AWS Compute Blog

Building well-architected serverless applications: Approaching application lifecycle management – part 2

by Julian Wood | on 17 JUN 2020 | in Amazon CloudWatch, Amazon DynamoDB, Amazon Simple Storage Service (S3), AWS Amplify, AWS CloudFormation, AWS CodeDeploy, AWS Lambda, AWS Well-Architected Tool, Serverless | Permalink | Share

This series of blog posts uses the AWS Well-Architected Tool with the Serverless Lens to help customers build and operate applications using best practices. In each post, I address the nine serverless-specific questions identified by the Serverless Lens along with the recommended best practices. See the Introduction post for a table of contents and explanation of the example application.

Question OPS2: How do you approach application lifecycle management?

This post continues part 1 of this Operational Excellence question. Previously, I covered using infrastructure as code with version control to deploy applications in a repeatable manner.

Good practice: Prototype new features using temporary environments

Storing application configuration as infrastructure as code allows deployment of multiple, repeatable, isolated versions of an application.

Create multiple temporary environments for new features you may need to prototype, and tear them down as you complete them. Temporary environments enable fine grained feature isolation and higher fidelity development when interacting with managed services. This allows you to gain confidence your workload integrates and operates as intended.

These environments can also be in separate accounts which help isolate limits, access to data, and resiliency. For best practices on multi-account deployments, see the AWS Partner Network blog post: Best Practices Guide for Multi-Account AWS Deployments.

There are a number of ways to deploy separate environments for an application. To make the deployment simpler, it is good practice to separate dynamic configuration from your infrastructure logic.

For an application managed via the AWS Serverless Application Model (AWS SAM), use an AWS SAM CLI parameter to specify a new `stack-name` which deploys a new copy of the application as a separate stack.

For example, there is an existing AWS SAM application with a `stack-name` of *app-test*. To deploy a new copy, specify a new `stack-name` of *app-newtest* with the following command line:

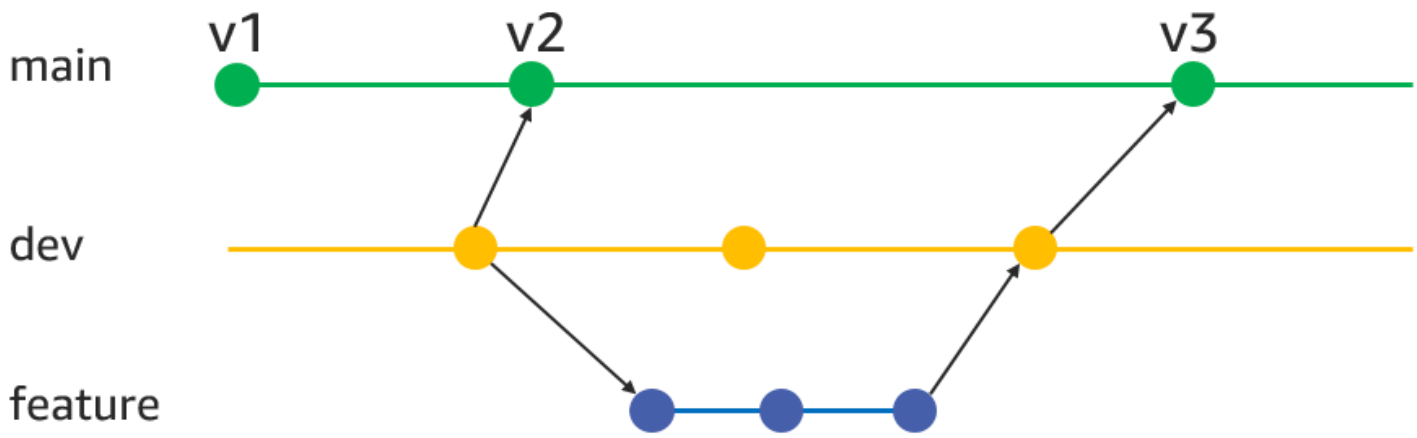
```
Bash
sam deploy --stack-name app-newtest
```

This deploys a whole new copy of the application in the same account as a separate stack.

For the serverless airline example used in this series, deploy a whole new copy of the application following the deployment instructions, either into the same AWS account, or a completely different account. This is useful when each developer in a team has a sandbox environment. In this example, you only need to configure payment provider credentials as environment variables and seed the database with possible flights as these are currently manual post installation tasks.

However, maintaining an entirely separate codebase copy of an application becomes difficult to manage and reconcile over time.

