Getting started with Terraform: Guidance for AWS CDK and AWS CloudFormation experts

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Getting started with Terraform: Guidance for AWS CDK and AWS CloudFormation experts

Table of Contents

Getting started with Terraform: Guidance for AWS CDK and AWS CloudFormation experts

Steven Guggenheimer, Amazon Web Services (AWS)

March 2024 (document history)

If your experience with provisioning cloud resources exclusively lies within the realm of AWS, you might have limited experience with infrastructure as code (IaC) tools beyond the AWS Cloud Development Kit (AWS CDK) and AWS CloudFormation. In fact, similar tools, such as Hashicorp Terraform, might be completely unfamiliar to you. However, the deeper you get into your cloud journey, the more inevitable it becomes that you'll encounter Terraform. It will be decidedly to your advantage to be familiar with its core concepts.

While Terraform, the AWS CDK, and CloudFormation achieve similar goals and share many core concepts, there are quite a few differences. You might not be prepared for these differences if you're approaching Terraform for the first time. After all, AWS CDK and CloudFormation stacks are all based within AWS accounts, so in that way, they have a direct relationship with most of the resources that they maintain. Terraform is not based within any single cloud provider's environment. This gives it the flexibility to support various different providers, but it must maintain resources from what amounts to a remote location.

This guide helps demystify the core concepts behind Terraform to help you handle any IaC challenge that comes your way. It focuses on how Terraform uses concepts, such as providers, modules, and state files, to provision resources. It also contrasts Terraform concepts with how the AWS CDK and CloudFormation perform similar operations.

Note

The AWS CDK helps developers deploy CloudFormation stacks by using programmatic coding languages. After you run `cdk synth`, your code is converted into CloudFormation templates. From that point forward, the process is identical between the AWS CDK and CloudFormation. For the sake of brevity, this guide usually refers to the AWS IaC process in CloudFormation terms, but the comparisons are just as apt for the AWS CDK.

CloudFormation and Terraform terminology

When comparing Terraform with the AWS CDK and CloudFormation, reconciling the IaC core concepts can be difficult because of the inconsistent terminology used to describe them. The following are these terms and how this guide will refer to them:

- **Stack** – A *stack* is IaC that is deployed into a CI/CD pipeline and trackable as a single unit. Although this term is common in CloudFormation, Terraform does not really use this term. A Terraform stack is a deployed root module with all of its child modules. However, in order to avoid confusion with the term *module*, this guide uses the term *stack* to describe a single deployment for both tools.
- **State** - The *state* is all currently tracked resources and their current configurations within an IaC deployment stack. As described in the Understanding Terraform states and backends section, Terraform uses the term *state* more than CloudFormation. This is because maintaining state is more visible in Terraform, but tracking and updating the state is equally important for CloudFormation.
- **IaC file –** An *IaC file* is a single file that contains infrastructure as code (IaC) language. CloudFormation refers to a single CloudFormation file as a *template*. However templates and template files in Terraform are something completely different. The equivalent to a CloudFormation template in Terraform is called a *configuration file*. To minimize confusion in this guide, the term *file* or *IaC file* is used to refer to both CloudFormation templates and Terraform configuration files.

The following table compares the terminology used for CloudFormation and Terraform. The intent of this table is to show similarities. These are not one-to-one comparisons. Each concept differs at least slightly between CloudFormation and Terraform. Concepts are explained in depth in relevant sections of this guide.

CloudFormation term	Terraform term	Section of this guide
CDK interfaces (such as *IBucket*)	Data source	Understanding Terraform data sources
Change set	Plan	Understanding Terraform modules

CloudFormation term	Terraform term	Section of this guide
Condition functions	Conditional expressions	Understanding Terraform functions, expressions, and meta-arguments
`DependsOn` attribute	`depends_on` meta-argument	Understanding Terraform functions, expressions, and meta-arguments
Intrinsic functions	Functions	Understanding Terraform functions, expressions, and meta-arguments
Modules	Modules	Understanding Terraform modules
Outputs	Output values	Understanding Terraform variables, local values, and outputs
Parameters	Variables	Understanding Terraform variables, local values, and outputs
Registry	Providers	Understanding Terraform providers
Template	Configuration file	All

Understanding Terraform resources

The primary reason for the existence of both AWS CloudFormation and Terraform is the creation and maintenance of cloud resources. But what exactly is a cloud resource? And are CloudFormation resources and Terraform resources the same thing? The answer is… yes and no. To illustrate this, this guide provides an example of using CloudFormation and then Terraform to create an Amazon Simple Storage Service (Amazon S3) bucket.

