View the secret in Secrets Manager from the AWS Management Console

163. In the AWS Management Console, choose Services and then choose **Secrets Manager**.

The **dbUserId** secret you created earlier is displayed here.

164. To view the details of the secret, choose **dbUserId**.

165. In the **Secret value** section, choose **Retrieve secret value** to view the value of the secret.

Test the secret using a Lambda function

The **sam-bookmark-app-secrets-function** function code should be updated to test the secret.

166. In the AWS Management Console, choose Services and then choose **Lambda**.

167. Enter `sam-bookmark-app-secrets-function` into the box, and select this function.

168. In the **Code source** section, add the following to the **index.js** file:

- Add the following constant after the ssmsecret constant (line 5):

```
const userId =  process.env.SM_USER_ID;
```

- Add the following code snippet to the end of the function code:

```
async function decodeSMSecret(smkey) {
    console.log("SM Key:", smkey);
    const params = {
        SecretId: smkey
    };
    const result = await sm.getSecretValue(params).promise();
    return result.SecretString;
}
```

The code should look like the following code after you add the two previous snippets:

```
const aws = require('aws-sdk');

const kmsSecret = process.env.KMS_SECRET;
const ssmSecret = process.env.SSM_SECRET;
const userId =  process.env.SM_USER_ID;

let decodedSecret;
let DecodedKMSSecret;

const kms = new aws.KMS();
const ssm = new aws.SSM();
const sm = new aws.SecretsManager();

exports.handler = async message => {
```

```
    console.log(message);
    let secretType = message.pathParameters.id
    console.log("Secret Type:", secretType);

    if(secretType == 'kms')
        decodedSecret = await decodeKMSSecret();
    else if (secretType == 'ssm')
        decodedSecret = await decodeSSMSecret();
    else if (secretType == 'sm') {
        var password = await decodeSMSecret(userId);
        decodedSecret = "Password is: " + password;
    }
    else
        decodedSecret = "Provide a valid secret type (kms, ssm, or sm (secrets
manager))";

    console.log(decodedSecret);
    const response = {
        statusCode: 200,
        headers: {},
        body: JSON.stringify('Plain text secret(s): ' + decodedSecret)
    };
    return response;
};

async function decodeKMSSecret() {
    if (DecodedKMSSecret) {
        return DecodedKMSSecret;
    }
    const params = {
      CiphertextBlob: Buffer.from(kmsSecret, 'base64')
    };
    const data = await kms.decrypt(params).promise();
    DecodedKMSSecret = data.Plaintext.toString('utf-8');
    return DecodedKMSSecret;
}

async function decodeSSMSecret() {
    const params = {
        Name: ssmSecret,
        WithDecryption: true
    };
    const result = await ssm.getParameter(params).promise();
    return result.Parameter.Value
}

async function decodeSMSecret(smkey) {
    console.log("SM Key:", smkey);
    const params = {
        SecretId: smkey
    };
    const result = await sm.getSecretValue(params).promise();
    return result.SecretString;
}
```

169. Choose Deploy

You should see a message that says
Successfully updated the function sam-bookmark-app-secrets-function.

170. Choose the **Configuration** tab to configure the environment variables.

171. In the left navigation pane, choose **Environment variables**.

172. In the **Environment variables** section, choose Edit

173. In the **Edit environment variables** page, choose Add environment variable and configure the following details:

- **Key:** Enter `SM_USER_ID`
- **Value:** Enter `dbUserId`

174. Choose Save

The Lambda function has been updated to read Secrets Manager and display the password.

Note There is an IAM permission, **secretsmanager:GetSecretValue**, needed for the Lambda function to read Secrets Manager. This permission has been added to the **SamDeploymentRole** in the pre-build process of the lab.

175. In the **Invoke URL** you saved from task 3.1, replace *{id}* with `sm`

176. Copy and paste the updated **Invoke URL** into a new browser tab, and press **Enter**.

You should see the following text in the browser.

```
"Plain text secret(s): Password is: secretsmanagerpassword"
```

Conclusion

Congratulations! You have successfully:

- Secured your application with AWS WAF web ACLs
- Secured access to your API with an API Gateway resource policy
- Secured your Lambda functions and other backend services with AWS KMS, Systems Manager Parameter Store, and Secrets Manager

End lab

Follow these steps to close the console, end your lab, and evaluate your lab experience.

177. Return to the **AWS Management Console**.

178. At the upper-right corner of the page, choose **awsstudent@<AccountNumber>**, and then choose **Sign out**.

179. Choose End Lab.

180. Choose Submit.

181. (Optional):

- Select the applicable number of stars to rate your lab experience.
  - 1 star = Very dissatisfied
  - 2 stars = Dissatisfied
  - 3 stars = Neutral
  - 4 stars = Satisfied
  - 5 stars = Very satisfied
- Enter a comment.
- Choose **Submit**.

You can close the window if you don't want to provide feedback.

For more information about AWS Training and Certification, see *https://aws.amazon.com/training/*.

Your feedback is welcome and appreciated.

If you would like to share any feedback, suggestions, or corrections, please provide the details in our *AWS Training and Certification Contact Form*.

Additional resources

- For more information on AWS WAF, see Using AWS WAF to control access.
- For more information on Secrets Manager, see AWS Secrets Manager user guide.
- For more information on AWS KMS, see AWS KMS Features.
- For more information on Systems Manager, see Sharing secrets with AWS Lambda using AWS Systems Manager

aws training and certification

Lab 6: Serverless CI/CD on AWS

Note: Do not include any personal, identifying, or confidential information into the lab environment. Information entered may be visible to others.

Corrections, feedback, or other questions? Contact us at *AWS Training and Certification*.

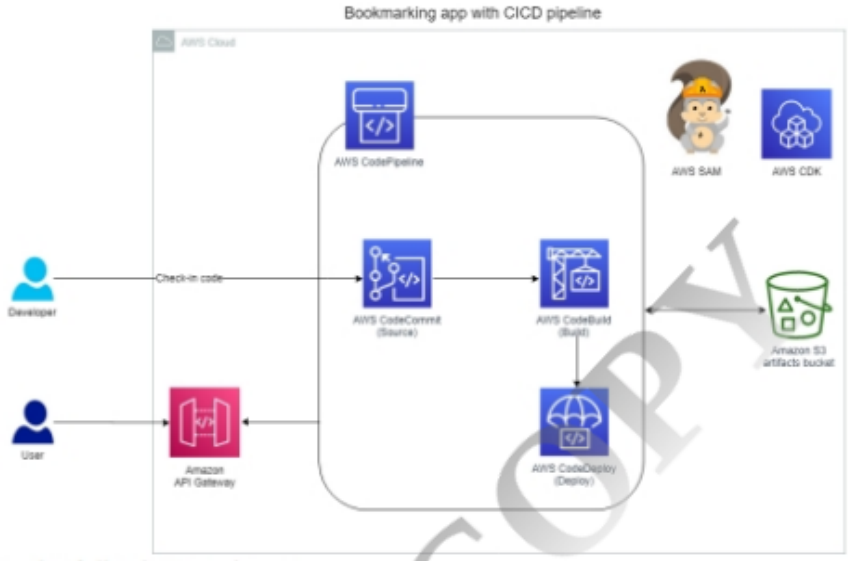
Overview

Now you have a fully functional serverless application that allows users to sign in, save bookmarks for their own use, and submit bookmarks to be shared in the team knowledge base. Several tasks happen in parallel when a new bookmark is submitted for the knowledge base: The submitter's entry is entered into a contest, notifications are sent, and an automated publishing workflow is initiated. You have also incorporated logging and tracing, which will give you operational visibility into the production application. Finally, you incorporated security at all layers to protect your application.