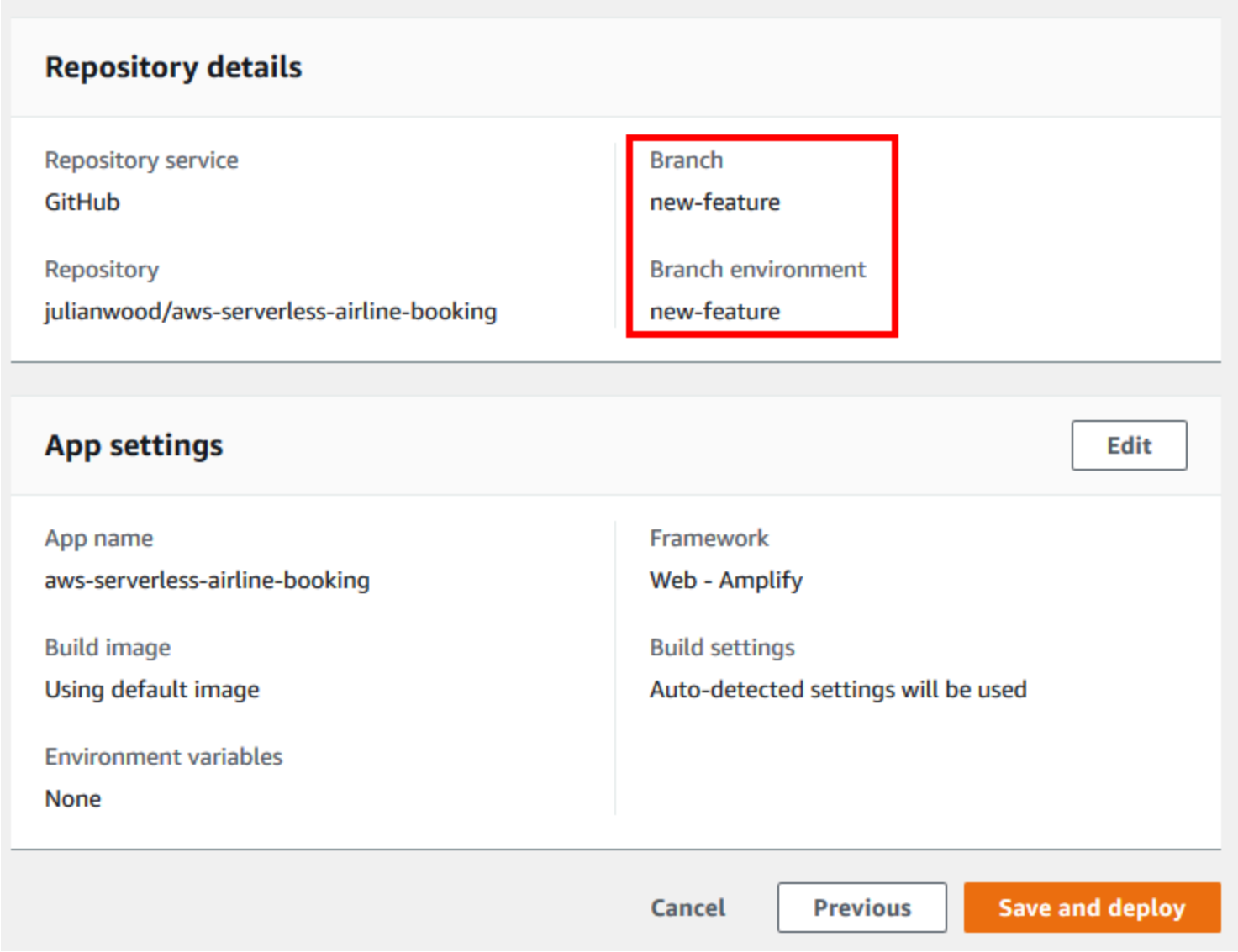
As the airline application code is stored in a fork in a GitHub account, use git branches for separate environments. In typical development teams, developers may deploy a *main* branch to production, have a *dev* branch as staging, and create *feature* branches when working on new functionality. This allows safe prototyping in sandbox environments without affecting the main codebase, and use git as a mechanism to merge code and resolve conflicts. Changes are automatically pushed to production once they are merged into the *main* (or production) branch.



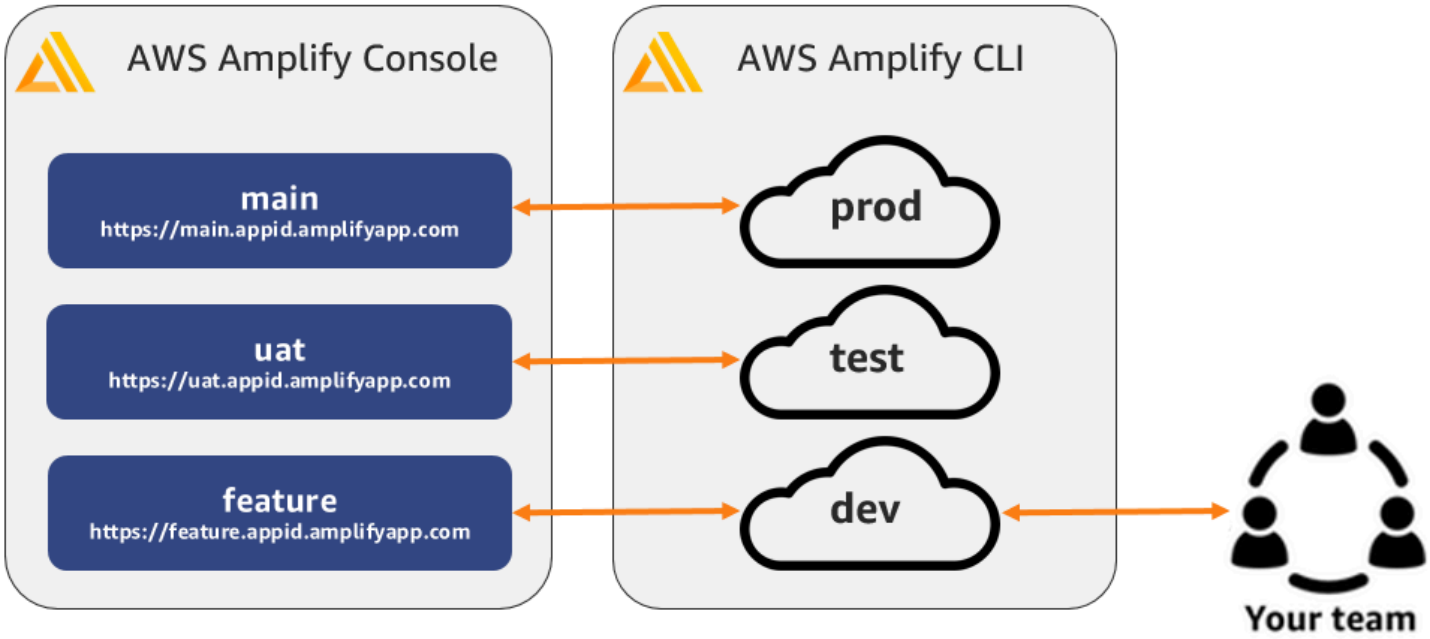
Git branching flow

As the airline example is using AWS Amplify Console, there are a few different options to create a new environment linked to a feature branch.