The following CloudFormation code example creates a sample Amazon S3 bucket.

```
{
    "myS3Bucket": {
        "Type": "AWS::S3::Bucket",
        "Properties": {
            "BucketName": "my-s3-bucket",
            "BucketEncryption": {
                "ServerSideEncryptionConfiguration": [
                    {
                        "ServerSideEncryptionByDefault": {
                            "SSEAlgorithm": "AES256"
                        }
                    }
                ]
            },
            "PublicAccessBlockConfiguration": {
                "BlockPublicAcls": true,
                "BlockPublicPolicy": true,
                "IgnorePublicAcls": true,
                "RestrictPublicBuckets": true
            },
            "VersioningConfiguration": {
                "Status": "Enabled"
            }
        }
    }
}
```

The following Terraform code example creates an identical Amazon S3 bucket.

```
resource "aws_s3_bucket" "myS3Bucket" {
  bucket = "my-s3-bucket"
}

resource "aws_s3_bucket_server_side_encryption_configuration" "bucketencryption" {
  bucket = aws_s3_bucket.myS3Bucket.id
  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}

resource "aws_s3_bucket_public_access_block" "publicaccess" {
  bucket                  = aws_s3_bucket.myS3Bucket.id
  block_public_acls       = true
  block_public_policy     = true
  ignore_public_acls      = true
  restrict_public_buckets = true
}

resource "aws_s3_bucket_versioning" "versioning" {
  bucket = aws_s3_bucket.myS3Bucket.id
  versioning_configuration {
    status = "Enabled"
  }
}
```

For Terraform, a provider defines the resource, and then developers declare and configure those resources. Providers are a concept that this guide discusses in the next section. The Terraform example creates completely separate resources for several of the S3 bucket's settings. Creating separate resources for settings is not necessarily typical of how the Terraform AWS Provider treats AWS resources. However, this example shows an important distinction. While a CloudFormation resource is strictly defined by the CloudFormation resource specification, Terraform has no such requirement. In Terraform, the concept of a resource is a bit more nebulous.

Although the tools might differ regarding the exact guardrails that define what a single resource is, generally speaking, a *cloud resource* is any particular entity that exists in the cloud and that can be created, updated, or deleted. So regardless of how many resources are involved, the two previous examples both create the exact same thing with the exact same settings within an AWS account.

Understanding Terraform providers

In Terraform, a *provider* is a plugin that interacts with cloud providers, third-party tools, and other APIs. To use Terraform with AWS, you use the AWS Provider, which interacts with AWS resources.

If you've never used the AWS CloudFormation registry to incorporate third-party extensions into your deployment stacks, then Terraform providers might take some getting used to. Because CloudFormation is native to AWS, the provider of AWS resources is already there by default. Terraform, on the other hand, has no single default provider, so nothing can be assumed about the origins of a given resource. This means that the first thing that needs to be declared in a Terraform configuration file is exactly where the resources are going and how they're going to get there.

This distinction adds an extra layer of complexity to Terraform that doesn't exist with CloudFormation. However, that complexity provides increased flexibility. You can declare multiple providers within a single Terraform module, and then the underlying resources that are created can interact with each other as part of the same deployment layer.

This can be useful in numerous ways. Providers don't necessarily have to be for separate cloud providers. Providers can represent any source for cloud resources. For example, take Amazon Elastic Kubernetes Service (Amazon EKS). When you provision an Amazon EKS cluster, you might want to use Helm charts to manage third-party extensions and use Kubernetes itself to manage pod resources. Because AWS, Helm, and Kubernetes all have their own Terraform providers, you can provision and integrate these resources all at the same time and then pass values among them.

In the following code example for Terraform, the AWS Provider creates an Amazon EKS cluster, and then the resulting Kubernetes configuration information is passed along to both the Helm and Kubernetes providers.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 4.33.0"
    }

    helm = {
      source  = "hashicorp/helm"
      version = "2.12.1"
    }
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = "2.26.0"
    }
  }
  required_version = ">= 1.2.0"
}

provider "aws" {
  region = "us-west-2"
}

resource "aws_eks_cluster" "example_0" {
  name     = "example_0"
  role_arn = aws_iam_role.cluster_role.arn
  vpc_config {
    endpoint_private_access = true
    endpoint_public_access  = true
    subnet_ids              = var.subnet_ids
  }
}

locals {
  host        = aws_eks_cluster.example_0.endpoint
  certificate = base64decode(aws_eks_cluster.example_0.certificate_authority.data)
}

provider "helm" {
  kubernetes {
    host                   = local.host
    cluster_ca_certificate = local.certificate
    # exec allows for an authentication command to be run to obtain user
    # credentials rather than having them stored directly in the file
    exec {
      api_version = "client.authentication.k8s.io/v1beta1"
      args        = ["eks", "get-token", "--cluster-name",
 aws_eks_cluster.example_0.name]
      command     = "aws"
    }
  }
}
```

```
provider "kubernetes" {
  host                   = local.host
  cluster_ca_certificate = local.certificate
  exec {
    api_version = "client.authentication.k8s.io/v1beta1"
    args        = ["eks", "get-token", "--cluster-name",
 aws_eks_cluster.example_0.name]
    command     = "aws"
  }
}
```

There's a trade-off regarding providers when it comes to the two IaC tools. Terraform relies wholly on externally located provider packages, which are the engine that drives its deployments. CloudFormation internally supports all major AWS processes. With CloudFormation, you need to worry about third-party providers only if you want to incorporate a third-party extension. There are pros and cons to each approach. Which one is right for you is beyond the scope of this guide, but it's important to remember the difference when evaluating both tools.

Using Terraform aliases

In Terraform, you can pass custom configurations into each provider. So what if you want to use multiple provider configurations within the same module? In that case you'd have to use an alias. Aliases help you select which provider to use at a per-resource or per-module level. When you have more than one instance of the same provider, you use an alias to define the non-default instances. For example, your default provider instance might be a specific AWS Region, but you use aliases to define alternate regions.

The following Terraform example shows how to use an alias to provision buckets in different AWS Regions. The default Region for the provider is `us-west-2`, but you can use the east alias to provision resources in `us-east-2`.