As you have learned, a key part of developing serverless applications is to use your visibility into how the application is used in production to determine where to make future modifications. You should continue to monitor the production application to understand access patterns, resolve operational errors, and iteratively optimize your application to minimize costs and continually improve the user experience. Because your application uses small decoupled components, you can more easily modify components independently.

To support these types of small, frequent updates, you would like to automate the build and deployment processes so that the development team continues to focus on developing the application. In this lab, you learn how to set up a continuous integration and continuous delivery (CI/CD) pipeline and do canary deployments on AWS.

The following diagram shows the architecture components that have been or will be deployed in this lab.



This lab uses the following services:

- AWS Serverless Application Model (AWS SAM)
- AWS Cloud9
- Amazon DynamoDB
- Amazon EventBridge
- Amazon Simple Notification Service (Amazon SNS)
- AWS Step Functions
- AWS CodeCommit
- AWS CodePipeline
- AWS CodeBuild
- AWS CodeDeploy
- AWS Cloud Development Kit (AWS CDK)

Objectives

After completing this lab, you will be able to:

- Create a CodeCommit repository and a CI/CD pipeline
- Use AWS SAM to define the resources that your application needs and the AWS CDK to define the resources for the deployment infrastructure
- Implement canary deployments using AWS SAM

- Monitor your canary deployment with CodeDeploy

Prerequisites

This lab requires:

- Access to a notebook computer with Wi-Fi and Microsoft Windows, macOS, or Linux (Ubuntu, SUSE, or Red Hat)
- For Microsoft Windows users, administrator access to the computer
- An internet browser such as Chrome, Firefox, or Internet Explorer 9 (previous versions of Internet Explorer are not supported)
- A text editor

⚠ **Note** The lab environment is not accessible using an iPad or tablet device, but you can use these devices to access the lab guide.

Duration

This lab requires approximately **90 minutes** to complete.

Start lab

1. To launch the lab, at the top of the page, choose Start Lab.

This starts the process of provisioning the lab resources. An estimated amount of time to provision the lab resources is displayed. You must wait for the resources to be provisioned before continuing.

ⓘ If you are prompted for a token, use the one distributed to you (or credits you have purchased).

2. To open the lab, choose Open Console.

The **AWS Management Console** sign-in page opens in a new web browser tab.

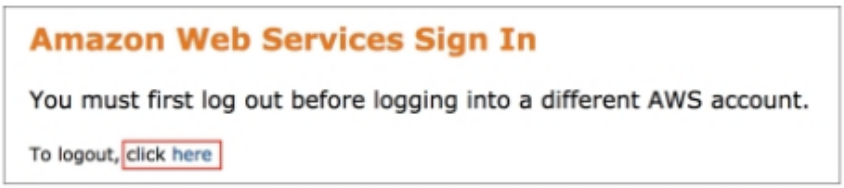
3. On the **Sign in as IAM user** page:

- For **IAM user name**, enter `awsstudent`.
- For **Password**, copy and paste the **Password** value listed to the left of these instructions.
- Choose Sign in.

⚠ **Do not change the Region unless instructed.**

Common sign-in errors

Error: You must first sign out



If you see the message, **You must first log out before logging into a different AWS account:**

- Choose the **click here** link.
- Close your **Amazon Web Services Sign In** web browser tab and return to your initial lab page.
- Choose Open Console again.

In some cases, certain pop-up or script blocker web browser extensions might prevent the **Start Lab** button from working as intended. If you experience an issue starting the lab:

- Add the lab domain name to your pop-up or script blocker's allow list or turn it off.
- Refresh the page and try again.

Task 1: Understanding key services and setting up the project

In this task, you look at the different services that you use in this lab. You then download the application code using the AWS Cloud9 integrated development environment (IDE), unzip the bookmark application, and inspect the source code in the AWS Cloud9 IDE.

- **AWS CodeCommit** is a fully managed source control service that hosts secure Git-based repositories. The service makes it easy for teams to collaborate on code in a secure and highly scalable ecosystem. CodeCommit eliminates the need to operate your own source control system or worry about scaling its infrastructure. You can use CodeCommit to securely store anything from source code to binaries, and the service works seamlessly with your existing Git tools.
- **AWS CodePipeline** is a fully managed continuous delivery service that helps you automate your release pipelines for fast and reliable application and infrastructure updates. CodePipeline automates the build, test, and deploy phases of your release process every time there is a code change based on the release model that you define. This enables you to rapidly and reliably deliver features and updates. You can easily integrate CodePipeline with third-party services such as GitHub or with your own custom plugin. With CodePipeline, you pay for only what you use. There are no upfront fees or long-term commitments.
- **AWS CodeBuild** is a fully managed continuous integration service that compiles source code, runs tests, and produces software packages that are ready to deploy. With CodeBuild, you don't need to provision, manage, or scale your own build servers. CodeBuild scales continuously and processes multiple builds concurrently, so your builds are not left waiting in a queue. You can get started quickly by using prepackaged build environments, or you can create custom build environments that use your own build tools. With CodeBuild, you are charged by the minute for the compute resources that you use.
- **AWS CodeDeploy** is a fully managed deployment service that automates software deployments to a variety of compute services such as Amazon Elastic Compute Cloud (Amazon EC2), AWS Fargate, AWS Lambda, and your on-premises servers. CodeDeploy makes it easier for you to rapidly release new features, helps you avoid downtime during application deployment, and handles the complexity of updating your applications. You can use CodeDeploy to automate software deployments, which eliminates the need for error-prone manual operations. The service scales to match your deployment needs.

Set up the application using AWS Cloud9

4. In the AWS Management Console, choose Services and select **Cloud9**.

You will use the AWS Cloud9 terminal throughout this lab.

5. On the left side of the page, choose the menu ☰ icon to expand the menu, and choose **Your environments**.

If the menu is already expanded, move on to the next step.

6. For the **BookmarkAppDevEnv** environment, choose Open IDE

Within a few seconds, the AWS Cloud9 environment launches.

Note If the browser is running in an incognito session, a pop-up window with an error message will be displayed when the Cloud9 instance is opened. Choose the **OK** button to continue. Browser in a non incognito mode is recommended.

7. In the AWS Cloud9 terminal, run the following commands to download the application code, unzip the contents, and add more space:

```
wget https://us-west-2-tcprod.s3-us-west-2.amazonaws.com/courses/ILT-TF-200-
SVDVSS/v1.0.29/lab-6-CICD/scripts/app-code.zip
unzip app-code.zip
cd app-code
chmod +x resize.sh
bash resize.sh 20
```

Inspect the source code in AWS Cloud9

8. In the left navigation pane of the AWS Cloud9 terminal, choose the arrow next to the **app-code** folder to expand it.

9. In the **app-code** folder, choose the arrow next to the **backend** folder to expand it.

Inspect the Lambda functions and the AWS SAM template that has been created.

10. In the **backend** folder, open the **template.yaml** file.

In the **Parameters** section toward the end of the file, the parameters *LAMBDA_ROLE_ARN* and *STEP_FUNCTIONS_ROLE_ARN* should be replaced with the actual values.

11. In the AWS Cloud9 terminal, run the following commands to replace the values:

```
cd backend
export LAMBDA_ROLE_ARN=$(aws iam  list-roles --query "Roles[?contains(RoleName,
'LambdaDeployment')].Arn" --output text)
sed -Ei "s|<LAMBDA_ROLE_ARN>|${LAMBDA_ROLE_ARN}|g" template.yaml
export STEP_FUNCTIONS_ROLE_ARN=$(aws iam  list-roles --query
"Roles[?contains(RoleName, 'StateMachine')].Arn" --output text)
sed -Ei "s|<STEP_FUNCTIONS_ROLE_ARN>|${STEP_FUNCTIONS_ROLE_ARN}|g" template.yaml
cd ..
```

Note The **template.yaml** file also contains **${AWS::AccountId}**. Do not replace this value with the **AWS Account** information. The **LambdaDeploymentRole** is used when the AWS SAM template is automatically deployed through the CI/CD process.