You can create a whole new Amplify Console app deployment, either in a separate Region, or in a separate AWS account, which then connects to a feature branch by following the deployment instructions. Create a new branch called *new-feature* in GitHub and in the Amplify Console, select **Connect App**, and navigate to the repository and the *new-feature* branch. Configure the payment provider credentials as environment variables.

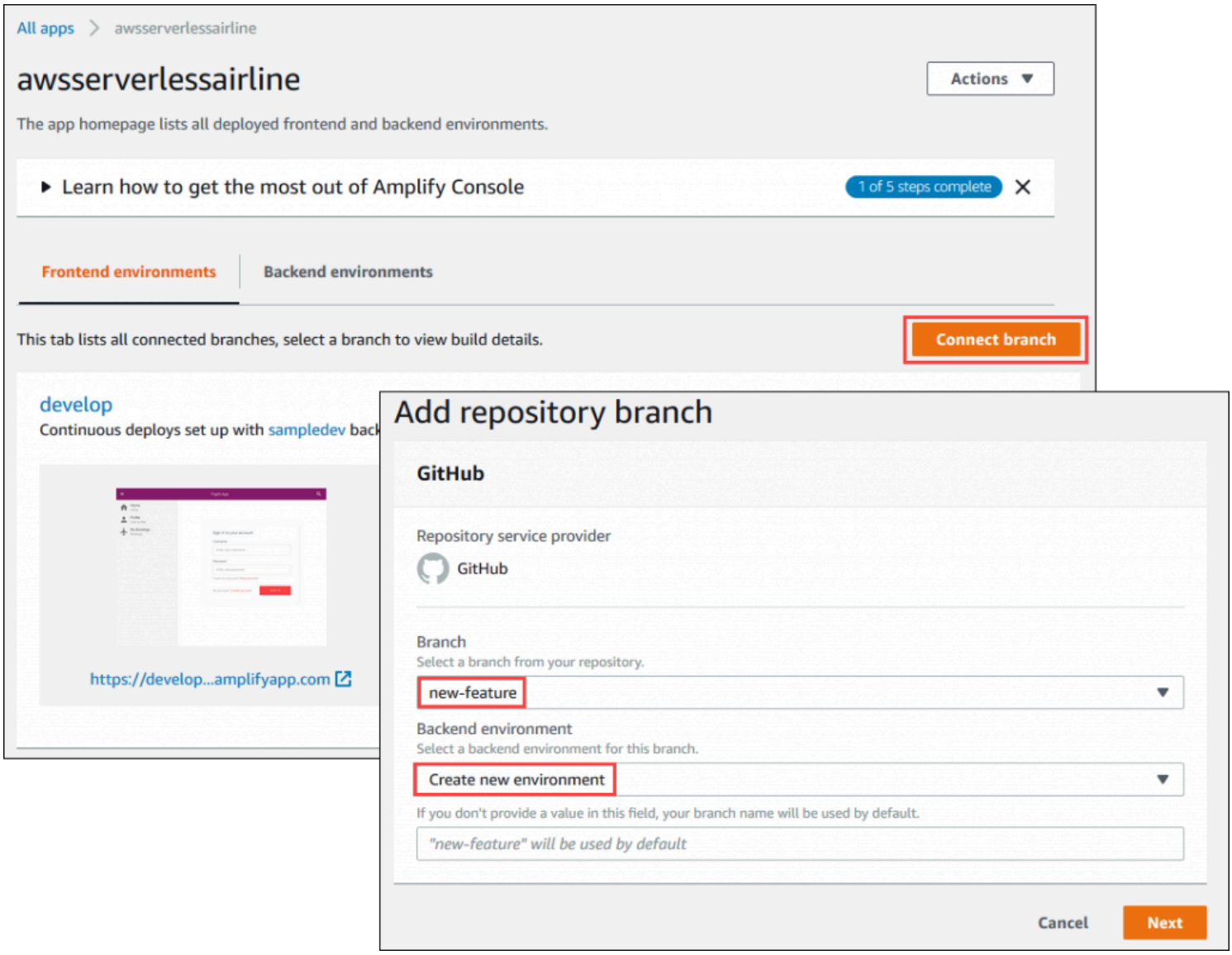


You can also connect the existing Amplify Console deployment to a git branch, deploying the *new-feature* branch into the same AWS account and Region.