```
provider "aws" {
  region = "us-west-2"
}

provider "aws" {
  alias  = "east"
  region = "us-east-2"
}
```

```
resource "aws_s3_bucket" "myWestS3Bucket" {
  bucket = "my-west-s3-bucket"
}

resource "aws_s3_bucket" "myEastS3Bucket" {
  provider = aws.east
  bucket   = "my-east-s3-bucket"
}
```

When you use an `alias` along with the `provider` meta-argument, as shown in the previous example, you can specify a different provider configuration for specific resources. Provisioning resources in multiple AWS Regions in a single stack is just the beginning. Aliasing providers is incredibly convenient in many ways.

For example, it's very common to provision multiple Kubernetes clusters at a time. Aliases can help you configure additional Helm and Kubernetes providers so that you can use these third-party tools differently for different Amazon EKS resources. The following Terraform code example illustrates how to use aliases to perform this task.

```
resource "aws_eks_cluster" "example_0" {
  name     = "example_0"
  role_arn = aws_iam_role.cluster_role.arn
  vpc_config {
    endpoint_private_access = true
    endpoint_public_access  = true
    subnet_ids              = var.subnet_ids[0]
  }
}

resource "aws_eks_cluster" "example_1" {
  name     = "example_1"
  role_arn = aws_iam_role.cluster_role.arn
  vpc_config {
    endpoint_private_access = true
    endpoint_public_access  = true
    subnet_ids              = var.subnet_ids[1]
  }
}

locals {
  host         = aws_eks_cluster.example_0.endpoint
  certificate  = base64decode(aws_eks_cluster.example_0.certificate_authority.data)
  host1        = aws_eks_cluster.example_1.endpoint
  certificate1 = base64decode(aws_eks_cluster.example_1.certificate_authority.data)
}

provider "helm" {
  kubernetes {
    host                   = local.host
    cluster_ca_certificate = local.certificate
    exec {
      api_version = "client.authentication.k8s.io/v1beta1"
      args        = ["eks", "get-token", "--cluster-name",
 aws_eks_cluster.example_0.name]
      command     = "aws"
    }
  }
}

provider "helm" {
  alias = "helm1"
  kubernetes {
    host                   = local.host1
    cluster_ca_certificate = local.certificate1
    exec {
      api_version = "client.authentication.k8s.io/v1beta1"
      args        = ["eks", "get-token", "--cluster-name",
 aws_eks_cluster.example_1.name]
      command     = "aws"
    }
  }
}

provider "kubernetes" {
  host                   = local.host
  cluster_ca_certificate = local.certificate
  exec {
    api_version = "client.authentication.k8s.io/v1beta1"
    args        = ["eks", "get-token", "--cluster-name",
 aws_eks_cluster.example_0.name]
    command     = "aws"
  }
}

provider "kubernetes" {
  alias                  = "kubernetes1"
  host                   = local.host1
  cluster_ca_certificate = local.certificate1
  exec {
    api_version = "client.authentication.k8s.io/v1beta1"
    args        = ["eks", "get-token", "--cluster-name",
 aws_eks_cluster.example_1.name]
    command     = "aws"
  }
}
```

Understanding Terraform modules

In the realm of infrastructure as code (IaC), a *module* is a self-contained block of code that is isolated and packaged together for reuse. The concept of modules is an inescapable aspect of Terraform development. For more information, see Modules in the Terraform documentation. AWS CloudFormation also supports modules. For more information, see Introducing AWS CloudFormation modules in the AWS Cloud Operations and Migrations Blog.

The major difference between modules in Terraform and CloudFormation is that CloudFormation modules are imported by using a special resource type (`AWS::CloudFormation::ModuleVersion`). In Terraform, every configuration has at least one module, known as the root module. Terraform resources that are in the **main.tf** file or files in a Terraform configuration file are considered to be in the root module. The root module can then call other modules for inclusion within the stack. The following example shows a root module provisioning an Amazon Elastic Kubernetes Service (Amazon EKS) cluster by using the open source eks module.

```
terraform {
  required_providers {
    helm = {
      source  = "hashicorp/helm"
      version = "2.12.1"
    }
  }
  required_version = ">= 1.2.0"
}

module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "20.2.1"
  vpc_id  = var.vpc_id
}

