AWS Compute Blog

# Building well-architected serverless applications: Approaching application lifecycle management – part 3

by Julian Wood | on 18 JUN 2020 | in Amazon API Gateway, Amazon Simple Notification Service (SNS), AWS Amplify, AWS CloudFormation, AWS CodeBuild, AWS CodeCommit, AWS CodeDeploy, AWS CodePipeline, AWS Config, AWS Lambda, AWS Personal Health Dashboard, AWS Serverless Application Model, AWS Systems Manager, 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 2 of this Operational Excellence question where I look at deploying to multiple stages using temporary environments, and rollout deployments. In part 1, I cover using infrastructure as code with version control to deploy applications in a repeatable manner.

## Good practice: Use configuration management

Use environment variables and configuration management systems to make and track configuration changes. These systems reduce errors caused by manual processes, reduce the level of effort to deploy changes, and help isolate configuration from business logic.

Environment variables are suited for infrequently changing configuration options such as logging levels, and database connection strings. Configuration management systems are for dynamic configuration that might change frequently or contain sensitive data such as secrets.

### Environment variables

The serverless airline example used in this series uses AWS Amplify Console environment variables to store application-wide settings.

For example, the Stripe payment keys for all branches, and names for individual branches, are visible within the Amplify Console in the *Environment variables* section.

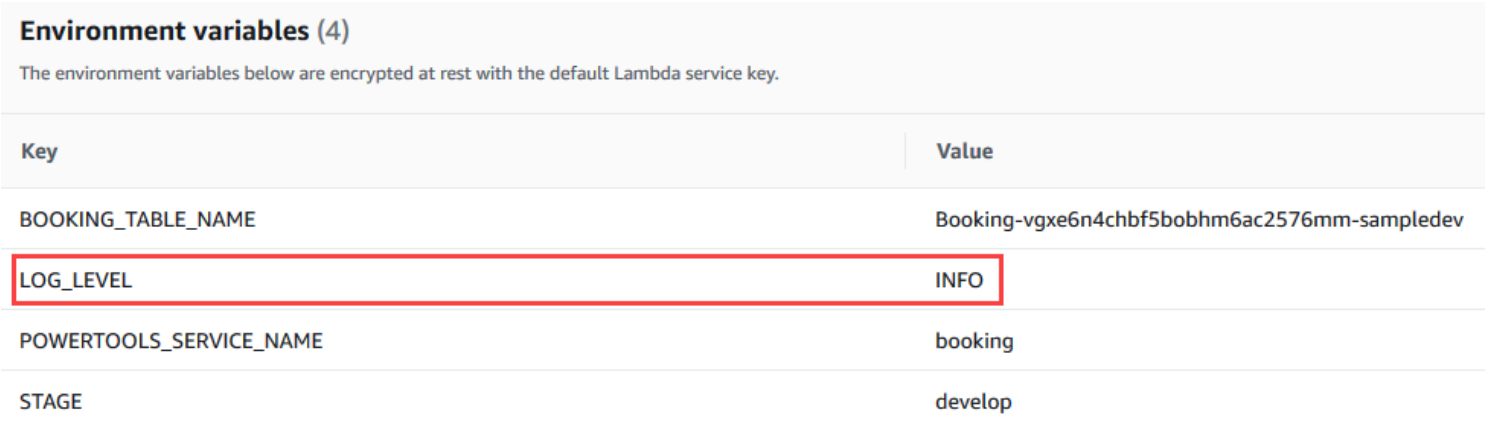

AWS Amplify environment variables

AWS Lambda environment variables are set up as part of the function configuration stored using the AWS Serverless Application Model (AWS SAM).

For example, the airline booking *ReserveBooking* AWS SAM template sets global environment variables including the *LOG_LEVEL* with the following code.

```yaml
Globals:
    Function:
        Environment:
            Variables:
                LOG_LEVEL: INFO
```

This is visible in the AWS Lambda console within the function configuration.




AWS Lambda environment variables in console

See the AWS Documentation for more information on using AWS Lambda environment variables and also how to store sensitive data. Amazon API Gateway can also pass stage-specific metadata to Lambda functions.