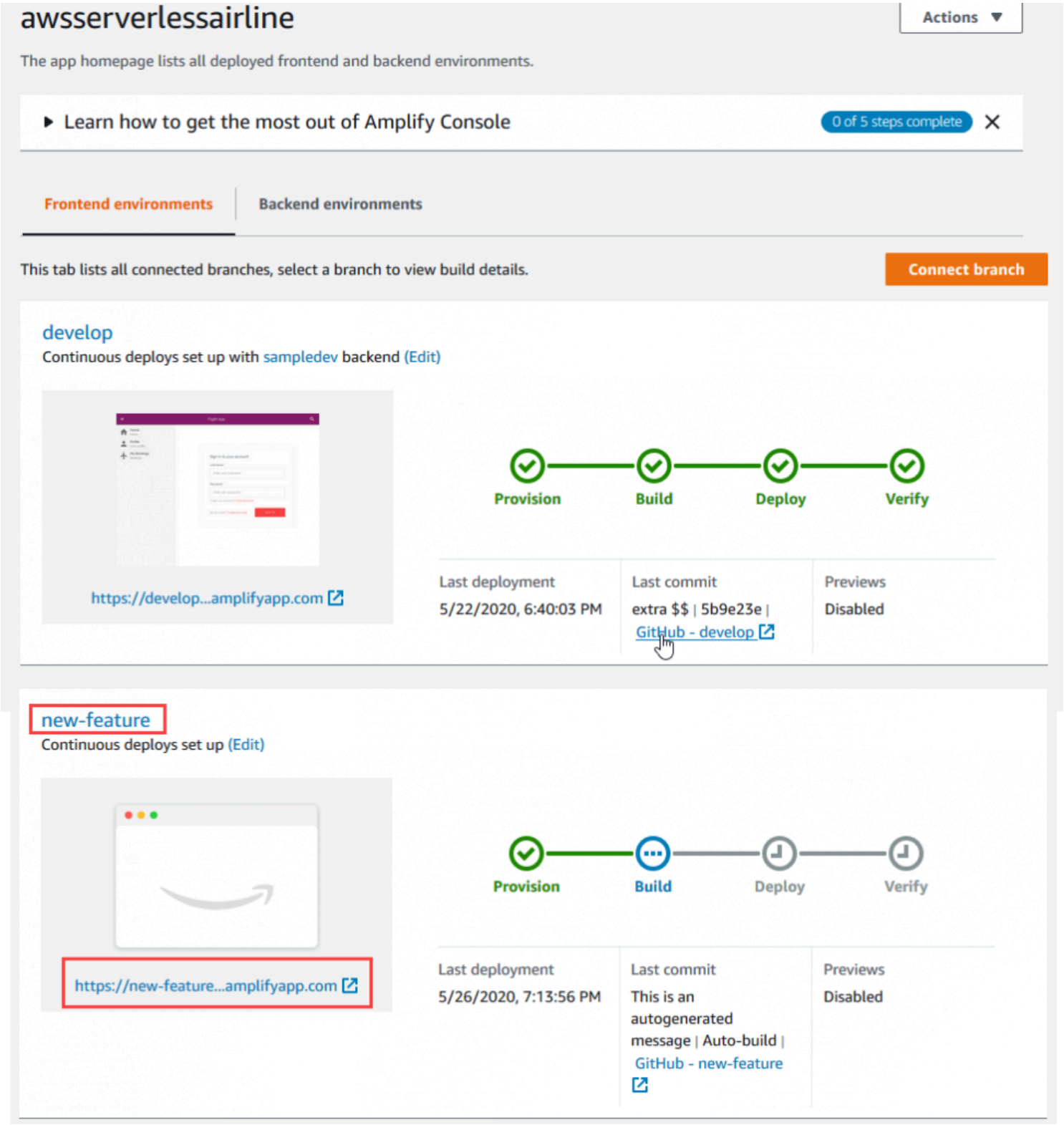
Amplify Environments

In the Amplify Console, navigate to the existing app, select **Connect Branch**, and choose the *new-feature* branch. Create a new *Backend environment* to deploy the full stack. If the feature branch is only frontend code changes, you can choose to use the same backend components.



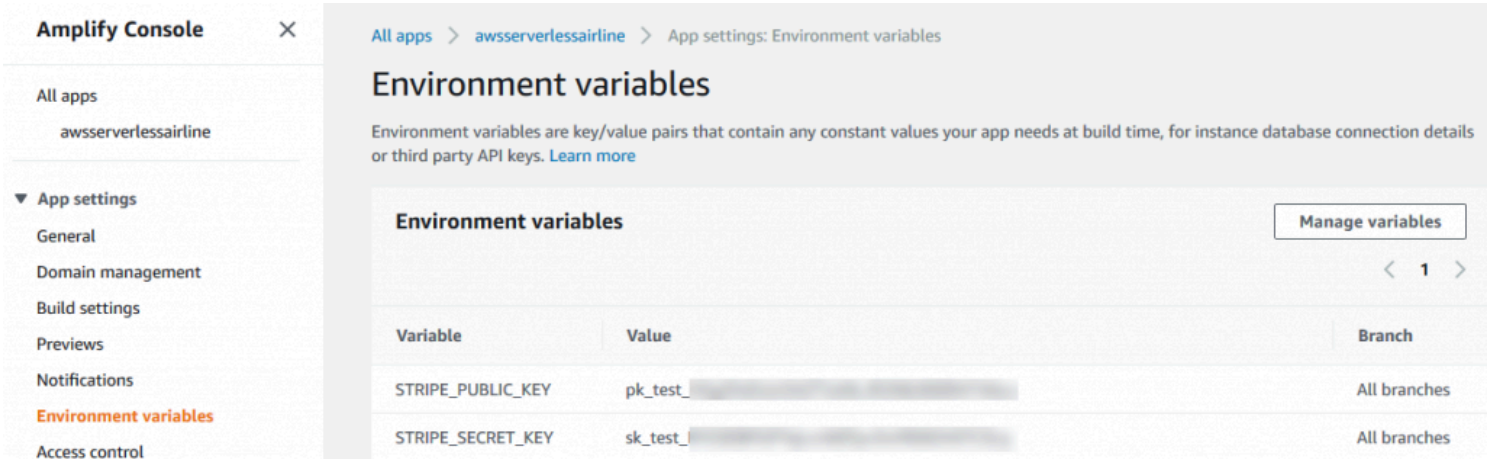
Connect Amplify Console to feature branch

Amplify Console then deploys a new stack in addition to the *develop* branch based on the code in the *feature-branch*.



New feature branch deploying within existing deployment.

You do not need to add the payment provider environment variables as these are stored per application, per Region, for all branches.



Amplify environment variables for All Branches.

Using git and branching with Amplify Console, you have automatic deployments when any changes are pushed to the GitHub repository. If there are any issues with a particular deployment, you can revert the changes in git which will kick off a redeploy to a known good version. Once you are happy with the feature, you can merge the changes into the production branch which will again kick off another deployment.

