# Deployment Strategies for Serverless

**Terminology**

To understand the deployment strategies for serverless applications, we will first cover the terminology of versions, aliases, and traffic shifting. Each AWS Lambda function can have any number of versions and aliases associated with them.

* ***Versions*** are snapshots of a function that includes the code and configuration, and it is a good practice to publish a new version each time you update your function code. When you invoke a specific version (using the function name and version number combination) you will get the same code and configuration regardless of the state of the function. This protects you against accidentally updating production code. To use versions, you should create an alias, which is a pointer to a version.
* ***Aliases*** have a name and an Amazon Resource Number (ARN) like the function and are accepted by the Invoke APIs. If you invoke an alias, Lambda will in turn invoke the version that the alias is pointing to. In production, you would first update your function code, publish a new version, and invoke the version directly to run tests against it. After you are satisfied, you would change the alias to point to the new version.
* ***Traffic shifting*** can shift incoming traffic between two versions of a Lambda function based on pre-assigned weights. You can use this feature to gradually shift traffic between two versions, helping you reduce the risk of new Lambda deployments. You can also change your Lambda function’s code without affecting other upstream dependencies that rely on the alias.

**Deploying serverless applications**

If you use AWS SAM to create your serverless application, it comes built-in with AWS [CodeDeploy](https://docs.aws.amazon.com/codedeploy/latest/userguide/welcome.html" \t "_blank) to provide gradual Lambda deployments. With a few lines of configuration, AWS SAM does the following for you:

* Deploys new versions of your Lambda function, and automatically creates aliases that point to the new version.
* Gradually shifts customer traffic to the new version until you're satisfied that it's working as expected, or you roll back the update.
* Defines pre-traffic and post-traffic test functions to verify that the newly deployed code is configured correctly and your application operates as expected.
* Rolls back the deployment if Amazon CloudWatch alarms are generated.

**Deployment options**

Now that you know about AWS SAM, you can learn about your deployment options. The following list describes other traffic-shifting options that are available:

* **Canary**: Traffic is shifted in two increments. You can choose from predefined canary options. The options specify the percentage of traffic that's shifted to your updated Lambda function version in the first increment, and the interval, in minutes, before the remaining traffic is shifted in the second increment.
* **Linear**: Traffic is shifted in equal increments with an equal number of minutes between each increment. You can choose from predefined linear options that specify the percentage of traffic that's shifted in each increment and the number of minutes between each increment.
* **All-at-once**: All traffic is shifted from the original Lambda function to the updated Lambda function version at once.

An all-at-once deployment will shift traffic instantly from one version to another, while canary and linear, are much safer, more gradual deployment options.

# AppSpec File

The application specification file (AppSpec file) is file used by AWS CodeDeploy to manage a deployment. It can be written in [YAML](http://www.yaml.org/) or JSON.

**AppSec file for Amazon ECS applications**

For Amazon Elastic Container Service (Amazon ECS) applications, the AppSpec file is used by AWS CodeDeploy to determine:

* Your Amazon ECS task definition file. This file is specified with its Amazon Resource Number (ARN) in the **TaskDefinition** instruction of the AppSpec file.
* The container and port in your replacement task set where your Application Load Balancer or Network Load Balancer reroutes traffic during a deployment. This setting is specified in the **LoadBalancerInfo** instruction of the AppSpec file.
* Optional information about your Amazon ECS service, such the platform version it runs on, its subnets, and its security groups.
* Optional Lambda functions to run during hooks that correspond with lifecycle events during an Amazon ECS deployment.

Here is an example of an AppSpec file for deploying an Amazon ECS service, written in YAML.

**Table

Description automatically generated with medium confidence**

**AppSpec file for AWS Lambda applications**

For Lambda applications, the AppSpec file is used by CodeDeploy to determine:

* Which Lambda function version to deploy.
* Which Lambda functions to use as validation tests.

For Lambda, you must specify your application alias, and the current and target version you’re deploying to. Here is an example of an AppSpec file for deploying a Lambda function version, written in YAML.



**Hooks**

The content in the **'hooks'** section of the AppSpec file varies, depending on the compute platform for your deployment. The **'hooks'** section for an Amazon EC2 or on-premises deployment contains mappings that link deployment lifecycle event hooks to one or more scripts. The **'hooks'** section for a Lambda or an Amazon ECS deployment specifies Lambda validation functions to run during a deployment lifecycle event. If an event hook isn’t present, no operation is run for that event. This section is required only if you are running scripts or Lambda validation functions as part of the deployment.

**Permissions**

The **'permissions'** section specifies how special permissions (if any) should be applied to the files and the directories in the **'files'** section after they are copied to the instance. You can specify multiple **object** instructions. This section is optional. It applies to Amazon Linux, Ubuntu Server, and RHEL instances only. **Note:**The **'permissions'** section is used only for Amazon EC2 or on-premises deployments. It’s not used for Lambda or Amazon ECS deployments.

