication protocol for industrial automation. OPC-UA provides an interoperability standard with data encryption, authentication, and authorization schemes.

operational-level agreement (OLA)

An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).

operational readiness review (ORR)

A checklist of questions and associated best practices that help you understand, evaluate, prevent, or reduce the scope of incidents and possible failures. For more information, see Operational Readiness Reviews (ORR) in the AWS Well-Architected Framework.

operational technology (OT)

Hardware and software systems that work with the physical environment to control industrial operations, equipment, and infrastructure. In manufacturing, the integration of OT and information technology (IT) systems is a key focus for Industry 4.0 transformations.

operations integration (OI)

The process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.

organization trail

A trail that's created by AWS CloudTrail that logs all events for all AWS accounts in an organization in AWS Organizations. This trail is created in each AWS account that's part of the organization and tracks the activity in each account. For more information, see Creating a trail for an organization in the CloudTrail documentation.

organizational change management (OCM)

A framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called *people acceleration*, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

origin access control (OAC)

In CloudFront, an enhanced option for restricting access to secure your Amazon Simple Storage Service (Amazon S3) content. OAC supports all S3 buckets in all AWS Regions, server-side encryption with AWS KMS (SSE-KMS), and dynamic `PUT` and `DELETE` requests to the S3 bucket.

origin access identity (OAI)

In CloudFront, an option for restricting access to secure your Amazon S3 content. When you use OAI, CloudFront creates a principal that Amazon S3 can authenticate with. Authenticated principals can access content in an S3 bucket only through a specific CloudFront distribution. See also OAC, which provides more granular and enhanced access control.

ORR

See operational readiness review.

OT

See operational technology.

outbound (egress) VPC

In an AWS multi-account architecture, a VPC that handles network connections that are initiated from within an application. The AWS Security Reference Architecture recommends

setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

P

permissions boundary

An IAM management policy that is attached to IAM principals to set the maximum permissions that the user or role can have. For more information, see Permissions boundaries in the IAM documentation.

personally identifiable information (PII)

Information that, when viewed directly or paired with other related data, can be used to reasonably infer the identity of an individual. Examples of PII include names, addresses, and contact information.

PII

See personally identifiable information.

playbook

A set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

PLC

See programmable logic controller.

PLM

See product lifecycle management.

policy

An object that can define permissions (see identity-based policy), specify access conditions (see resource-based policy), or define the maximum permissions for all accounts in an organization in AWS Organizations (see service control policy).

polyglot persistence

Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements. For more information, see Enabling data persistence in microservices.

portfolio assessment

A process of discovering, analyzing, and prioritizing the application portfolio in order to plan the migration. For more information, see Evaluating migration readiness.

predicate

A query condition that returns `true` or `false`, commonly located in a `WHERE` clause.

predicate pushdown

A database query optimization technique that filters the data in the query before transfer. This reduces the amount of data that must be retrieved and processed from the relational database, and it improves query performance.

preventative control

A security control that is designed to prevent an event from occurring. These controls are a first line of defense to help prevent unauthorized access or unwanted changes to your network. For more information, see Preventative controls in *Implementing security controls on AWS*.

principal

An entity in AWS that can perform actions and access resources. This entity is typically a root user for an AWS account, an IAM role, or a user. For more information, see *Principal* in Roles terms and concepts in the IAM documentation.

Privacy by Design

An approach in system engineering that takes privacy into account throughout the whole engineering process.

private hosted zones

A container that holds information about how you want Amazon Route 53 to respond to DNS queries for a domain and its subdomains within one or more VPCs. For more information, see Working with private hosted zones in the Route 53 documentation.

proactive control

A security control designed to prevent the deployment of noncompliant resources. These controls scan resources before they are provisioned. If the resource is not compliant with the control, then it isn't provisioned. For more information, see the Controls reference guide in the AWS Control Tower documentation and see Proactive controls in *Implementing security controls on AWS*.

product lifecycle management (PLM)

The management of data and processes for a product throughout its entire lifecycle, from design, development, and launch, through growth and maturity, to decline and removal.

production environment

See environment.

programmable logic controller (PLC)

In manufacturing, a highly reliable, adaptable computer that monitors machines and automates manufacturing processes.

pseudonymization

The process of replacing personal identifiers in a dataset with placeholder values. Pseudonymization can help protect personal privacy. Pseudonymized data is still considered to be personal data.

publish/subscribe (pub/sub)

A pattern that enables asynchronous communications among microservices to improve scalability and responsiveness. For example, in a microservices-based MES, a microservice can publish event messages to a channel that other microservices can subscribe to. The system can add new microservices without changing the publishing service.

Q

query plan

A series of steps, like instructions, that are used to access the data in a SQL relational database system.

query plan regression

When a database service optimizer chooses a less optimal plan than it did before a given change to the database environment. This can be caused by changes to statistics, constraints, environment settings, query parameter bindings, and updates to the database engine.