## Dynamic configuration

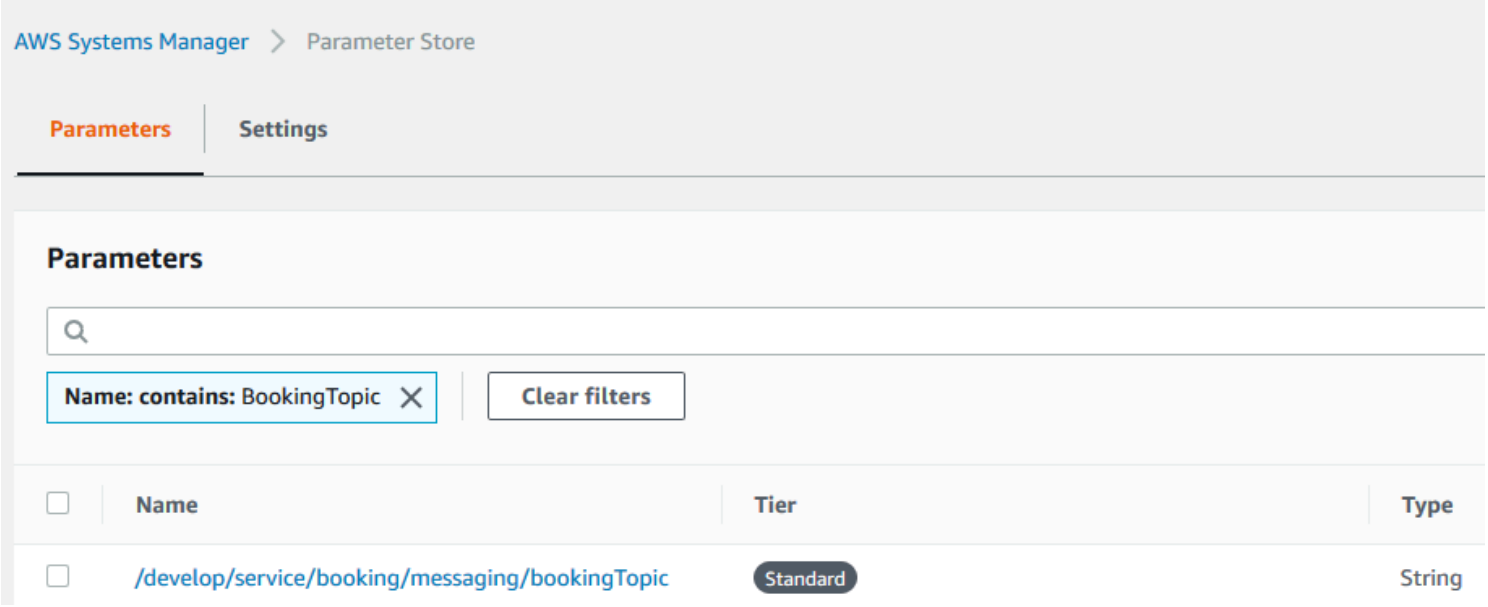
Dynamic configuration is also stored in configuration management systems to specify external values and is unique to each environment. This configuration may include values such as an Amazon Simple Notification Service (Amazon SNS) topic, Lambda function name, or external API credentials. AWS System Manager Parameter Store, AWS Secrets Manager, and AWS AppConfig have native integrations with AWS CloudFormation to store dynamic configuration. For more information, see the examples for referencing dynamic configuration from within AWS CloudFormation.

For the serverless airline application, dynamic configuration is stored in AWS Systems Manager Parameter Store. During CloudFormation stack deployment, a number of parameters are stored in Systems Manager. For example, in the booking service AWS SAM template, the booking SNS topic ARN is stored.

YAML

```yaml
BookingTopicParameter:
    Type: "AWS::SSM::Parameter"
    Properties:
        Name: !Sub /${Stage}/service/booking/messaging/bookingTopic
        Description: Booking SNS Topic ARN
        Type: String
        Value: !Ref BookingTopic
```

View the stored SNS topic value by navigating to the Parameter Store console, and search for *BookingTopic*.