12. Choose **File > Save** to save your changes to the **template.yaml** file.

Task 2: Checking the application source code in to the CodeCommit repo

In this task, you create a repository (repo) and check in the source code. You could use any source repo (for example, GitHub), but for this lab, use CodeCommit.

13. In the AWS Cloud9 terminal, run the following command to create a new CodeCommit repository:

```
aws codecommit create-repository --repository-name app-code
```

The terminal displays the following output:

```
{
    "repositoryMetadata": {
        "accountId": "${AccountId}",
        "repositoryId": "xxxxxxx",
        "repositoryName": "app-code",
        "lastModifiedDate": 1603210225.175,
        "creationDate": 1603210225.175,
        "cloneUrlHttp": "https://git-codecommit.us-west-2.amazonaws.com/v1/repos/app-
code",
        "cloneUrlSsh": "ssh://git-codecommit.us-west-2.amazonaws.com/v1/repos/app-
code",
        "Arn": "arn:aws:codecommit:us-west-2:${AccountId}:app-code"
    }
}
```

CodeCommit supports AWS Identity and Access Management (IAM) authentication, and because you are running this from an AWS Cloud9 workspace, you can leverage the fact that your terminal is already pre-authenticated with valid AWS credentials.

14. In the AWS Cloud9 terminal, run the following commands:

```
git config --global credential.helper '!aws codecommit credential-helper $@'
git config --global credential.UseHttpPath true
```

Note These commands specify the use of the Git credential helper with the AWS credential profile and enable the Git credential helper to send the path to repositories. The credential helper uses the default AWS credential profile or the Amazon EC2 instance role.

15. In the AWS Cloud9 terminal, run the following commands to do an initial commit of the code:

```
cd ~/environment/app-code
git init
git checkout -b main
git add .
git commit -m "Initial commit"
```

Push the code

16. In the AWS Cloud9 terminal, run the following commands to fetch the *AWS_REGION* and substitute it in the git command.

```
sudo yum -y install jq
echo export AWS_REGION=$(curl -s 169.254.169.254/latest/dynamic/instance-
identity/document | jq -r '.region') >> ~/environment/app-code/labVariables
source ~/environment/app-code/labVariables
git remote add origin https://git-codecommit.${AWS_REGION}.amazonaws.com/v1/repos/app-
code
```

Note This code adds your CodeCommit repository URL as a remote on your local git project.

17. To push the code to CodeCommit, run the following command:

```
git push -u origin main
```

Verify in CodeCommit

18. In the AWS Management Console, choose Services and select CodeCommit.

19. In the CodeCommit console, under **Repositories**, choose the **app-code** repository to view its contents.

Task 3: Building the CI/CD pipeline

In this task, you learn how to automate the bookmark application build process and deployment by creating a pipeline using CodePipeline.

Introducing the AWS CDK

You use the AWS CDK as the pipeline vending mechanism in this lab. The AWS CDK is a software development framework for defining cloud infrastructure in code and provisioning it through AWS CloudFormation.

You can describe your infrastructure by writing code in TypeScript, C#, Python, or Java. Your code is then synthesized into CloudFormation templates and, by using the AWS CDK CLI, can then be deployed into your AWS environment.

Understand how AWS SAM and the AWS CDK work together

Serverless developers use the AWS SAM framework to define their applications, the AWS SAM CLI to build and deploy them, and the AWS CDK to provision any infrastructure-related resources, such as the CI/CD pipeline. All of these tools share one underlying service: CloudFormation.

20. In the AWS Cloud9 terminal, run the following commands to uninstall any older versions of the AWS CDK and install the latest version:

```
npm uninstall -g aws-cdk
npm install -g aws-cdk@latest --force
```

21. Run the following commands to create a folder outside of the **app-code** directory where the **pipeline** code will reside:

```
cd ~/environment
mkdir pipeline
cd pipeline
```

22. Run the following command to initialize a new AWS CDK project within the **pipeline** folder:

```
npx aws-cdk@1.x init app --language typescript
```

23. Run the following commands to install the AWS CDK modules that are used to build the pipeline:

```
git checkout -b main
npm install --save @aws-cdk/aws-codedeploy @aws-cdk/aws-codebuild
npm install --save @aws-cdk/aws-codecommit @aws-cdk/aws-codepipeline-actions
npm install --save @aws-cdk/aws-s3
```

Project structure

After a few seconds, the project should have the following structure, which shows only the most relevant files and folders. Within the AWS CDK project, the main file that you interact with is **pipeline-stack.ts**.

```
app-code                        # SAM application root
├── backend
    ├── src                     # Lambda functions
    └──template.yaml            # SAM template
└── test
    ├── fake-bookmark.js         # Lambda code
    ├── simple-get.yaml
    ├── simple-post.yaml
pipeline                    # CDK project root
    └── bin
        └── pipeline.ts         # Entry point for CDK project
    └── lib
        └── pipeline-stack.ts   # Pipeline definition
    ├── cdk.json
    ├── jest.config.js
    ├── package.json
    └── tsconfig.json
```

24. In the AWS Cloud9 IDE, choose the arrow next to the **pipeline** folder to open it, and then choose the arrow next to the **bin** folder to open it.

25. In the **bin** folder, open the **pipeline.ts** file.

This file is the entry point to the AWS CDK project.

26. In the **pipeline.ts** file, find the code on line 7 that reads **(app, 'PipelineStack');** and replace *PipelineStack* with the following:

```
bookmark-app-cicd
```

Note Leave other instances of *PipelineStack* as is in the **pipeline.ts** file.

27. Choose **File > Save** to save your changes to the **pipeline.ts** file.

Pipeline as code

28. In your AWS Cloud9 workspace, under the **pipeline** folder, choose the arrow next to the **lib** folder to open it.

29. In the **lib** folder, open the **pipeline-stack.ts** file.

Note You will add code to this file later to build the CI/CD pipeline.

30. In the AWS Cloud9 terminal, run the following commands to build the AWS CDK project:

```
cd ~/environment/pipeline
npm run build
```

31. Run the following command to deploy the pipeline project by using the AWS CDK CLI:

```
cdk deploy
```

A new cloud stack has been created in your account: **bookmark-app-cicd**.

32. In the AWS Management Console, choose Services and select **CloudFormation**.

33. Verify that the new **bookmark-app-cicd** cloud stack is deployed without any errors.

Task 4: Creating stages

In this task, you build the artifacts bucket and add the source stage, build stage, and deploy stage to your pipeline.

Artifacts bucket

Each CodePipeline needs an artifacts bucket, also known as an artifact store. CodePipeline uses this bucket to pass artifacts to the downstream jobs, and it's also where AWS SAM uploads the artifacts during the build process.

34. In the AWS Cloud9 workspace, find the **pipeline-stack.ts** file that you opened earlier, and replace the entire code snippet in the file with the following code:

```
// lib/pipeline-stack.ts

import * as cdk from '@aws-cdk/core';
import s3 = require('@aws-cdk/aws-s3');
import codecommit = require('@aws-cdk/aws-codecommit');
import codepipeline = require('@aws-cdk/aws-codepipeline');
import codepipeline_actions = require('@aws-cdk/aws-codepipeline-actions');
import codebuild = require('@aws-cdk/aws-codebuild');

export class PipelineStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // The code that defines your stack goes here
    const artifactsBucket = new s3.Bucket(this, "ArtifactsBucket");
  }
}
```

Note This code creates the artifacts bucket. To correct the indentation in the file, inside the Cloud 9 IDE, select **Edit > Code Formatting > Apply Code Formatting**, or use the keyboard shortcut CTRL + SHIFT + B.

35. Save the file.

36. In the AWS Cloud9 terminal, run the following commands to build and deploy the project:

```
npm run build
cdk deploy
```

Note If you get a build error, make sure that all of the @aws-cdk dependencies in the **package.json** file have the same version number. If not, fix the version numbers, delete the node_modules, and run npm install.