As it is simple to set up multiple test environments, make sure to practice good application hygiene, as well as cost management, by identifying and deleting any temporary environments that are no longer required. It may be helpful to include the stack owner's contact details via CloudFormation tags. Use Amazon CloudWatch scheduled tasks to notify and tag temporary environments for deletion, and provide a mechanism to delay its deletion if needed.

Prototyping locally

With AWS SAM or a third-party framework, you can run API Gateway, and invoke Lambda function code locally for faster development iteration. Local debugging and testing can help for quick confirmation that function code is working, and is also useful for some unit tests. Local testing cannot duplicate the full functionality of the cloud. It is suited to testing services with custom logic, such as Lambda, rather than trying to duplicate all cloud managed services such as Amazon SNS, or Amazon S3 locally. Don't try to bring the cloud to the test, rather bring the testing to the cloud.

Here is an example of executing a function locally.

I use AWS SAM CLI to invoke the *Airline-GetLoyalty* Lambda function locally to test some functionality. AWS SAM CLI uses Docker to simulate the Lambda runtime. As the function only reads from DynamoDB, I use stubbed data, or can set up DynamoDB Local.

1. I pass a JSON event to the function to simulate the event from API Gateway, as well as passing in environment variables as JSON. Create sample events using sam local generate-event.

2. I run `sam build GetFunc` to build the function dependencies, in this case NodeJS.

Bash
```
$ sam build GetFunc
Building resource 'GetFunc'
```

```
Running NodejsNpmBuilder:NpmPack
Running NodejsNpmBuilder:CopyNpmrc
Running NodejsNpmBuilder:CopySource
Running NodejsNpmBuilder:NpmInstall
Running NodejsNpmBuilder:CleanUpNpmrc

Build Succeeded
```

3. I run `sam local invoke` passing in the event payload and environment variables. This spins up a Docker container, executes the function, and returns the result.

Bash

```
$ sam local invoke --event src/get/event.json --env-vars local-env-vars.json GetFunc
Invoking index.handler (nodejs10.x)

Fetching lambci/lambda:nodejs10.x Docker container image......
Mounting /home/ec2-user/environment/samples/aws-serverless-airline-booking/src/backend/
START RequestId: 7be7e9a5-9f2f-1520-fbd1-a013485105d3 Version: $LATEST
END RequestId: 7be7e9a5-9f2f-1520-fbd1-a013485105d3
REPORT RequestId: 7be7e9a5-9f2f-1520-fbd1-a013485105d3 Init Duration: 249.89 ms Duratio

{"statusCode": 200,"body": "{\"points\":0,\"level\":\"bronze\",\"remainingPoints\":500
```

For more information on using AWS SAM to run API Gateway, and invoke Lambda functions locally, see the AWS Documentation. For third-part framework solutions, see Invoking AWS Lambda functions locally with Serverless framework and Develop locally against cloud services with Stackery.

Improvement plan summary:

1. Use a serverless framework to deploy temporary environments named after a feature.
2. Implement a process to identify temporary environments that may not have been deleted over an extended period of time
3. Prototype application code locally and test integrations directly with managed services

Good practice: Use a rollout deployment mechanism

Use a rollout deployment for production workloads as opposed to all-at-once mechanisms. Rollout deployments reduce the risk of a failed deployment by gradually deploying application changes to a limited set of customers. Use all-at-once deployments to deploy the entire application. This is best suited for non-production systems.

AWS Lambda versions and aliases

For production Lambda functions, it is best to deploy a new function version for every deployment. Versions can represent the stable version or reflect particular features. Create Lambda aliases which are pointers to particular function versions. Invoke Lambda functions using the aliases, with a specific alias for the stable production version. If an alias is not specified, the latest application code deployment is invoked which may not reflect a stable version or a desired feature. Use the new feature alias version for testing without affecting users of the stable production version.

AWS Lambda function versions and aliases

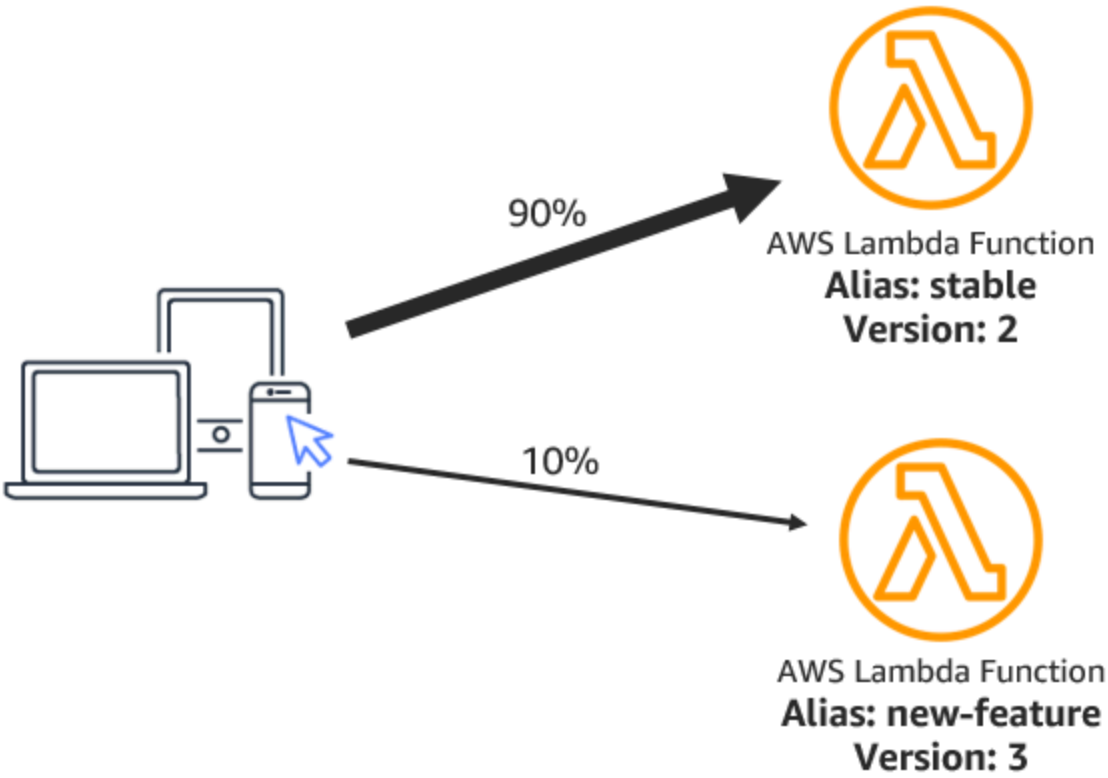
See AWS Documentation to manage Lambda function versions and aliases using the AWS Management Console, or Lambda API.