provider "helm" {
  kubernetes {
    host                   = module.eks.cluster_endpoint
    cluster_ca_certificate =
 base64decode(module.eks.cluster_certificate_authority_data)
  }
}
```

You might have noticed that the above configuration file does not include the AWS Provider. That’s because modules are self-contained and can include their own providers. Because Terraform providers are global, providers from a child module can be used in the root module. This is not true about all module values though. Other internal values within a module are scoped by default to that module only and need to be declared as outputs to be accessible in the root module. You can leverage open source modules to simplify resource creation within your stack. For example, the eks module does more than provision an EKS cluster—it provisions a fully functioning Kubernetes environment. Using it can save you from writing dozens of extra lines of code, provided that the eks module configuration suits your needs.

Calling modules

Two of the primary Terraform CLI commands that you run during Terraform deployment are terraform init and terraform apply. One of the default steps that the `terraform init` command performs is to locate all child modules and import them as dependencies into the `.terraform/modules` directory. During development, whenever you add a new externally sourced module, you must re-initialize before using the `apply` command. When you hear a reference to a Terraform *module*, it’s referring to the packages in this directory. Strictly speaking, the module that you declare in your code is the *calling module*, so in practice, the module keyword calls the actual module, which is stored as a dependency.

In this way, the calling module serves as a more succinct representative of the full module to be replaced when deployment takes place. You can leverage this idea by creating your own modules within your stacks to enforce logical separations of resources by using whatever criteria you’d like. Just remember that the end goal of doing this should be to reduce your stack complexity. Because sharing data between modules requires you to output that data from within the module, sometimes relying too heavily on modules can overly complicate things.

The root module

Because every Terraform configuration has at least one module, it can help to examine the module properties of the module you’ll be dealing with the most: the root module. Whenever you’re working on a Terraform project, the root module consists of all the `.tf` (or `.tf.json`) files in your top-level directory. When you run `terraform apply` in that top-level directory, Terraform attempts to run every `.tf` file it finds there. Any files in subdirectories are ignored unless they are called in one of these top-level configuration files.

This provides some flexibility in how you structure your code. It is also the reason why it's more accurate to refer to your Terraform deployment as a module than as a file because several files could be involved in a single process. There is a standard module structure that Terraform recommends for best practices. However, if you were to put any `.tf` file in your top-level directory, it would run along with the rest of the files. In fact, all top-level `.tf` files in a module are deployed when you run `terraform apply`. So which file does Terraform run first? The answer to that question is very important.

There's a series of steps that Terraform performs after initialization and before stack deployment. First, the existing configurations are analyzed, and then a dependency graph is created. The dependency graph determines what resources are called for and in what order they should be addressed. Resources that contain properties that are referenced in other resources, for example, would be handled before their dependent resources. Similarly, resources that explicitly declare dependence by using the `depends_on` parameter would be handled after the resources that they specify. When possible, Terraform can implement parallelism and handle non-dependent resources simultaneously. You can see the dependency graph before deploying by using the terraform graph command.

After the dependency graph is created, Terraform determines what needs to be done during the deployment. It compares the dependency graph with the most recent state file. The result of this process is called a *plan*, and it is very much like a CloudFormation change set. You can see the current plan by using the terraform plan command.

As a best practice, it's recommended to stay as close as possible to the standard module structure. In cases where your configuration files are becoming too long to efficiently manage and logical separations could simplify management, you can spread your code across several files. Keep in mind how the dependency graph and plan process works to make your stacks run as efficiently as possible.

Understanding Terraform states and backends

One of the most important concepts in infrastructure as code (IaC) is the concept of *state*. IaC services maintain state, which allows you to declare a resource in an IaC file without having it recreated each time you deploy. IaC files document the state of all resources at the end of a deployment so that it can then compare that state to the target state, as declared in the next deployment. So if the current state contains an Amazon Simple Storage Service (Amazon S3) bucket named `my-s3-bucket` and the incoming changes also contain that same bucket, the new process will apply any changes found to the existing bucket rather than trying to create an all new bucket.

The following table provides examples of the general IaC state process.

Current state	Target state	Action
No S3 bucket named `my-s3-bucket`	S3 bucket named `my-s3-bucket`	Create an S3 bucket named `my-s3-bucket`
`my-s3-bucket` with no bucket versioning configured	`my-s3-bucket` with no bucket versioning configured	No action
`my-s3-bucket` with no bucket versioning configured	`my-s3-bucket` with bucket versioning configured	Configure `my-s3-bucket` to have bucket versioning
`my-s3-bucket` with bucket versioning configured	No S3 bucket named `my-s3-bucket`	Attempt to delete `my-s3-bucket`

To understand the different ways in which AWS CloudFormation and Terraform track state, it's important to remember the first basic difference between the two tools: CloudFormation is hosted inside of the AWS Cloud, and Terraform is essentially remote. This fact allows CloudFormation to maintain state internally. You can go to the CloudFormation console and view the event history of a given stack, but the CloudFormation service itself enforces the state rules for you.

The three modes that CloudFormation operates under for a given resource are `Create`, `Update`, and `Delete`. The current mode is determined based on what happened in the last deployment, and it cannot be influenced otherwise. You can perhaps update CloudFormation resources manually in order to influence which mode is determined, but you can't pass a command to CloudFormation that says "For this resource, operate under `Create` mode."

Because Terraform is not hosted in the AWS Cloud, the process of maintaining state must be more configurable. For this reason, the Terraform state is maintained within an automatically generated state file. A Terraform developer has to deal with state much more directly than they would with CloudFormation. The important thing to remember is that tracking state is equally as important for both tools.

By default, the Terraform state file is stored locally at the top-level of the main directory that runs your Terraform stack. If you run the `terraform apply` command from your local development environment, you can see Terraform generate the **terraform.tfstate** file that it uses to maintain state in real time. For better or for worse, this gives you much more control over state in Terraform than you have in CloudFormation. While you should never update the state file directly, there are several Terraform CLI commands you can run that will update state between deployments. For example, terraform import allows you to add resources created outside of Terraform into your deployment stack. Conversely, you can remove a resource from state by running terraform state rm.

The fact that Terraform needs to store its state somewhere leads to another concept that doesn't apply to CloudFormation: the *backend*. A Terraform backend is the place where a Terraform stack stores its state file after deployment. This is also where it expects to find the state file when a new deployment begins. When you run your stack locally, as described above, you can keep a copy of the Terraform state in the top-level local directory. This is known as a *local backend*.

When developing for a continuous integration and continuous deployment (CI/CD) environment, the local state file is generally included in the **.gitignore** file to keep it out of version control. Then there's no local state file present within the pipeline. In order to work properly, that pipeline stage needs to find the correct state file somewhere. This is why Terraform configuration files often contain a backend block. The backend block indicates to the Terraform stack that it needs to look somewhere besides its own top-level directory to find the state file.

A Terraform backend can be located almost anywhere: an Amazon S3 bucket, an API endpoint, or even a remote Terraform workspace. The following is an example of a Terraform backend stored in an Amazon S3 bucket.