37. In the AWS Management Console, choose Services and select **CloudFormation**.

38. Verify the updated **bookmark-app-cicd** cloud stack. The stack will have a status of **UPDATE_COMPLETE**. An Amazon Simple Storage Service (Amazon S3) bucket has been created.

Source stage

The source stage is the first step of any CI/CD pipeline, and it represents your source code. This stage is in charge of initiating the pipeline based on new code changes (that is, Git push or pull requests). In this section of the lab, use CodeCommit as the source provider, but CodePipeline also supports Amazon S3, GitHub, and Amazon Elastic Container Registry (Amazon ECR) as source providers.

39. In the **pipeline-stack.ts** file, append the following code snippet after your bucket definition at line 16 inside the constructor:


```
// Import existing CodeCommit app-code repository
const codeRepo = codecommit.Repository.fromRepositoryName(
  this,
  'AppRepository', // Logical name within CloudFormation
  'app-code' // Repository name
);

// Pipeline creation starts
const pipeline = new codepipeline.Pipeline(this, 'Pipeline', {
  artifactBucket: artifactsBucket
});

// Declare source code as an artifact
const sourceOutput = new codepipeline.Artifact();

// Add source stage to pipeline
pipeline.addStage({
  stageName: 'Source',
  actions: [
    new codepipeline_actions.CodeCommitSourceAction({
      actionName: 'CodeCommit_Source',
      repository: codeRepo,
      output: sourceOutput,
      branch: 'main',
    }),
  ],
});
```

Note Because you already created the CodeCommit repository, you do not need to create a new one; rather, you need to import it using the repository name. To correct the indentation in the file, inside the Cloud 9 IDE, select **Edit > Code Formatting > Apply Code Formatting**, or use the keyboard shortcut CTRL + SHIFT + B.

Also notice how a **sourceOutput** object is defined as a pipeline artifact. This is necessary for any files that you want CodePipeline to pass to downstream stages. In this case, the source code should be passed to the build stage.

40. Save the file.

Build stage

The build stage is where AWS SAM builds and packages your serverless application. Use CodeBuild as the build provider for your pipeline.

CodeBuild is a great option because you pay for only the time when your build is running, which makes it cost-effective compared to running a dedicated build server 24 hours a day. The service is also container based, which means that you can bring your own Docker container image where your build runs or use a managed image that CodeBuild provides.

41. In the **pipeline-stack.ts** file, append the following code snippet after the source stage definition in the constructor at line 43 to add a build stage to the file:

```
// Declare build output as artifacts
const buildOutput = new codepipeline.Artifact();

// Declare a new CodeBuild project
```



```
const buildProject = new codebuild.PipelineProject(this, 'Build', {
  environment: { buildImage: codebuild.LinuxBuildImage.AMAZON_LINUX_2_2 },
  environmentVariables: {
    'PACKAGE_BUCKET': {
      value: artifactsBucket.bucketName
    }
  }
});

// Add the build stage to our pipeline
pipeline.addStage({
  stageName: 'Build',
  actions: [
    new codepipeline_actions.CodeBuildAction({
      actionName: 'Build',
      project: buildProject,
      input: sourceOutput,
      outputs: [buildOutput],
    }),
  ],
});
```

Note To correct the indentation in the file, inside the Cloud 9 IDE, select **Edit > Code Formatting > Apply Code Formatting**, or use the keyboard shortcut CTRL + SHIFT + B.

42. Save the file.

43. In the AWS Cloud9 terminal, run the following commands to deploy the pipeline:

```
npm run build
cdk deploy
```

Note The CLI will ask you to confirm the changes before deploying. This occurs because you are giving admin permissions to the IAM role that deploys the application. This is generally not a bad practice because only CloudFormation—not a user—can assume this role. However, if your organization has a stricter security posture, you may want to consider creating a custom IAM deployment role with a fine-grained policy.

44. When the message **Do you wish to deploy these changes (y/n)?** appears, enter **y** and press ENTER.

Note Wait for the above step to complete, before proceeding to the next step. This might take couple of minutes.

45. In the AWS Management Console, choose Services and select **CodePipeline**.

46. Choose the newly created pipeline with **bookmark** in the name.

The build step should have failed. This is expected because you haven't specified what commands to run during the build yet, so CodeBuild doesn't know how to build the serverless application.

To fix this issue, you need to build the **buildspec** file. A **buildspec** file is a series of commands in YAML format that CodeBuild runs to build your application. This file is placed in the root folder of an AWS SAM application, and CodeBuild automatically finds it and runs it during build time.

47. In the AWS Cloud9 workspace, to create the **buildspec** file, open the context (right-click) menu for the **app-code** folder, and select **New File**. Name the file as `buildspec.yml`.

Note The extension of the file can be either .yml or .yaml, and CodeBuild finds it either way.

48. Copy and paste the following content into the **buildspec.yml** file:

```
# ~/environment/app-code/buildspec.yml

version: 0.2
phases:
  install:
    runtime-versions:
      nodejs: 12
    commands:
      # Install packages or any pre-reqs in this phase.
      # Upgrading SAM CLI to latest version
      - pip3 install --upgrade aws-sam-cli
      - sam --version

  build:
    commands:
      # Use Build phase to build your artifacts (compile, etc.)
      - cd backend
      - sam build

  post_build:
    commands:
      # Use Post-Build for notifications, git tags, upload artifacts to S3
      - cd ..
      - sam package --template backend/template.yaml --s3-bucket $PACKAGE_BUCKET --
output-template-file packaged.yaml

artifacts:
  discard-paths: yes
  files:
    # List of local artifacts that will be passed down the pipeline
    - packaged.yaml
```

49. Save the file.

Note Take a moment to understand the structure of the file. For more information, see Build Specification Reference for CodeBuild for more information.

Examine the commands in the **buildspec.yml** file:

- The sam build command is used to build the AWS SAM app.
- The sam build command iterates through the functions in the application, looking for the manifest file (such as **requirements.txt** or **package.json**) that contains the dependencies and automatically creates deployment artifacts.
- The sam package command packages an AWS SAM application.

- The sam package command creates a ZIP file of your code and dependencies, and uploads it to Amazon S3.
- The sam package command then returns a copy of your AWS SAM template, replacing references to local artifacts with the Amazon S3 location where the command uploaded the artifacts.

50. In the AWS Cloud9 terminal, run the following commands to commit your changes and push them to the repository:

```
cd ~/environment/app-code
git add .
git commit -m "Added buildspec.yml"
git push
```

Deploy stage

The deploy stage is where your AWS SAM application and all of its resources are created in an AWS account. The most common way to do this is by using CloudFormation ChangeSets to deploy. This means that this stage has two actions: CreateChangeSet and Deploy.

51. In the **pipeline-stack.ts** file, append the following code snippet after the build stage definition in the constructor at line 68 to add a deploy stage to the file:

```
// Deploy stage
pipeline.addStage({
  stageName: 'Dev',
  actions: [
    new codepipeline_actions.CloudFormationCreateReplaceChangeSetAction({
      actionName: 'CreateChangeSet',
      templatePath: buildOutput.atPath("packaged.yaml"),
      stackName: 'bookmark-app',
      adminPermissions: true,
      changeSetName: 'bookmark-app-dev-changeset',
      runOrder: 1
    }),
    new codepipeline_actions.CloudFormationExecuteChangeSetAction({
      actionName: 'Deploy',
      stackName: 'bookmark-app',
      changeSetName: 'bookmark-app-dev-changeset',
      runOrder: 2
    }),
  ],
});
```

Note To correct the indentation in the file in the following steps, inside the Cloud 9 IDE, select **Edit > Code Formatting > Apply Code Formatting**, or use the keyboard shortcut CTRL + SHIFT + B.