R

RACI matrix

See responsible, accountable, consulted, informed (RACI).

ransomware

A malicious software that is designed to block access to a computer system or data until a payment is made.

RASCI matrix

See responsible, accountable, consulted, informed (RACI).

RCAC

See row and column access control.

read replica

A copy of a database that's used for read-only purposes. You can route queries to the read replica to reduce the load on your primary database.

re-architect

See 7 Rs.

recovery point objective (RPO)

The maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

recovery time objective (RTO)

The maximum acceptable delay between the interruption of service and restoration of service.

refactor

See 7 Rs.

Region

A collection of AWS resources in a geographic area. Each AWS Region is isolated and independent of the others to provide fault tolerance, stability, and resilience. For more information, see Specify which AWS Regions your account can use.

regression

An ML technique that predicts a numeric value. For example, to solve the problem of "What price will this house sell for?" an ML model could use a linear regression model to predict a house's sale price based on known facts about the house (for example, the square footage).

rehost

See 7 Rs.

release

In a deployment process, the act of promoting changes to a production environment.

relocate

See 7 Rs.

replatform

See 7 Rs.

repurchase

See 7 Rs.

resiliency

An application's ability to resist or recover from disruptions. High availability and disaster recovery are common considerations when planning for resiliency in the AWS Cloud. For more information, see AWS Cloud Resilience.

resource-based policy

A policy attached to a resource, such as an Amazon S3 bucket, an endpoint, or an encryption key. This type of policy specifies which principals are allowed access, supported actions, and any other conditions that must be met.

responsible, accountable, consulted, informed (RACI) matrix

A matrix that defines the roles and responsibilities for all parties involved in migration activities and cloud operations. The matrix name is derived from the responsibility types defined in the

matrix: responsible (R), accountable (A), consulted (C), and informed (I). The support (S) type is optional. If you include support, the matrix is called a *RASCI matrix*, and if you exclude it, it's called a *RACI matrix*.

responsive control

A security control that is designed to drive remediation of adverse events or deviations from your security baseline. For more information, see Responsive controls in *Implementing security controls on AWS*.

retain

See 7 Rs.

retire

See 7 Rs.

rotation

The process of periodically updating a secret to make it more difficult for an attacker to access the credentials.

row and column access control (RCAC)

The use of basic, flexible SQL expressions that have defined access rules. RCAC consists of row permissions and column masks.

RPO

See recovery point objective.

RTO

See recovery time objective.

runbook

A set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

S

SAML 2.0

An open standard that many identity providers (IdPs) use. This feature enables federated single sign-on (SSO), so users can log into the AWS Management Console or call the AWS API

operations without you having to create user in IAM for everyone in your organization. For more information about SAML 2.0-based federation, see About SAML 2.0-based federation in the IAM documentation.

SCADA

See supervisory control and data acquisition.

SCP

See service control policy.

secret

In AWS Secrets Manager, confidential or restricted information, such as a password or user credentials, that you store in encrypted form. It consists of the secret value and its metadata. The secret value can be binary, a single string, or multiple strings. For more information, see What's in a Secrets Manager secret? in the Secrets Manager documentation.

security control

A technical or administrative guardrail that prevents, detects, or reduces the ability of a threat actor to exploit a security vulnerability. There are four primary types of security controls: preventative, detective, responsive, and proactive.

security hardening

The process of reducing the attack surface to make it more resistant to attacks. This can include actions such as removing resources that are no longer needed, implementing the security best practice of granting least privilege, or deactivating unnecessary features in configuration files.

security information and event management (SIEM) system

Tools and services that combine security information management (SIM) and security event management (SEM) systems. A SIEM system collects, monitors, and analyzes data from servers, networks, devices, and other sources to detect threats and security breaches, and to generate alerts.

security response automation

A predefined and programmed action that is designed to automatically respond to or remediate a security event. These automations serve as detective or responsive security controls that help you implement AWS security best practices. Examples of automated response actions include modifying a VPC security group, patching an Amazon EC2 instance, or rotating credentials.

server-side encryption

Encryption of data at its destination, by the AWS service that receives it.

service control policy (SCP)

A policy that provides centralized control over permissions for all accounts in an organization in AWS Organizations. SCPs define guardrails or set limits on actions that an administrator can delegate to users or roles. You can use SCPs as allow lists or deny lists, to specify which services or actions are permitted or prohibited. For more information, see Service control policies in the AWS Organizations documentation.

service endpoint

The URL of the entry point for an AWS service. You can use the endpoint to connect programmatically to the target service. For more information, see AWS service endpoints in *AWS General Reference*.

service-level agreement (SLA)

An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

service-level indicator (SLI)

A measurement of a performance aspect of a service, such as its error rate, availability, or throughput.

service-level objective (SLO)

A target metric that represents the health of a service, as measured by a service-level indicator.

shared responsibility model

A model describing the responsibility you share with AWS for cloud security and compliance. AWS is responsible for security *of* the cloud, whereas you are responsible for security *in* the cloud. For more information, see Shared responsibility model.