```
terraform {
  backend "s3" {
    bucket = "my-s3-bucket"
    key    = "state-file-folder"
    region = "us-east-1"
  }
}
```

In order to avoid storing sensitive information within Terraform configuration files, backends also support partial configurations. In the previous example, the credentials needed to access the bucket are not present in the configuration. Credentials can be obtained from environment variables or by using other means, such as AWS Secrets Manager. For more information, see Securing sensitive data by using AWS Secrets Manager and HashiCorp Terraform.

A common backend scenario is a local backend that is used in your local environment for testing purposes. The **terraform.tfstate** file is included in the **.gitignore** file so that it is not pushed to the remote repository. Then, each environment within the CI/CD pipeline would maintain its own backend. In this scenario, multiple developers might have access to this remote state, so you would want to protect the integrity of the state file. If multiple deployments are running and updating state at the same time, the state file could become corrupted. For this reason, in situations with non-local backends, the state file is typically locked during deployment.

Understanding Terraform data sources

It's very common for deployment stacks to rely on data from previously existing resources. Most IaC tools have a way of importing resources that were created by some other process. These imported resources are usually read only (although IAM roles are a notable exception) and are used to access data needed by resources within the stack. AWS CloudFormation allows for importing resources, but this idea can be better explained by looking at the AWS Cloud Development Kit (AWS CDK).

The AWS CDK helps developers use existing programming languages to generate CloudFormation templates. The end result of an AWS CDK operation is an imported resource in CloudFormation. However the syntax used with the AWS CDK makes for an easier comparison with Terraform. Here's an example of importing a resource by using the AWS CDK.

```
const importedBucket: IBucket = Bucket.fromBucketAttributes(
    scope,
    "imported-bucket",
    {
        bucketName: "My_S3_Bucket"
    }
);
```

An imported resource is usually created by calling a static method on the same class you use to create a new resource of the same kind. Calling `new Bucket(...` would create a new resource, and calling `Bucket.fromBucketAttributes(...` imports an existing one. You pass a subset of the bucket's properties into the function so the AWS CDK can find the right bucket. Another difference, however, is that creating a new bucket returns a full instance of the `Bucket` class, with all properties and methods available within. Importing the resource returns an `IBucket`, which is a type that contains only the properties that `Bucket` must have. Although you can import a resource from an external stack, the options of what you can do with it are limited.

In Terraform, a similar goal is accomplished by using data sources. Most defined Terraform resources have an accompanying data source available alongside it. The following is an example of a Terraform S3 bucket resource followed by its corresponding data source.

```
# S3 Bucket resource:
resource "aws_s3_bucket" "My_S3_Bucket" {
  bucket = "My_S3_Bucket"
}

# S3 Bucket data source:
data "aws_s3_bucket" "My_S3_Bucket" {
  bucket = "My_S3_Bucket"
}
```

The only difference between these two items is the name prefix. As shown in the documentation for a data source, there are fewer parameters available that you can pass to a data source than a resource. This is because the resource uses those parameters to declare all of the properties of a new S3 bucket, while the data source needs just enough information to uniquely identify and import the data of an existing resource.

The similarity between the syntax of a Terraform resource and a data source can be convenient, but it can also be problematic. It's common for novice Terraform developers to accidentally used a data source rather than a resource in their configuration. Terraform data sources are always read only. You can use them in place of the corresponding resource for read actions (such as supplying an ID name to another resource). However, you can't use them for write actions, which fundamentally change some aspect of the underlying resource. For this reason, you can think of a Terraform data source as a cloned version of the underlying resource.

Similar to the previous AWS CDK IBucket example, data sources are useful for read-only scenarios. If you need to get data from an existing resource but don't need to maintain that resource within your stack, use a data source. A good example of this is when you're creating an Amazon EC2 instance that uses the account's default VPC. Because that VPC already exists, all you need to do is pull in its data. The following code sample shows how to use data to identify the target VPC.