52. Save the file.

53. In the AWS Cloud9 terminal, run the following commands from within the pipeline directory:

```
cd ~/environment/pipeline
npm run build
cdk deploy
```

Note The CLI asks you to confirm the changes before deploying because you are giving admin permissions to the IAM role that deploys the application.

54. When the message **Do you wish to deploy these changes (y/n)?** appears, enter **y** and press ENTER.

Note Wait for the above step to complete, before proceeding to the next step. This might take couple of minutes.

55. Navigate to the CodePipeline console, and refresh the page.

56. Choose the pipeline with **bookmark** in the name.

The deploy stage has been added; however, it is currently grayed out because it hasn't been initiated.

57. Choose Release Change

Note This initiates a new run of the pipeline.

58. In the pop-up window, choose Release

The pipeline runs each stage. After it finishes, all stages will be green. See **Succeeded** in the **Deploy** stage.

Congratulations! You have created a CI/CD pipeline for a serverless application.

Note It takes several minutes for the pipeline to run.

59. In the AWS Management Console, choose Services and select **CloudFormation**.

60. Choose the **Resources** tab to verify the new cloud stack, named **bookmark-app**.

This tab lists all of the resources created that are defined in the AWS SAM template.

Task 5: Updating a Lambda function to test the automated deployment

In this task, you start by using Artillery, which is a load testing and functionality tool. You run the **simple-post.yaml** file from the test folder under **app-code** in AWS Cloud9. This adds bookmarks by invoking the **createBookmark** function.

61. In the AWS Cloud9 workspace, choose the arrow next to the **app-code** folder to expand it, if it is not already expanded.

62. Choose the arrow next to the **test** folder to expand it.

Note The AWS Cloud9 workspace contains two **test** folders. Expand the **test** folder that is a subfolder of **app-code** and not the **test** folder that is a subfolder of the **pipeline** folder.

63. In the **app-code > test** folder, open the **simple-post.yaml** file.

64. In the AWS Cloud9 terminal, run the following script to replace the **API_GATEWAY_URL** in the **simple-post.yaml** file.

```
cd ~/environment/app-code/test
echo export API_GATEWAY_ID=$(aws apigateway get-rest-apis --query
'items[?name==`Bookmark App`].id' --output text) >> ~/environment/app-
code/labVariables
source ~/environment/app-code/labVariables
echo export API_GATEWAY_URL=https://${API_GATEWAY_ID}.execute-
api.${AWS_REGION}.amazonaws.com/dev >> ~/environment/app-code/labVariables
source ~/environment/app-code/labVariables
sed -Ei "s|<API_GATEWAY_URL>|${API_GATEWAY_URL}|g" simple-post.yaml
cd ..
```

Note The script is running the AWS CLI command to get the API Gateway ID and the AWS region to construct the API Gateway URL. This URL is then substituted in the placeholder <API_GATEWAY_URL> in the **simple-post.yaml** file.

65. Save the file.

66. In the AWS Cloud9 terminal, run the following code to install Artillery and Faker and launch the **simple-post.yaml** script:

```
cd ../app-code/test
npm install artillery -g
npm install faker@5.5.3
artillery run simple-post.yaml
```

The **simple-post.yaml** script runs for 30 seconds, adding data through the API and then invoking the **createBookmark** function.

67. In the AWS Management Console, choose Services and open **DynamoDB** in a new tab.

68. In the left navigation pane, choose **Tables**.

69. Choose the **bookmarksTable**.

70. Choose **Explore table items** at the top-right corner of the page.

You see the list of items added to the **bookmarksTable** by the artillery run.

71. In the AWS Cloud9 terminal, run the following commands to retrieve the bookmark details of an item from the **bookmarksTable** and to substitute it in the curl command.

```
source ~/environment/app-code/labVariables
echo export ID=$(aws dynamodb scan --table-name bookmarksTable --query Items[0].id --
output text) >> ~/environment/app-code/labVariables
source ~/environment/app-code/labVariables
curl ${API_GATEWAY_URL}/bookmarks/${ID}
```

Note The *API_GATEWAY_URL* value has been fetched in the above steps and stored in the *labVariables* file. The *labVariables* file contains all of the values fetched so far using the AWS CLI commands.

Note The bookmark details are retrieved for the provided ID.

Now, update the **getBookmark** Lambda function and observe how the function is automatically deployed with the pipeline.

72. In the AWS Cloud9 workspace, under the **backend** folder, expand the **src** folder.

73. In the **src** folder, expand the **getBookmark** folder, and open the **index.js** file.

74. In the **index.js** file, replace the **return** block in the function with the following code snippet:

```
  return {
  statusCode: 200,
  headers: {"Access-Control-Allow-Origin": '*'},
  body: JSON.stringify([('Successfully retrieved bookmark '),results.Item])
};
```

75. Save the file.

76. In the AWS Cloud9 terminal, run the following commands to check in the changes you made in the previous steps:

```
cd ~/environment/app-code
git add .
git commit -m "updated getBookmark function"
git push
```

The pipeline should automatically begin the build process and deploy the AWS SAM template with the changes.

77. Navigate to the **CodePipeline** console to observe the build process.

Make sure that the deployment is completed successfully before moving on to the next step.

78. In the AWS Management Console, choose Services and open **Lambda** in a new tab.

79. In the search field, enter `getBookmark` and choose the function with **getBookmark** in the name.

View the function code to review the updates that have been deployed.

80. In the AWS Cloud9 workspace, run the following curl command again to test the changes.

```
source ~/environment/app-code/labVariables
curl ${API_GATEWAY_URL}/bookmarks/${ID}
```

Note The bookmark details are retrieved for the provided bookmark ID, along with the updated text **Successfully retrieved bookmark**.

The changes that you made were automatically deployed using the CI/CD pipeline.

Task 6: Understanding canary deployments and how to implement them

In this task, you learn about canary deployments and how they play an important role in rolling out changes to production.

A canary deployment is a technique that reduces the risk of deploying a new version of an application by slowly rolling out the changes to a small subset of users before rolling the new version out to the entire customer base. Using blue/green and canary deployments is well established as a best practice for reducing the risk of software deployments. In traditional applications, you slowly and incrementally update the servers in your fleet while simultaneously verifying application health. However, these concepts don't map directly to a serverless world. You can't incrementally deploy your software across a fleet of servers when there are no servers.

However, a couple of services and features make this possible.

Lambda versions and aliases

Lambda allows you to publish multiple versions of the same function. Each version has its own code and associated dependencies, and its own function settings (such as memory allocation, timeout, and environment variables). You can then refer to a given version by using a Lambda alias. An alias is a name that can be pointed to a given version of a Lambda function.

81. In the AWS Cloud9 workspace, under the **backend** folder, open the **template.yaml** file.

82. In the **template.yaml** file, find the line that reads *Role: !Ref LambdaDeploymentRole* (line 117). This line is in the **getBookmark** function under the **Properties** section.