# Deploying Updates to Lambda with SAM and CodeDeploy

If you are running a serverless application that uses AWS Lambda, you might need to address additional considerations when you deploy updates to your Lambda functions. As an example, dependencies between service resources could cause issues for your serverless deployment if the various resources that your function needs aren’t created in the correct order. To help with serverless deployments, AWS offers the AWS Serverless Application Model (AWS SAM), which is an open-source framework that you can use as a scaffold for organizing your serverless resources. When you create or update an AWS SAM template, you use YAML and a shorthand syntax specific to AWS SAM to model your application. With AWS SAM, you can use code to define your application’s functions and other resources, such as APIs or event-source mappings. When you deploy your application, AWS SAM expands its syntax into AWS CloudFormation syntax. By turning your serverless infrastructure into code, you can build and deploy your serverless applications more quickly and reliably. Say that you have been working on a new version of a Lambda function for your application. The updated function has passed all your tests, and you’re now ready to deploy the new version. After deployment, you want production traffic to gradually shift from the older version of the function to the new version so you can monitor how it performs. You could deploy the new version of the function (and all its dependencies) through the console or the AWS Command Line Interface (AWS CLI), but doing so can be a very manual process. Instead of going the manual route, you could deploy the DevOps way, using AWS SAM and CodeDeploy to both deploy the new version and automatically shift production traffic to it. To do so, edit the AWS SAM template to configure the necessary CodeDeploy settings. By including the **AutoPublishAlias: live** setting in the AWS SAM definition for your Lambda function, CodeDeploy deploys a new version of the Lambda function. After it deploys the function, it also creates an alias called live that points to the new version, which CodeDeploy will use to shift traffic between the two versions. To specify which deployment model you want to use, add a **DeploymentPreference** section to your template’s function definition. The deployment options include:

* Canary - This option shifts traffic in two increments spaced by an interval (in minutes). You can use predefined options to specify the percentage of traffic shifted in the first increment and the duration of the interval before CodeDeploy shifts the remaining traffic. For example, **Canary10Percent30Minutes** will send 10 percent of traffic to the new version and 30 minutes later, it will complete the deployment by sending all traffic to the new version.
* Linear - This option shifts traffic in equal increments spaced by an equal number of minutes. You can use predefined options to specify the percentage of traffic shifted in the increments, and the duration of the interval between each increment. For example, **Linear10PercentEvery10Minutes** will continuously shift 10 percent of traffic to the new version every 10 minutes, until 100 percent of the traffic gets sent to the new version.
* All-at-once - This option shifts all traffic from the old version to the new version.

You can also use the **DeploymentPreference** section to configure additional deployment resources. Examples include Amazon CloudWatch alarms that monitor for deployment errors, or hooks that test whether the Lambda function is performing as expected before or after any traffic shifting occurs.

Here’s an example AWS SAM template that defines the deployment for a Lambda function:

Text

Description automatically generated

With your CodeDeploy settings in the AWS SAM template, you can deploy your new version (for example, by using the AWS SAM CLI). To see the application traffic shift from the old version and the new one as you specified in real time, you can use either the CodeDeploy console or the Lambda console. You could also perform tests or view logs to check whether the new deployment is working as expected. And if you need to update your Lambda function in the future, you can apply DevOps practices and use AWS SAM and CodeDeploy to redeploy and monitor your updates in an automated, repeatable, and reliable way.

# Abstracting IaC on AWS

A fundamental principle of DevOps is to treat infrastructure the same way developers treat code. Infrastructure provisioning, orchestration, and deployment should also support the use of the infrastructure as code (IaC). AWS provides a number of IaC offerings. One offering is [AWS CloudFormation](http://aws.amazon.com/cloudformation/), which is a service you can use to specify any cloud infrastructure you need in a simple template file. CloudFormation then provisions the infrastructure for you. Developers can use CloudFormation to create AWS resources in an orderly and predictable fashion. Depending on the complexity of the architecture you want to create—especially if you are creating many serverless resources—CloudFormation templates can be lengthy. For example, a CloudFormation template might need many lines, which could make it more difficult to reuse when you want to share code with the DevOps team. To help manage this complexity, AWS created the [AWS Cloud Development Kit (AWS CDK)](https://aws.amazon.com/cdk/). The AWS CDK is an abstraction layer that provides a framework for designing cloud infrastructure in code using a language of your choice. It will then build a CloudFormation template out of the infrastructure you define. You can use the AWS CDK to model application infrastructure with TypeScript, Python, Java, and .NET. Developers can use their existing integrated development environment (IDE) and take advantage of tools like autocomplete and inline documentation to accelerate infrastructure development. The AWS Serverless Application Model (AWS SAM) and the AWS CDK both abstract IaC to make deployments easier. (The previous course covered AWS SAM. If you’re not familiar with the topic, you can refer to the previous readings.) By generating your CloudFormation templates from source code, you can extend the integration with external sources. You can use toolkits and resources that are available for the supported programming languages—like variables, data structures, external API calls, loops, functions, arrays, and other—and do any tasks that are possible with a programming language. For example, say you want to create five Amazon Elastic Compute Cloud (Amazon EC2) instances. If you used only CloudFormation, you would need to copy and paste five individual resources of the type **AWS::EC2::Instance** into the template. However, if you use Python with the AWS CDK, you can create the five EC2 instances by putting [a construct](https://docs.aws.amazon.com/cdk/api/latest/docs/@aws-cdk_aws-s3.Bucket.html)that renders the instances inside a **for** loop, which is not possible by using only CloudFormation.

**Benefits of using the AWS CDK**

* **Easier cloud onboarding -**The AWS CDK accelerates your onboarding to AWS because there are fewer new things to learn (after you already know the basics of the programming language you decided to use).
* **Faster development process -**The AWS CDK gives you the expressive power of programming languages for defining infrastructure.
* **Customizable and shareable -**With the AWS CDK, you can design your own reusable components that meet your organization’s requirements for security, compliance, and governance.
* **No context switching -**You can build your cloud application with the AWS CDK without leaving your IDE or command line interface (CLI).

The AWS CDK Toolkit is the primary tool for interacting with your AWS CDK application. In the AWS Command Line Interface (AWS CLI), use the **cdk** command. The AWS CDK Toolkit runs your application, checks the application model you defined, and produces and deploys the CloudFormation templates that were rendered by the AWS CDK.