```
data "aws_vpc" "default" {
  default = true
}

resource "aws_instance" "instance1" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
  subnet_id     = data.aws_vpc.default.main_route_table_id
}
```

Understanding Terraform variables, local values, and outputs

Variables enhance code flexibility by allowing for placeholders within blocks of code. Variables can represent different values whenever the code is reused. Terraform distinguishes between its variable types by their modular scope. Input variables are external values that can be injected into a module, output values are internal values that can be shared externally, and local values always stay within their original scope.

Variables

AWS CloudFormation uses parameters to represent custom values that can be set and reset from one stack deployment to the next. Similarly, Terraform uses input variables, or *variables*. Variables can be declared anywhere in a Terraform configuration file and are usually declared with the required data type or default value. All three of the following expressions are valid Terraform variable declarations.

```
variable "thing_i_made_up" {
  type = string
}

variable "random_number" {
  default = 5
}

variable "dogs" {
  type = list(object({
    name  = string
    breed = string
  }))

  default = [
    {
      name  = "Sparky",
      breed = "poodle"
    }
  ]
}
```

To access Sparky's breed within the configuration, you'd use the variable `var.dogs[0].breed`. If a variable has no default and is not classified as nullable, then the value of the variable must be set for each deployment. Otherwise, it's optional to set a new value for the variable. In a root module, you can set current variable values on the command line, as environment variables, or in the terraform.tfvars file. The following example shows how to enter variable values in the **terraform.tfvars** file, which is stored in the module's top-level directory.

```
# terraform.tfvars
dogs = [
   {
      name  = "Sparky",
      breed = "poodle"
   },
   {
      name  = "Fluffy",
      breed = "chihuahua"
   }
]

random_number = 7

thing_i_made_up = "Kabibble"
```

The value for `dogs` in this example **terraform.tfvars** file would override the default value in the variable declaration. If you're declaring variables within a child module, you can set the variable values directly within the module declaration block, as shown in the following example.

```
module "my_custom_module" {
  source        = "modulesource/custom"
  version       = "0.0.1"
  random_number = 8
}
```

Some of the other arguments you can use when declaring a variable include:

- `sensitive` – Setting this to `true` prevents the variable value from being exposed in Terraform process outputs.
- `nullable` – Setting this to `true` allows the variable to have no value. This is convenient for variables where a default is not set.

- `description` – Add a description of the variable to the metadata for the stack.
- `validation` – Set validation rules for the variable.

One of the most convenient aspects of Terraform variables is the ability to add one or more validation objects within the variable declaration. You can use validation objects to add a condition that the variable must pass or else the deployment fails. You can also set a custom error message to show whenever the condition is violated.

For example, you're setting up a Terraform configuration file that members of your team will run. Before deploying the stacks, a team member needs to create a **terraform.tfvars** file to set an important configuration value. To remind them, you could do something like the following.

```
variable "important_config_setting" {
  type = string

  validation {
    condition     = length(var.important_config_setting) > 0
    error_message = "Don't forget to create the terraform.tfvars file!"
  }

  validation {
    condition     = substr(var.important_config_setting, 0, 7) == "prefix-"
    error_message = "Remember that the value always needs to start with 'prefix-'"
  }
}
```

As shown in this example, you can set multiple conditions inside a single variable. Terraform only shows error messages for failed conditions. In this way, you can enforce all kinds of rules on variable values. If a variable value causes a pipeline failure, you would know exactly why.

Local values

If there are any values within a module that you'd like to alias, use the `locals` keyword rather than declaring a default variable that will never be updated. As the name suggests, a `locals` block contains terms that are scoped internally to that specific module. If you want to transform a string value, such as by adding a prefix to a variable value for use in a resource name, using a local value might be a good solution. A single `locals` block can declare all local values for your module, as shown in the following example.

```
locals {
  moduleName    = "My Module"
  localConfigId = concat("prefix-", var.important_config_setting)
}
```

Just remember that when you're accessing the value, the `locals` keyword becomes singular, such as `local.LocalConfigId`.

Output values

If Terraform input variables are like CloudFormation parameters, then you could say that Terraform output values are like CloudFormation outputs. Both are used to expose values from within a deployment stack. However, because the Terraform module is more ingrained into the fabric of the tool, Terraform output values are also used to expose values within a module to a parent module or other child modules, even if those modules are all within the same deployment stack. If you're building two custom modules and the first module needs to access the ID value of the second module, then you'll need to add the following `output` block to the second module.

```
output "module_id" {
  value = local.module_id
}
Then in the first module you could use it like this:
module "first_module" {
  source = "path/to/first/module"
}

resource "example_resource" "example_resource_name" {
  module_id = module.first_module.module_id
}
```

Because Terraform output values can be used within the same stack, you can also use the `sensitive` attribute in an `output` block to suppress the value from being displayed in the stack output. Additionally, an `output` block can use `precondition` blocks in the same way that variables use `validation` blocks: to ensure variables follow a certain set of rules. This helps make sure that all values within a module exist as expected before proceeding with deployment.