Alias routing

Use Lambda alias' routing configuration to introduce traffic shifting to send a small percentage of traffic to a second function alias or version for a rolling deployment. This is commonly called a canary release.

For example, configure Lambda alias named *stable* to point to function version *2*. A new function version *3* is deployed with alias *new-feature*. Use the *new-feature* alias to test the new deployment without impacting production traffic to the *stable* version.

During production rollout, use alias routing. For example, 90% of invocations route to the *stable* version while 10% route to alias *new-feature* pointing to version *3*. If the 10% is successful, deployment can continue until all traffic is migrated to version 3, and the *stable* alias is then pointed to version 3.



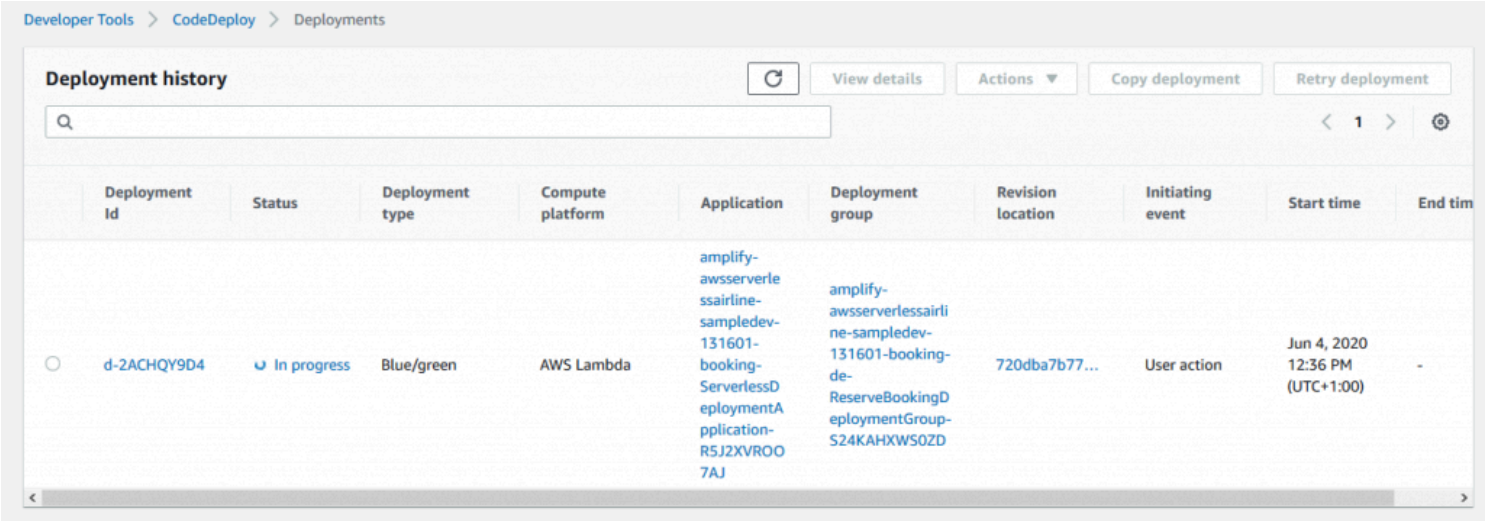
AWS Lambda alias routing

AWS SAM supports gradual Lambda deployments with a feature called Safe Lambda deployments using AWS CodeDeploy. This creates new versions of a Lambda function, and automatically creates aliases pointing to the new version. Customer traffic gradually shifts to the new version or rolls back automatically if any specified CloudWatch alarms trigger. AWS SAM supports canary, linear, and all-at-once deployments preference types.

Pre-traffic and post-traffic Lambda functions can also verify if the newly deployed code is working as expected.