83. Add the following lines after the **Role: !Ref LambdaDeploymentRole** line:

```
AutoPublishAlias: live
DeploymentPreference:
    Type: Canary10Percent5Minutes
```

Note The indentation for the code snippet above should appear as follows when pasted into the **template.yaml** file:

```
getBookmark:
  Type: AWS::Serverless::Function
  Properties:
    FunctionName: !Sub ${AWS::StackName}-getBookmark
    Description: !Sub
      - ${ResourceName} Function
      - ResourceName: getBookmark
    CodeUri: src/getBookmark
    Environment:
      Variables:
        TABLE_NAME: !Ref bookmarksTable
        TABLE_ARN: !GetAtt bookmarksTable.Arn
    Role: !Ref LambdaDeploymentRole
    AutoPublishAlias: live
    DeploymentPreference:
      Type: Canary10Percent5Minutes
    Events:
      apiGET:
        Type: Api
        Properties:
          Path: /bookmarks/{id}
          Method: GET
          RestApiId: !Ref api
  Metadata:
    FinTag: getBookmark
```

84. Save the file.

Deployment preference types

Use the *Canary10Percent5Minutes* strategy for this lab, which means that traffic is shifted in two increments. In the first increment, only 10 percent of the traffic is shifted to the new Lambda version, and after 5 minutes, the remaining 90 percent is shifted. You can choose other deployment strategies in CodeDeploy, such as the following:

- Canary10Percent30Minutes
- Canary10Percent5Minutes
- Canary10Percent10Minutes
- Canary10Percent15Minutes
- Linear10PercentEvery10Minutes
- Linear10PercentEvery1Minute
- Linear10PercentEvery2Minutes

- Linear10PercentEvery3Minutes
- AllAtOnce

The *Linear* strategy means that traffic is shifted in equal increments with an equal time interval between each increment.

85. In the AWS Cloud9 terminal, run the following commands to validate the AWS SAM template:

```
cd ~/environment/app-code/backend
sam validate
```

If the template is correct, a line appears that says the **template.yaml** file is a valid AWS SAM template. If an error appears, then you likely have an indentation issue on the .yaml file.

86. In the AWS Cloud9 terminal, run the following commands from the root directory of the **app-code** project to push the changes:

```
cd ~/environment/app-code
git add .
git commit -m "Canary deployments with SAM"
git push
```

Canary deployments are considerably more successful if the code is monitored during the deployment. You can configure CodeDeploy to automatically roll back the deployment if a specified Amazon CloudWatch metric has breached the alarm threshold. Common metrics to monitor are Lambda invocation errors or invocation duration (latency).

87. In the AWS Cloud9 workspace, in the **template.yaml** file, add the following code snippet at line 131 after the line that reads **FinTag: getBookmark**. This line is at the end of the **getBookmark** function definition.

The following code defines a CloudWatch alarm.

```
CanaryErrorsAlarm:
  Type: AWS::CloudWatch::Alarm
  Properties:
    AlarmDescription: Lambda function canary errors
    ComparisonOperator: GreaterThanThreshold
    EvaluationPeriods: 2
    MetricName: Errors
    Namespace: AWS/Lambda
    Period: 60
    Statistic: Sum
    Threshold: 0
    Dimensions:
      - Name: Resource
        Value: "getBookmark:live"
      - Name: FunctionName
        Value: !Ref getBookmark
      - Name: ExecutedVersion
        Value: !GetAtt getBookmark.Version.Version
```


Note It is important to maintain the indentation when inserting the new code. The indentation should look like the following under the **getBookmark** function definition:

```
      Properties:
        Path: /bookmarks/{id}
        Method: GET
        RestApiId: !Ref api
    Metadata:
      FinTag: getBookmark

  CanaryErrorsAlarm:
    Type: AWS::CloudWatch::Alarm
    Properties:
      AlarmDescription: Lambda function canary errors
      ComparisonOperator: GreaterThanThreshold
      EvaluationPeriods: 2
      MetricName: Errors
      Namespace: AWS/Lambda
      Period: 60
      Statistic: Sum
      Threshold: 0
      Dimensions:
        - Name: Resource
          Value: !Sub "getBookmark:live"
        - Name: FunctionName
          Value: !Ref getBookmark
        - Name: ExecutedVersion
          Value: !GetAtt getBookmark.Version.Version

  updateBookmark:
    Type: AWS::Serverless::Function
```

88. In the **template.yaml** file, copy and paste the following lines at line 121 under the **DeploymentPreference** section of the **getBookmark** function definition:

```
Alarms:
  - !Ref CanaryErrorsAlarm
```

Note It is important to maintain the indentation when inserting the new lines. The indentation should align with the **DeploymentPreference** section as follows:


```
getBookmark:
  Type: AWS::Serverless::Function
  Properties:
    FunctionName: !Sub ${AWS::StackName}-getBookmark
    Description: !Sub
      - ${ResourceName} Function
      - ResourceName: getBookmark
    CodeUri: src/getBookmark
    Environment:
      Variables:
        TABLE_NAME: !Ref bookmarksTable
        TABLE_ARN: !GetAtt bookmarksTable.Arn
    Role: !Ref LambdaDeploymentRole
    AutoPublishAlias: live
    DeploymentPreference:
      Type: Canary10Percent5Minutes
      Alarms:
        - !Ref CanaryErrorsAlarm
    Events:
      apiGET:
        Type: Api
        Properties:
          Path: /bookmarks/{id}
          Method: GET
          RestApiId: !Ref api
  Metadata:
    FinTag: getBookmark
```

89. In the **template.yaml** file, in the **api** section, find the following **uri** at line 46 for the **/bookmarks/{id}** get method:

```
uri: !Sub arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${getBookmark.Arn}/invocations
```

90. Change that **uri** to the following:

```
uri: !Sub arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${getBookmark.Arn}:live/invocations
```

Note This update ensures that API Gateway can correctly return the version of the **getBookmark** Lambda function that is being invoked.

91. Save the file.

92. In the AWS Cloud9 terminal, run the following commands to validate the AWS SAM template:

```
cd ~/environment/app-code/backend
sam validate
```

If the template is correct, a line appears that says the **template.yaml** file is a valid AWS SAM template. If an error appears, then you likely have an indentation issue on the .yaml file.

93. In the AWS Cloud9 workspace, in the **getBookmark** folder, open the **index.js** file.

94. In the **index.js** file, replace the **return** block in the function with the following code to create a new version:

```
  return {
  statusCode: 200,
  headers: {"Access-Control-Allow-Origin": '*'},
  body: JSON.stringify(['Successfully retrieved bookmark using the new version' ,
results.Item])
};
```

95. Save the file.

96. In the AWS Cloud9 terminal, run the following commands to push the changes:

```
cd ~/environment/app-code
git add .
git commit -m "Added CloudWatch alarm to monitor the canary"
git push
```

97. Navigate to the browser tab with the CodePipeline console, and wait for the pipeline to get to the deployment stage (Deploy).

When the Deploy stage is **In Progress**, in the left navigation pane, choose **CodeDeploy** and open it in a new tab. You can watch the deployment progress here.

98. In the left navigation pane, select the arrow next to **Deploy**, and choose **Deployments**.

99. Choose the **Deployment id** to review the details.

The deployment status shows that 10 percent of the traffic has been shifted to the new version of the Lambda function (the canary). CodeDeploy holds the remaining percent until the specified time interval has elapsed. In this case, the specified interval is 5 minutes.

100. In the AWS Cloud9 terminal, run the following script to test the Lambda version invocation:

```
source ~/environment/app-code/labVariables
counter=1
while [ $counter -le 120 ]
do
    curl ${API_GATEWAY_URL}/bookmarks/${ID}
    sleep 1
    ((counter++))
    printf "\n"
done
```

Note The difference in the return statement value during the deployment process indicates that the different versions of the Lambda function are being invoked.

Wait 5 minutes until the remaining traffic is shifted to the new version. You can verify this shift by checking the **Deployment** details in the CodeDeploy console.

101. After the traffic has shifted to the new version, go to the AWS Cloud9 workspace, and run the script again to see only the new version of the Lambda function being invoked.

Rollbacks

Monitoring the health of your canary allows CodeDeploy to make a decision about whether a rollback is needed or not. If any of the specified CloudWatch alarms gets to ALARM status, CodeDeploy rolls back the deployment automatically.

Next, you break the Lambda function on purpose so that the **CanaryErrorsAlarm** alarm is invoked during deployment.