Finding Systems Manager Parameter Store values

Select the Parameter name and see the Amazon SNS ARN.

Value

arn:aws:sns:eu-west-1:[redacted]:amplify-awsserverlessairline-sampledev-131601-booking-develop-BookingTopic-6R6GEK8N2A3N

Viewing SNS topic value

The loyalty service then references this value within another stack.

When the Amplify Console Makefile deploys the loyalty service, it retrieves this value for the booking service from Parameter Store, and references it as a *parameter-override*. The deployment is also parametrized with the `$${AWS_BRANCH}` environment variable if there are multiple environments within the same AWS account and Region.

```
Bash
sam deploy \
    --parameter-overrides \
    BookingSNSTopic=/$${AWS_BRANCH}/service/booking/messaging/bookingTopic
```

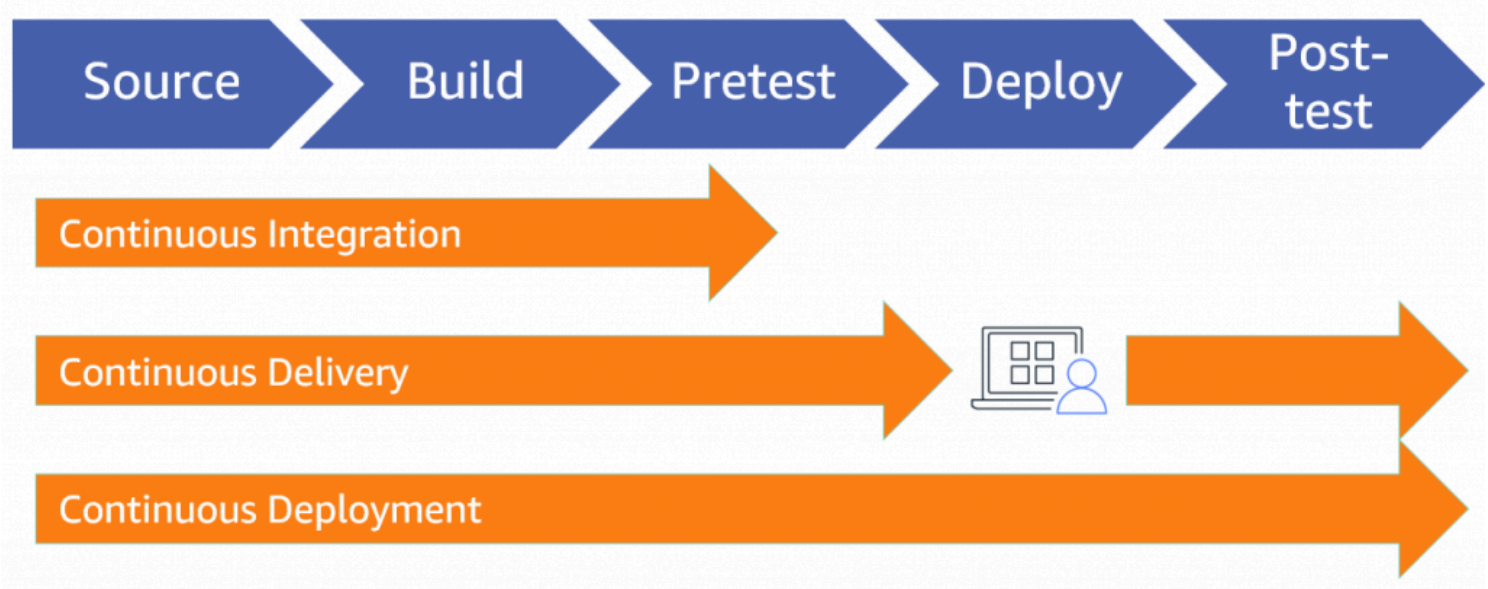
Environment variables and configuration management systems help with managing application configuration.

### Improvement plan summary

1. Use environment variables for configuration options that change infrequently such as logging levels, and database connection strings.
2. Use a configuration management system for dynamic configuration that might change frequently or contain sensitive data such as secrets.