```
output "important_config_setting" {
  value = var.important_config_setting
  precondition {
    condition     = length(var.important_config_setting) > 0
    error_message = "You forgot to create the terraform.tfvars file again."
  }
}
```

Understanding Terraform functions, expressions, and meta-arguments

One criticism of IaC tools that use declarative configuration files rather than common programming languages is that they make it more difficult to implement custom programmatic logic. In Terraform configurations, this issue is addressed by using functions, expressions, and meta-arguments.

Functions

One of the great advantages to using code to provision your infrastructure is the ability to store common workflows and reuse them again and again, often passing different arguments each time. Terraform functions are similar to AWS CloudFormation intrinsic functions, although their syntax is more similar to how functions are called in programmatic languages. You might have already noticed some Terraform functions, such as like substr, concat,length, and base64decode, in the examples in this guide. Like CloudFormation with intrinsic functions, Terraform has a series of built-in functions that are available for use in your configurations. For example, if a particular resource attribute takes a very large JSON object that would be inefficient to paste directly into the file, you could put the object in a **.json** file and use Terraform functions to access it. In the following example, the `file` function returns the contents of the file in string form, and then the `jsondecode` function converts it into an object type.

```
resource "example_resource" "example_resource_name" {
  json_object = jsondecode(file("/path/to/file.json"))
}
```

Expressions

Terraform also allows for conditional expressions, which are similar to CloudFormation `condition` functions except that they use the more traditional ternary operator syntax. In the following example, the two expressions return the exact same result. The second example is what Terraform calls a splat expression. The asterisk causes Terraform to loop through the list and create a new list by using just the `id` property of each item.

```
resource "example_resource" "example_resource_name" {
  boolean_value  = var.value ? true : false
  numeric_value  = var.value > 0 ? 1 : 0
  string_value   = var.value == "change_me" ? "New value" : var.value
  string_value_2 = var.value != "change_me" ? var.value : "New value"
}
There are two ways to express for loops in a Terraform configuration:
resource "example_resource" "example_resource_name" {
  list_value   = [for object in var.ids : object.id]
  list_value_2 = var.ids[*].id
}
```

Meta-arguments

In the previous code example, `list_value` and `list_value_2` are referred to as *arguments*. You might be familiar with some of these meta-arguments already. Terraform also has a few *meta-arguments*, which act just like arguments but with some extra functionality:

- The depends_on meta-argument is very similar to the CloudFormation DependsOn attribute.
- The provider meta-argument allows you to use multiple provider configurations at once.
- The lifecycle meta-argument allows you to customize resource settings, similar to removal and deletion policies in CloudFormation.

Other meta-arguments allow for function and expression functionality to be added directly to a resource. For example, the count meta-argument is a useful mechanism to create multiple similar resources at the same time. The following example demonstrates how to create two Amazon Elastic Container Service (Amazon EKS) clusters without using the `count` meta-argument.

```
resource "aws_eks_cluster" "example_0" {
  name     = "example_0"
  role_arn = aws_iam_role.cluster_role.arn
  vpc_config {
    endpoint_private_access = true
    endpoint_public_access  = true
    subnet_ids              = var.subnet_ids[0]
  }
}

resource "aws_eks_cluster" "example_1" {
  name     = "example_1"
  role_arn = aws_iam_role.cluster_role.arn
  vpc_config {
    endpoint_private_access = true
    endpoint_public_access  = true
    subnet_ids              = var.subnet_ids[1]
  }
}
```

The following example demonstrates how to use the `count` meta-argument to create two Amazon EKS clusters.

```
resource "aws_eks_cluster" "clusters" {
  count    = 2
  name     = "cluster_${count.index}"
  role_arn = aws_iam_role.cluster_role.arn
  vpc_config {
    endpoint_private_access = true
    endpoint_public_access  = true
    subnet_ids              = var.subnet_ids[count.index]
  }
}
```

To give each a unit name, you can access the list index within the resource block at `count.index`. But what if you want to create multiple similar resources that are a little more complex? That's where the for_each meta-argument comes in. The `for_each` meta-argument is very similar to `count`, except that you pass in a list or an object instead of a number. Terraform creates a new resource for each member of the list or object. It is similar to if you set `count = length(list)`, except you can access the contents of the list rather than the loop index.

This works for both a list of items or a single object. The following example would create two resources that have `id-0` and `id-1` as their IDs.

```
variable "ids" {
  default = [
    { id = "id-0" },
    { id = "id-1" },
  ]
}

resource "example_resource" "example_resource_name" {
  # If your list fails, you might have to call "toset" on it to convert it to a set
  for_each = toset(var.ids)
  id       = each.value
}
```

The following example would create two resources as well, one for Sparky, the poodle, and one for Fluffy, the chihuahua.

```
variable "dogs" {
  default = {
    poodle    = "Sparky"
    chihuahua = "Fluffy"
  }
}