102. In the AWS Cloud9 workspace, in the **getBookmark** folder, open the **index.js** file.

103. Replace the entire function code with the following code to create an error on every invocation:

```
const AWS = require('aws-sdk');
var dynamodb = new AWS.DynamoDB()

exports.handler = async message => {
  throw new Error("this will cause a deployment rollback");
}
```

104. Save the file.

105. In the AWS Cloud9 terminal, run the following commands to push the changes:

```
git add .
git commit -m "Breaking the lambda function on purpose"
git push
```

In the CodePipeline console, wait for the pipeline to reach the deployment phase (Deploy). It should turn blue when it begins.

While the deployment is running, you need to generate traffic to the new Lambda function to make it fail and invoke the CloudWatch alarm. In a real production environment, your users would likely generate organic traffic to the canary function, so you might not need to do this.

106. In the AWS Cloud9 terminal, run the following script to test the canary deployment:

```
source ~/environment/app-code/labVariables
counter=1
while [ $counter -le 120 ]
do
    curl ${API_GATEWAY_URL}/bookmarks/${ID}
    sleep 1
    ((counter++))
    printf "\n"
done
```


Note During deployment, only 10 percent of the traffic is routed to the new version, so the Lambda function is invoked many times by looping it. One out of 10 invocations should invoke the new broken Lambda function, which is what you want to do to cause a rollback.

107. Navigate to the CodeDeploy console, and choose the **Deployment** that is **In progress** to view its details.

After a few minutes, CodeDeploy detects that the **CanaryErrorsAlarm** alarm has been invoked, and CodeDeploy starts rolling back the deployment.

Conclusion

Congratulations! You have successfully:

- Created a CodeCommit repository and a CI/CD pipeline
- Used AWS SAM to define the resources that your application needs and the AWS CDK to define the resources for the deployment infrastructure
- Implemented canary deployments using AWS SAM
- Monitored your canary deployment with CodeDeploy

End lab

Follow these steps to close the console, end your lab, and evaluate your lab experience.

108. Return to the **AWS Management Console**.

109. At the upper-right corner of the page, choose **awsstudent@<AccountNumber>**, and then choose **Sign out**.

110. Choose End Lab.

111. Choose Submit.

112. (Optional):

- Select the applicable number of stars to rate your lab experience.
  - 1 star = Very dissatisfied
  - 2 stars = Dissatisfied
  - 3 stars = Neutral
  - 4 stars = Satisfied
  - 5 stars = Very satisfied

- Enter a comment.
- Choose **Submit**.

You can close the window if you don't want to provide feedback.
For more information about AWS Training and Certification, see *https://aws.amazon.com/training/*.

Your feedback is welcome and appreciated.

If you would like to share any feedback, suggestions, or corrections, please provide the details in our *AWS Training and Certification Contact Form*.

Additional resources

- For more information about CI/CD, see https://aws.amazon.com/getting-started/projects/set-up-ci-cd-pipeline/.
- For more information about the AWS CDK, see https://docs.aws.amazon.com/cdk/latest/guide/getting_started.html.
- For more information about a canary-based deployment using Lambda, see https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/perform-a-canary-based-deployment-using-the-blue-green-strategy-and-aws-lambda.html

aws training and certification

Try-It-Out Exercises: Day 1

Note: Do not include any personal, identifying, or confidential information into the lab environment. Information entered may be visible to others.

Corrections, feedback, or other questions? Contact us at *AWS Training and Certification*.

Overview

During the course, you use this environment to interact with the AWS Management Console and some of the services that the course discusses. The environment will be available to you throughout the day but will be reset for the following day's class.

Prerequisites

This lab requires:

- Access to a notebook computer with Wi-Fi and Microsoft Windows, macOS, or Linux (Ubuntu, SUSE, or Red Hat)
- For Microsoft Windows users, administrator access to the computer
- An Internet browser such as Chrome, Firefox, or Internet Explorer 9 (previous versions of Internet Explorer are not supported)

⚠ **Note** The lab environment is not accessible using an iPad or tablet device, but you can use these devices to access the lab guide.

Start lab

1. To launch the lab, at the top of the page, choose Start Lab.

This starts the process of provisioning the lab resources. An estimated amount of time to provision the lab resources is displayed. You must wait for the resources to be provisioned before continuing.

ⓘ If you are prompted for a token, use the one distributed to you (or credits you have purchased).

2. To open the lab, choose Open Console.

The **AWS Management Console** sign-in page opens in a new web browser tab.

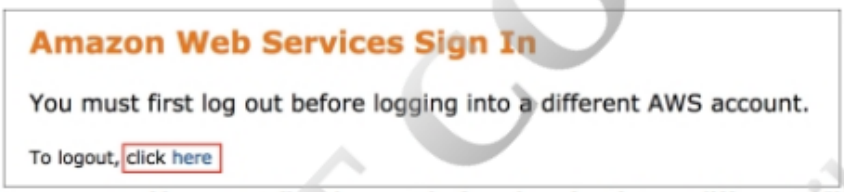
3. On the **Sign in as IAM user** page:

- For **IAM user name**, enter `awsstudent`.
- For **Password**, copy and paste the **Password** value listed to the left of these instructions.
- Choose Sign in.

⚠ **Do not change the Region unless instructed.**

Common sign-in errors

Error: You must first sign out



If you see the message, **You must first log out before logging into a different AWS account:**

- Choose the **click here** link.
- Close your **Amazon Web Services Sign In** web browser tab and return to your initial lab page.
- Choose Open Console again.

In some cases, certain pop-up or script blocker web browser extensions might prevent the **Start Lab** button from working as intended. If you experience an issue starting the lab:

- Add the lab domain name to your pop-up or script blocker's allow list or turn it off.
- Refresh the page and try again.

Module 2

Use the Amazon API Gateway console to build an HTTP API integrated with an AWS Lambda function

In this task, you build an HTTP API with a GET route that is integrated with an Lambda function called **bakeryFunction**. When you call the API endpoint in your browser, the function will randomly suggest a snack pick that you might buy from the bakery.

Create the API

4. Navigate to the **API Gateway** console, choose Create API, and choose Build to build an HTTP API.
5. Enter the API name of your choice (the examples provided use the API name `Bakery`). Choose Review and Create and then Create

Create a route

6. From the **Navigation** menu, choose **Routes**, and then choose Create
7. Set the method to **GET**, name the path whatever you like (for example, `snackpick`), and choose Create

Add the Lambda integration

8. Choose the **GET** route you just created, and then choose Attach integration and then Create and attach an integration
9. Set the **Integration type** to **Lambda function**.
10. In the **Integration details** section, under **Lambda function**, choose **bakeryFunction**.

    Note Notice the **Invoke permissions** section with the option to **Grant API Gateway permission to invoke your Lambda function**. Leave this turned on. This automatically creates the permissions your API needs to invoke the **bakeryFunction** function.

11. Choose Create so that your method is automatically deployed to the **$default** stage. In the left navigation pane, choose **Stages** under **Deploy**. In the **Stages** pane, select **$default** and find the **Invoke URL**.

Test the endpoint

12. Copy the URL into your browser, and add your path name to the end of it (for example, https://4o8lnbzky4.execute-api.us-west-2.amazonaws.com/snackpick). When you invoke it, you should get a response that says something like the following:

```
Here is my next snack pick: Doughnut
(Refresh the page for the next pick)
```

Note You can manage multiple versions of your API using stages. When your HTTP API is created, the **$default** stage is created and set to autodeploy.

Usually, the **$default** stage is your production API, so you may turn off autodeploy on the **$default** stage and create another stage (for example, **dev**) for testing your updates before you deploy the production stage. For the lab, leave **$default** set to autodeploy.

13. On the **Stages** panel, choose Create to add a new stage. Give your new stage a name, turn on the **Enable automatic deployment** option, and choose Create

Note the difference in the **Invoke URL** for the stage you created versus the **$default** stage.

Review the Lambda permissions associated with your API

14. Navigate to the **Lambda** console, and choose **bakeryFunction**. In the **Configuration** tab, choose **Triggers**, and you should find the API Gateway trigger listed.