## Best practice: Use CI/CD including automated testing across separate accounts

Continuous integration/delivery/deployment is one of the cornerstones of cloud application development and a vital part of a DevOps initiative.




Explanation of CI/CD stages

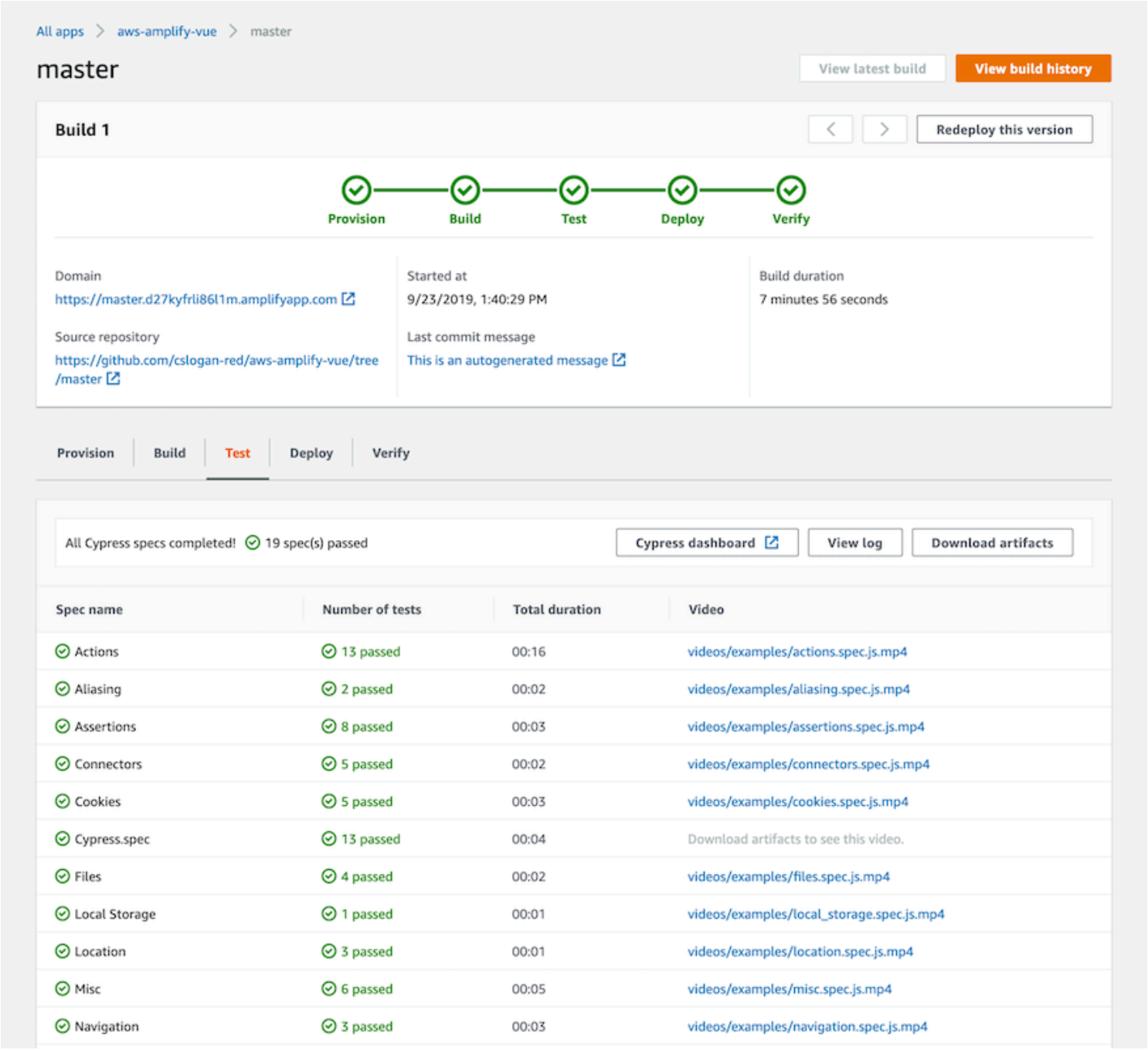
Building CI/CD pipelines increases software delivery quality and feedback time for detecting and resolving errors. I cover how to deploy multiple stages in isolated environments and accounts, which helps with creating separate testing CI/CD pipelines in part 2. As the serverless airline example is using AWS Amplify Console, this comes with a built-in CI/CD pipeline.