SIEM

See security information and event management system.

single point of failure (SPOF)

A failure in a single, critical component of an application that can disrupt the system.

SLA

See service-level agreement.

SLI

See service-level indicator.

SLO

See service-level objective.

split-and-seed model

A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your organization's capabilities and services, improves developer productivity, and supports rapid innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

SPOF

See single point of failure.

star schema

A database organizational structure that uses one large fact table to store transactional or measured data and uses one or more smaller dimensional tables to store data attributes. This structure is designed for use in a data warehouse or for business intelligence purposes.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

subnet

A range of IP addresses in your VPC. A subnet must reside in a single Availability Zone.

supervisory control and data acquisition (SCADA)

In manufacturing, a system that uses hardware and software to monitor physical assets and production operations.

symmetric encryption

An encryption algorithm that uses the same key to encrypt and decrypt the data.

synthetic testing

Testing a system in a way that simulates user interactions to detect potential issues or to monitor performance. You can use Amazon CloudWatch Synthetics to create these tests.

T

tags

Key-value pairs that act as metadata for organizing your AWS resources. Tags can help you manage, identify, organize, search for, and filter resources. For more information, see Tagging your AWS resources.

target variable

The value that you are trying to predict in supervised ML. This is also referred to as an *outcome variable*. For example, in a manufacturing setting the target variable could be a product defect.

task list

A tool that is used to track progress through a runbook. A task list contains an overview of the runbook and a list of general tasks to be completed. For each general task, it includes the estimated amount of time required, the owner, and the progress.

test environment

See environment.

training

To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don't know the target.

transit gateway

A network transit hub that you can use to interconnect your VPCs and on-premises networks. For more information, see What is a transit gateway in the AWS Transit Gateway documentation.

trunk-based workflow

An approach in which developers build and test features locally in a feature branch and then merge those changes into the main branch. The main branch is then built to the development, preproduction, and production environments, sequentially.

trusted access

Granting permissions to a service that you specify to perform tasks in your organization in AWS Organizations and in its accounts on your behalf. The trusted service creates a service-linked role in each account, when that role is needed, to perform management tasks for you. For more information, see Using AWS Organizations with other AWS services in the AWS Organizations documentation.

tuning

To change aspects of your training process to improve the ML model's accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

two-pizza team

A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development.

U

uncertainty

A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: *Epistemic uncertainty* is caused by limited, incomplete data, whereas *aleatoric uncertainty* is caused by the noise and randomness inherent in the data. For more information, see the Quantifying uncertainty in deep learning systems guide.

undifferentiated tasks

Also known as *heavy lifting*, work that is necessary to create and operate an application but that doesn't provide direct value to the end user or provide competitive advantage. Examples of undifferentiated tasks include procurement, maintenance, and capacity planning.

upper environments

See environment.

V

vacuuming

A database maintenance operation that involves cleaning up after incremental updates to reclaim storage and improve performance.

version control

Processes and tools that track changes, such as changes to source code in a repository.

VPC peering

A connection between two VPCs that allows you to route traffic by using private IP addresses. For more information, see What is VPC peering in the Amazon VPC documentation.

vulnerability

A software or hardware flaw that compromises the security of the system.

W

warm cache

A buffer cache that contains current, relevant data that is frequently accessed. The database instance can read from the buffer cache, which is faster than reading from the main memory or disk.

warm data

Data that is infrequently accessed. When querying this kind of data, moderately slow queries are typically acceptable.

window function

A SQL function that performs a calculation on a group of rows that relate in some way to the current record. Window functions are useful for processing tasks, such as calculating a moving average or accessing the value of rows based on the relative position of the current row.

workload

A collection of resources and code that delivers business value, such as a customer-facing application or backend process.

workstream

Functional groups in a migration project that are responsible for a specific set of tasks. Each workstream is independent but supports the other workstreams in the project. For example, the portfolio workstream is responsible for prioritizing applications, wave planning, and collecting migration metadata. The portfolio workstream delivers these assets to the migration workstream, which then migrates the servers and applications.

WORM

See write once, read many.

WQF

See AWS Workload Qualification Framework.

write once, read many (WORM)

A storage model that writes data a single time and prevents the data from being deleted or modified. Authorized users can read the data as many times as needed, but they cannot change it. This data storage infrastructure is considered immutable.

Z

zero-day exploit

An attack, typically malware, that takes advantage of a zero-day vulnerability.

zero-day vulnerability

An unmitigated flaw or vulnerability in a production system. Threat actors can use this type of vulnerability to attack the system. Developers frequently become aware of the vulnerability as a result of the attack.

zombie application

An application that has an average CPU and memory usage below 5 percent. In a migration project, it is common to retire these applications.