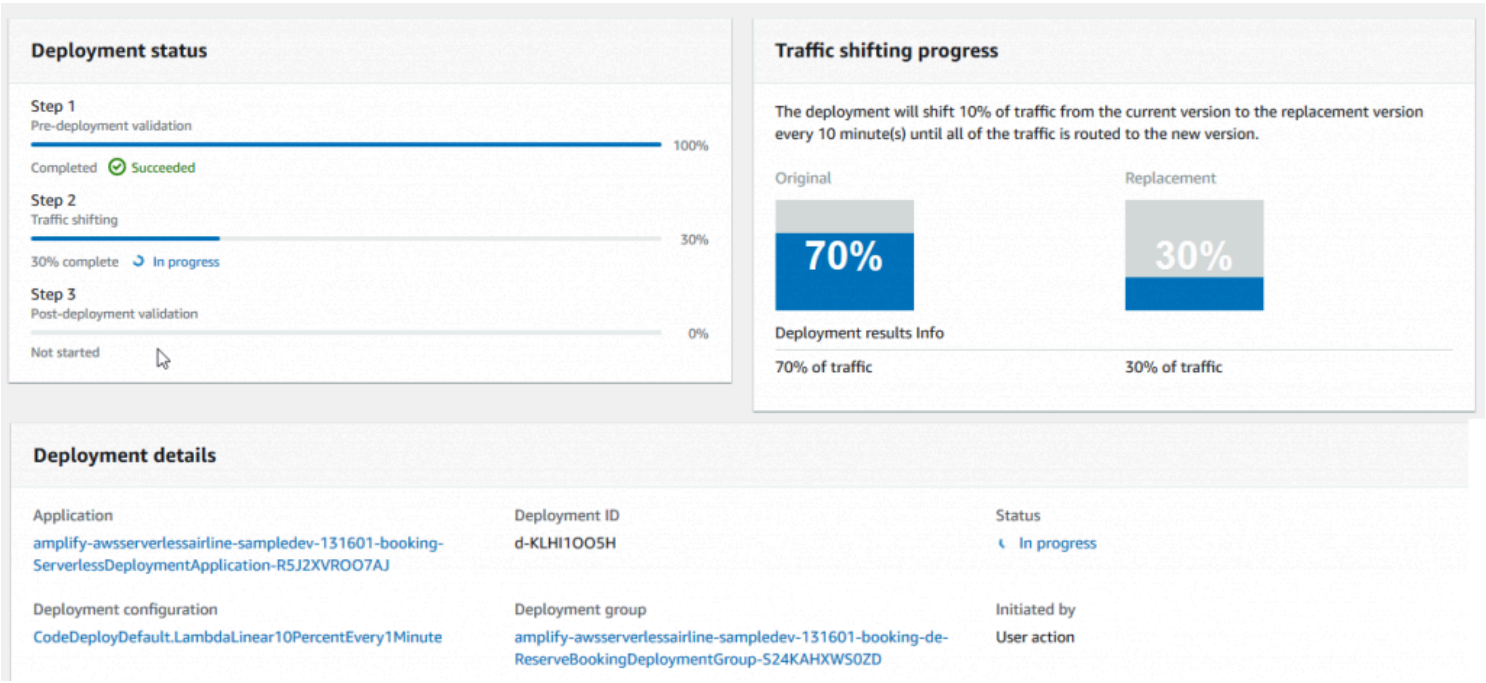
In the airline example, create a safe deployment for the *ReserveBooking* Lambda function by adding the example AWS SAM template code specified in the instructions. This migrates 10 percent of traffic every 10 minutes with CloudWatch alarms to check for any function errors. You could also alarm on latency, or any custom metric.

During the Amplify Console build phase, the safe deployment is initiated. Navigate to the CodeDeploy console and see the deployment in progress.



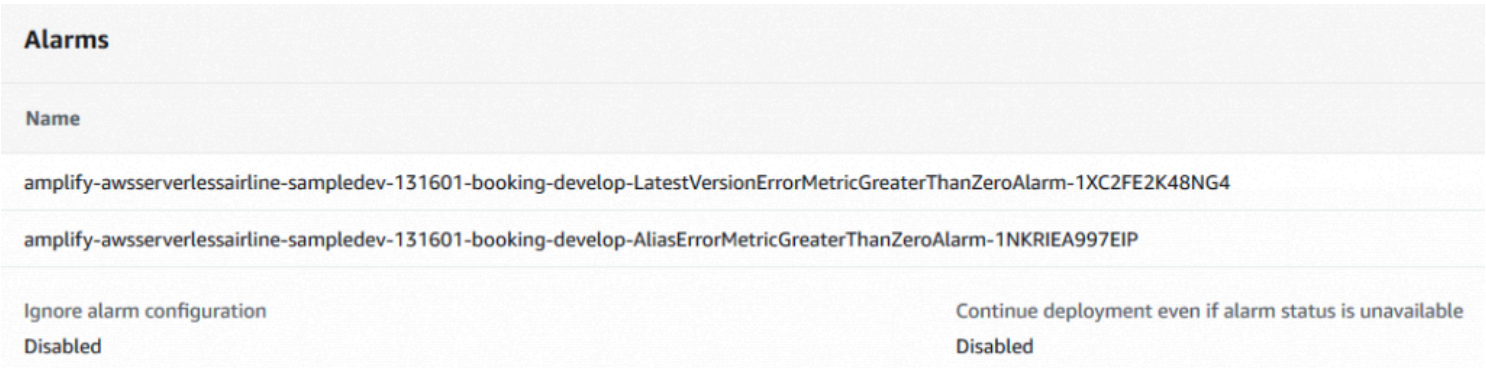
AWS CodeDeploy deployment in progress

Selecting the deployment, you can see the *Traffic shifting progress* and the *Deployment details*.



AWS CodeDeploy traffic shifting in progress.

Within *Deployment details*, select the *DeploymentGroup*, and view the CloudWatch Alarms CodeDeploy is using to test the rollout.