Automate the build, deployment, testing, and rollback of the workload using KPI and operational alerts. This eases troubleshooting, enables faster remediation and feedback time, and enables automatic and manual rollback/roll-forward should an alert trigger.

I cover metrics, KPIs, and operational alerts in this series in the Application Health part 1, and part 2 posts. I cover rollout deployments with traffic shifting based on metrics in this question's part 2.

CI/CD pipelines should include integration, and end-to-end tests. I cover local unit testing for Lambda and API Gateway in part 2.

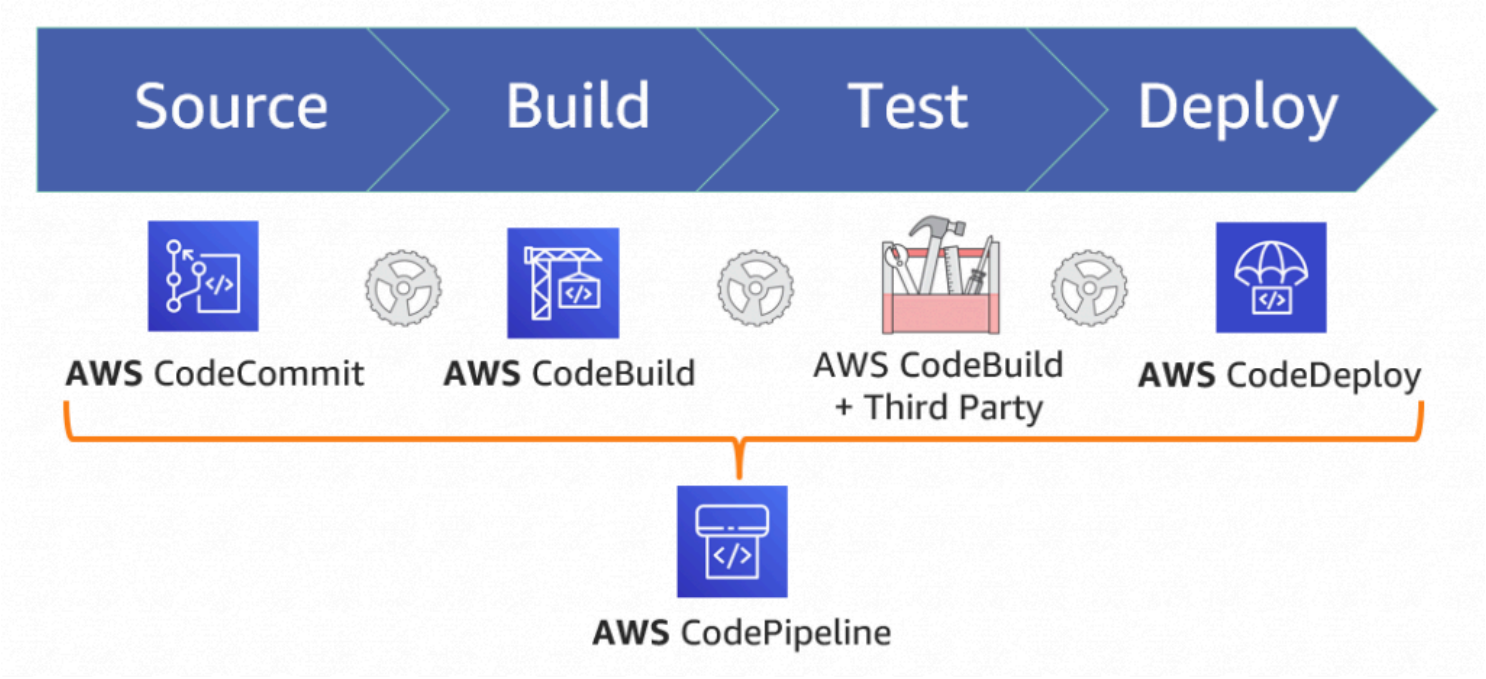
Add an optional testing stage to Amplify Console to catch regressions before pushing code to production. Use the test step to run any test commands at build time using any testing framework of your choice. Amplify Console has deeper integration with the Cypress test suite that allows you to generate a UI report for your tests. Here is an example to set up end-to-end tests with Cypress.