resource "example_resource" "example_resource_name" {
  for_each = var.dogs
  breed    = each.key
  name     = each.value
}
```

Just like you can access the loop index in count by using count.index, you can access the key and the value of each item in a for_each loop by using the each object. Because for_each iterates over both lists and objects, the each key and value can get a little confusing to keep track of. The following table shows the different ways that you can use the for_each meta-argument and how you can reference the values upon each iteration.

Example	`for_each` type	First iteration	Second iteration

A	`[“poodle”, “chihuahua”]`	`each.key = "poodle"`
		
		`each.value = null`
			`each.key = "chihuahua"`
			
			`each.value = null`

B	`[`
	
	`{`
	
	`type = "poodle",`
		`each.key = {`
		
		`type = “poodle”,`
		
		`name = “Sparky”`
			`each.key = {`
			
			`type = “chihuahua”,`
			
			`name = “Fluffy”`

Example	`for_each` type	First iteration	Second iteration

	`name = "Sparky"`
	
	`},`
	
	`{`
	
	`type = "chihuahua",`
	
	`name = "Fluffy"`
	
	`}`
	
	`]`
		`}`
		
		`each.value = null`
			`}`
			
			`each.value = null`

C	`{`
	
	`poodle = “Sparky”,`
	
	`chihuahua = “Fluffy”`
	
	`}`
		`each.key = “poodle”`
		
		`each.value = “Sparky”`
			`each.key = “chihuahua”`
			
			`each.value = “Fluffy”`

Example	`for_each` type	First iteration	Second iteration

D	`{`
	
	`dogs = {`
	
	`poodle = “Sparky”,`
	
	`chihuahua = “Fluffy”`
	
	`},`
	
	`cats = {`
	
	`persian = “Felix”,`
	
	`burmese = “Morris”`
	
	`}`
	
	`}`
		`each.key = “dogs”`
		
		`each.value = {`
		
		`poodle = “Sparky”,`
		
		`chihuahua = “Fluffy”`
		
		`}`
			`each.key = “cats”`
			
			`each.value = {`
			
			`persian = “Felix”,`
			
			`burmese = “Morris”`
			
			`}`

Example	`for_each` type	First iteration	Second iteration

E	`{`
	`dogs = [`
	`{`
	`type = “poodle”,`
	`name = “Sparky”`
	`},`
	`{`
	`type = “chihuahua”,`
	`name = “Fluffy”`
	`}`
	`],`
	`cats = [`
	`{`
	`type = “persian”,`
	`name = “Felix”`
	`},`
	`{`
	`type = “burmese”,`
		`each.key = “dogs”`
		`each.value = [`
		`{`
		`type = “poodle”,`
		`name = “Sparky”`
		`},`
		`{`
		`type = “chihuahua”,`
		`name = “Fluffy”`
		`}`
		`]`
			`each.key = “cats”`
			`each.value = [`
			`{`
			`type = “persian”,`
			`name = “Felix”`
			`},`
			`{`
			`type = “burmese”,`
			`name = “Morris”`
			`}`
			`]`

Example	`for_each` type	First iteration	Second iteration

	`name = “Morris”`
	
	`}`
	
	`]`
	
	`}`		

So if `var.animals` was equal to row E, then you could create one resource per animal by using the following code.

```
resource "example_resource" "example_resource_name" {
  for_each = var.animals
  type     = each.key
  breeds   = each.value[*].type
  names    = each.value[*].name
}
```

Alternatively, you could create two resources per animal by using the following code.

```
resource "example_resource" "example_resource_name" {
  for_each = var.animals.dogs
  type     = "dogs"
  breeds   = each.value.type
  names    = each.value.name
}

resource "example_resource" "example_resource_name" {
  for_each = var.animals.cats
  type     = "cats"
  breeds   = each.value.type
  names    = each.value.name
}
```

FAQ

When should I use Terraform instead of CloudFormation?

In general, if your workloads are primarily based in AWS, AWS CloudFormation provides a level of native support that Terraform can't match. However, if your workloads include quite a few third-party processes or they're spread out among multiple cloud providers, Terraform is a tool that you might want to consider.

When should I use the AWS CDK instead of CloudFormation?

When you use the AWS Cloud Development Kit (AWS CDK), you're also using CloudFormation. The AWS CDK allows you to use a common programming language to generate CloudFormation templates. If you're experienced in any of the programming languages that the AWS CDK supports, the AWS CDK can reduce the time required to generate CloudFormation templates.

Is there a tool like the AWS CDK that generates Terraform configurations?

Compared to the AWS CDK, the CDK for Terraform (CDKTF) uses the same construct library to provision resources and the same jsii engine to support multiple programming languages. You can use it to generate Terraform configurations in the same way that the AWS CDK generates CloudFormation templates.

How do I learn more about Terraform?

For more information about advanced Terraform concepts, see the Terraform documentation. It also describes the components of all major providers and open source modules.

Related resources

AWS documentation

- AWS CDK documentation
- AWS CloudFormation documentation
- Terraform: Beyond the Basics with AWS (AWS blog post)

Other resources

- CDK for Terraform documentation
- Terraform documentation

Appendix: Terraform attribute access examples

Resource

```
resource "aws_s3_bucket" "myS3Bucket" {
     bucket = "my-s3-bucket"
}

bucketName = aws_s3_bucket.myS3Bucket.bucket
```

Data source

```
data "aws_s3_bucket" "myS3Bucket" {
     bucket = "my-s3-bucket"
}

bucketName = data.aws_s3_bucket.myS3Bucket.bucket
```

Module

```
module "eks" {
    source = "terraform-aws-modules/eks/aws"
    version = "20.2.1"
}

vpc_id = module.eks.vpc_id
```

Variable

```
variable "my_variable" = {
   default = "dog"
}

animalType = var.my_variable
```

Local

```
locals {
  type = "dog"
}

animalType = local.type
```

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	—	March 29, 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.