Amazon CloudWatch Alarms AWS CodeDeploy is using to test the rollout

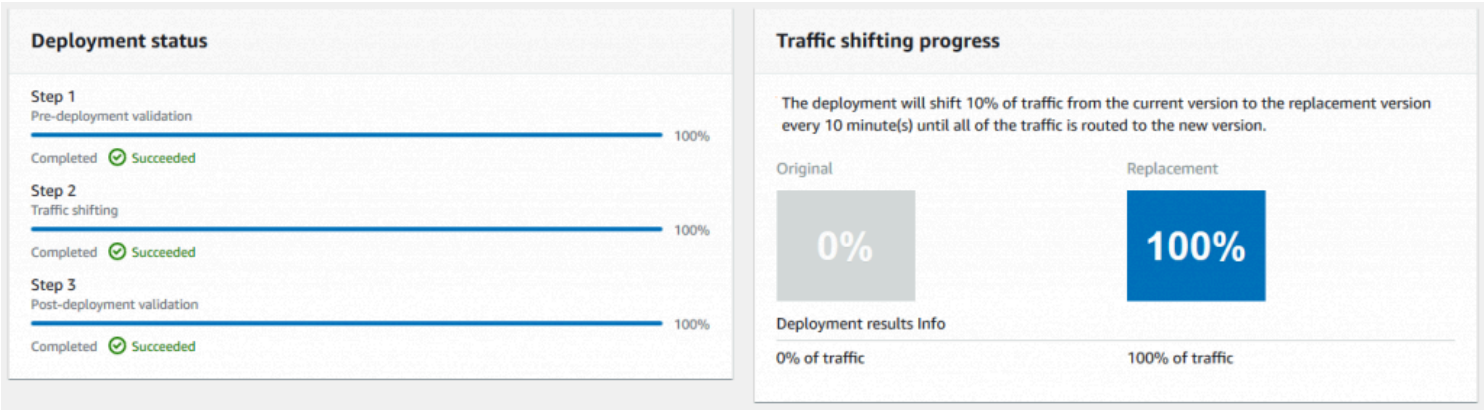
Within *Deployment details*, select the *Application*, select the **Revisions** tab, and select the latest *Revision location* and view the `CurrentVersion` and `TargetVersion` for this deployment.

App spec

```
{
  "version": "0.0",
  "Resources": [
    {
      "Airline-ReserveBooking-develop": {
        "Type": "AWS::Lambda::Function",
        "Properties": {
          "Name": "Airline-ReserveBooking-develop",
          "Alias": "live",
          "CurrentVersion": "3",
          "TargetVersion": "4"
        }
      }
    }
  ]
}
```

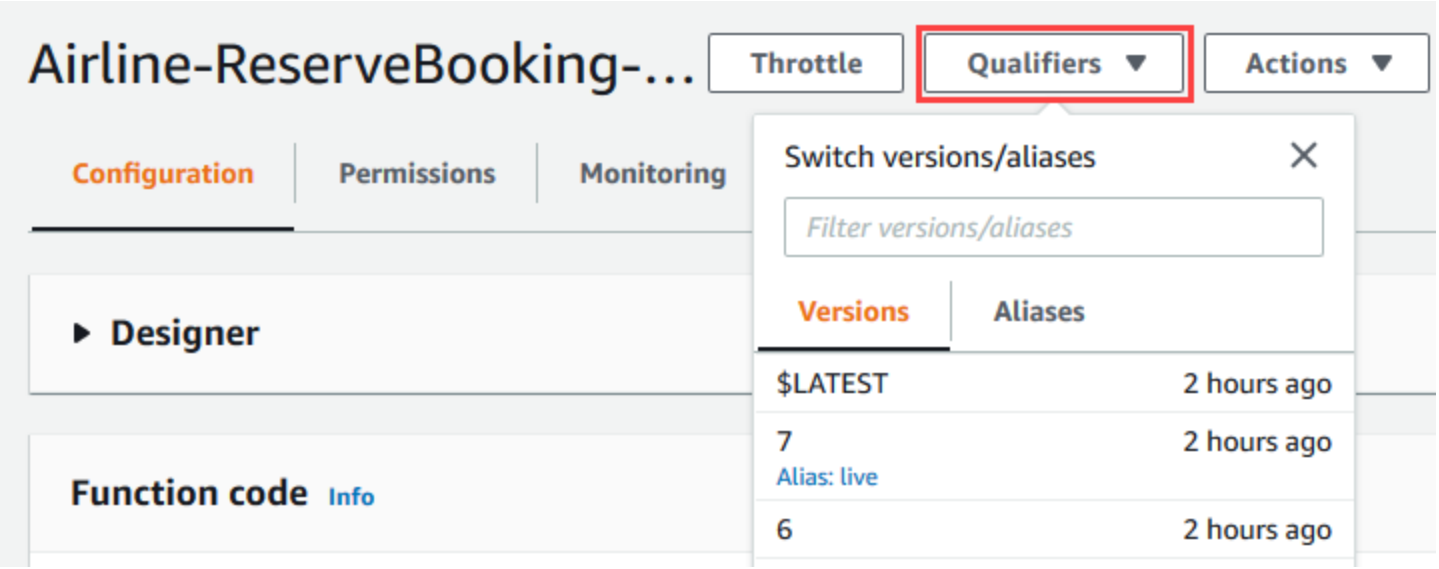
View deployment versions

View *Deployment status* and see the traffic has now shifted to the new version. The Amplify Console build also continues.



Traffic shifting complete

View the Lambda function versions and aliases in the Lambda console, selecting **Qualifiers**.



Viewing Lambda function version and aliases

Amazon API Gateway also supports canary release deployments at the API layer.

A rollout deployment provides traffic shifting, A/B testing, and the ability to roll back to any version at any point in time. AWS SAM makes it simple to add safe deployments to serverless applications.

Improvement plan summary

1. For production systems, use a linear deployment strategy to gradually rollout changes to customers.
2. For high volume production systems, use a canary deployment strategy when you want to limit changes to a fixed percentage of customers for an extended period of time.

Conclusion

Introducing application lifecycle management improves the development, deployment, and management of serverless applications. In this post I cover a number of methods to prototype new features using temporary environments. I show how to use rollout deployments to gradually shift traffic to new application code.

This well-architected question continues in part 3 where I look at configuration management, CI/CD for serverless applications, and managing function runtime deprecation.

TAGS: Amazon CloudWatch, Amazon DynamoDB, Amazon Simple Storage Services, AWS Amplify Console, AWS CloudFormation, AWS CodeDeploy, AWS Lambda, AWS Serverless Application Model, serverless, well-architected