Cypress testing example

There are a number of AWS and third-party solutions to host code and create CI/CD pipelines for serverless applications.

- AWS CodeCommit is a fully managed git-based source control service for hosting code.
- AWS CodeBuild is a fully managed continuous integration service for compiling code, running tests, and producing deployment artifacts.
- AWS CodePipeline is a fully managed continuous delivery service to automate release pipelines.
- AWS CodeDeploy is a fully managed software deployment service to deploy code to a number of compute services such as Lambda.




AWS Code Suite

For more information on how to use the AWS Code* services together, see the detailed Quick Start deployment guide Serverless CI/CD for the Enterprise on AWS.

All these AWS services have a number of integrations with third-party products so you can integrate your serverless applications with your existing tools. For example, CodeBuild can build from GitHub and Atlassian Bitbucket repositories. CodeDeploy integrates with a number of developer tools and configuration management systems. CodePipeline has a number of pre-built integrations to use existing tools for your serverless applications. For more information specifically on using CircleCI for serverless applications, see Simplifying Serverless CI/CD with CircleCI and the AWS Serverless Application Model.

### Improvement plan summary

1. Use a continuous integration/continuous deployment (CI/CD) pipeline solution that deploys multiple stages in isolated environments/accounts.
2. Automate testing including but not limited to unit, integration, and end-to-end tests.
3. Favor rollout deployments over all-at-once deployments for more resilience, and gradually learn what metrics best determine your workload's health to appropriately alert on.
4. Use a deployment system that supports traffic shifting as part of your pipeline, and rollback/roll-forward traffic to previous versions if an alert is triggered.

## Good practice: Review function runtime deprecation policy

Lambda functions created using AWS provided runtimes follow official long-term support deprecation policies. Third-party provided runtime deprecation policy may differ from official long-term support. Review your runtime

deprecation policy and have a mechanism to report on runtimes that, if deprecated, may affect your workload to operate as intended.

Review the AWS Lambda runtime policy support page to understand the deprecation schedule for your runtime.

AWS Health provides ongoing visibility into the state of your AWS resources, services, and accounts. Use the AWS Personal Health Dashboard for a personalized view and automate custom notifications to communication channels other than your AWS Account email.

Use AWS Config to report on AWS Lambda function runtimes that might be near their deprecation. Run compliance and operational checks with AWS Config for Lambda functions.

If you are unable to migrate to newer runtimes within the deprecation schedule, use AWS Lambda custom runtimes as an interim solution.

### Improvement plan summary

1. Identify and report runtimes that might deprecate and their support policy.

## Conclusion

Introducing application lifecycle management improves the development, deployment, and management of serverless applications. In part 1, I cover using infrastructure as code with version control to deploy applications in a repeatable manner. This reduces errors caused by manual processes and gives you more confidence your application works as expected. In part 2, I cover prototyping new features using temporary environments, and rollout deployments to gradually shift traffic to new application code.

In this post I cover configuration management, CI/CD for serverless applications, and managing function runtime deprecation.

In the next post in the series, I cover the third Operational Excellence question from the Well-Architected Serverless Lens – Controlling access to serverless APIs.

TAGS: Amazon API Gateway, Amazon Simple Notification Service, AWS Amplify, AWS CloudFormation, AWS CodeBuild, AWS CodeCommit, AWS CodeDeploy, AWS CodePipeline, AWS Config, AWS Lambda, AWS Personal Health, AWS Serverless Application Model, AWS Systems Manager, serverless, well-architected