15. If you go to the **Configuration** tab, choose **Permissions** in the left navigation pane, and scroll to the **Resource-based policy** section, you should find the policy that was created automatically when you created the Lambda integration in your API. To view the policy, choose View policy document

It includes a policy statement similar to the following example.

```
...
"Effect": "Allow",
"Principal": {
  "Service": "apigateway.amazonaws.com"
},
"Action": "lambda:InvokeFunction",
"Resource": "arn:aws:lambda:us-west-2:123456789012:function:bakeryFunction",
"Condition": {
  "ArnLike": {
   "AWS:SourceArn": "arn:aws:execute-api:us-west-
2:123456789012:4o8lnbzky4/*/*/snackpick"
  }
}
...
```

Optional task: Build and deploy a REST API from an example, and then generate the SDK for the API

You can do this task later on your own AWS account.

16. From the **API Gateway** console, choose Create API

17. On the REST API panel, choose the Build button. Then choose the **Example API** radio button, and choose Import

The PetStore API is created in your account, and you can look at the details in the **Resources** section.

18. REST APIs do not autodeploy, and there is not a **$default** stage. To deploy the example API, choose Actions and then **Deploy API**. Choose **[New Stage]** for the **Deployment stage**, give your stage a name such as `dev` and choose Deploy

19. To generate an SDK for this API, choose the stage you just deployed (for example, **dev**), select the **SDK Generation** tab, and choose a platform. Choose Generate SDK and save the generated file.

Module 5

Use the Amazon EventBridge console to build a custom event bus and add a custom event rule

In this task, you set up an event bus that could be used to route events to different Lambda functions based on the type of snack indicated in the event. You will set up a rule that routes events about the snack type **Pie** to a Lambda function called **pieFunction** and add a POST route to the HTTP API you set up earlier that is attached to your custom EventBridge bus (serverless-bus). Finally, you'll test it via the command line in AWS Cloud9 and verify the results using Amazon CloudWatch Logs.

Create the event bus

20. Navigate to the EventBridge console, and choose **Event buses**.

21. In the **Custom event bus** panel, choose Create event bus

22. Name the event bus `serverless-bus` and choose Create

Create the rule

The rule should route only events that include the snack type **Pie** in the detail of the event to the **pieFunction** function. Events without the snack type **Pie** will not be routed to the function.

23. From the EventBridge console menu, choose **Rules**.

24. Choose Create rule

25. Configure the following information to set up the rule:

On the **Define rule detail** page, in the **Rule detail** section, configure the following information:

- **Name:** Enter `pie-rule`.
- **Event bus:** Select **serverless-bus** from the dropdown menu.
- **Rule Type:** Select **Rule with an event pattern**.

26. Choose Next

27. In the **Build event pattern** page, configure the following information:

- **Event source:** Select **Other**.
- **Event pattern:** Select **Custom patterns**.
- In the **Event pattern** code box, copy and paste in the following code:

```
{
  "source": [
    "Bakery Store"
  ],
  "detail-type": [
    "snacks"
  ],
  "detail": {
    "snack-type": [
      "Pie"
    ]
  }
}
```

28. Choose Next
29. In the **Select target(s)** page, configure the following information:

- **Target types:** Select **AWS Service**.
- **Select a target:** Select **Lambda function** from the dropdown menu.
- **Function:** Select **pieFunction** from the dropdown menu.

30. Choose Next
31. In the **Configure tags - optional** page, choose Next
32. In the **Review and create** page, choose Create rule

Create the API route

ⓘ If you ended this lab at any point during the day and lost the API you built in Module 2, you can use the prebuilt **Bakery-Sample-Api** for the following steps.

33. Open your **Bakery** API Gateway HTTP endpoint, and add a new route with method type **POST** and a path name of **newsnack**.
34. With POST selected, choose Attach integration and then Create and attach an integration
35. In the **Integration type** dropdown list, select **Amazon EventBridge**.
36. Select **PutEvents** for the Integration action, and then configure the following information:

- **Detail:** Enter `$request.body.Detail`
- **Detail Type:** Enter `snacks`
- **Source:** Enter `Bakery Store`

- **Invocation role:** From the left navigation of the lab page, copy the **APIGatewayInvocationRole** and paste the value into the field.

Under **Advanced settings**, configure the following information:

- **Event bus name:** Enter `serverless-bus`

37. Choose Create

This update is automatically deployed to the **$default** stage.

Test the rule

38. Copy the **Invoke URL** for the **$default** stage of your API.

39. Open your **AWS Cloud9** environment.

40. In the following command, replace *API-GATEWAY-ENDPOINT-URL* with the **Invoke URL**. Run the updated command in the AWS Cloud9 terminal.

```
curl --location --request POST "API-GATEWAY-ENDPOINT-URL/newsnack" \
--header 'Content-Type: application/json' \
--data-raw '{"Detail": {"snack-type": "Pie","snack-name": "Apple Pie"}}'
```

The above call to the API directly invokes EventBridge, which in turn invokes the **pieFunction** function. Your command line response should look similar to the following:

```
{"Entries":[{"EventId":"29f6c069-4891-5188-e5a6-69a514eb30db"}],"FailedEntryCount":0}
```

41. In the **Lambda** console, open the **pieFunction** function.

42. Copy the **Function ARN** to a text editor to use later.

43. Choose the **Monitor** tab, and then select **View logs in CloudWatch**.

44. Choose the most recent log stream, and look for the event details that you included in your POST request.

45. Test the rule with another API call, but this time change the snack type to something other than **Pie**. This change should not result in an invocation of the Lambda function. Verify within CloudWatch Logs.

Use the Amazon SNS console to subscribe a Lambda function to a topic using attribute filtering

In this task, you set up an Amazon Simple Notification Service (Amazon SNS) topic that could be used to provide subscribers with snack event information. You'll subscribe the **pieFunction** function to the topic and configure attribute filtering on the subscription so that only events that include the snack type of **Pie** are delivered to the **pieFunction** subscriber. You'll also use the Amazon SNS console to create test messages and test the subscription filter policy.

Create an Amazon SNS topic with a Lambda function subscriber and a filter policy

46. Navigate to the **Amazon SNS** console. From the console navigation menu, choose **Topics**, and then choose Create topic
47. For the topic type, select **Standard**, give your topic a name, and choose Create topic
48. From the topic details page, choose Create subscription
49. Under **Protocol**, select **AWS Lambda**, and enter the **Function ARN** of the **pieFunction** that you copied earlier as the Endpoint.
50. Under **Subscription filter policy**, add the following filter:

```
{
  "type": [
    "Pie",
    "pie",
    "Pies",
    "pies"
  ]
}
```

51. Choose Create subscription

Test the filter policy

First, publish a message that doesn't have attributes that match the filter.

52. On the topic you created, choose Publish message and then configure the following information:

- **Subject**: Enter `New Bakery Items`
- **Message body**: Enter `A new donut flavor has been added in the bakery store`

Under **Message attributes**, configure the following information:

- **Type:** Choose **String** from the dropdown list
- **Name:** Enter `type`
- **Value:** Enter `Donut`

53. Choose Publish message

Now, publish a message with attributes that match the filter.

54. On the topic you created, choose Publish message and then configure the following information:

- **Subject**: Enter `New Bakery Items`

- **Message body**: Enter `A new pie flavor has been added in the bakery store`

Under **Message attributes**, configure the following information:

- **Type:** Choose **String** from the dropdown list
- **Name:** Enter `type`
- **Value:** Enter `Pie`

55. Choose Publish message
56. Go to CloudWatch Logs for the **pieFunction** function, and open the most recent log stream. You should find the invocation for the message that included **Pie** in the attribute but not for the message with the **Donut** attribute.

Module 6

Use the Amazon SQS console to set up a queue as a Lambda event source

In this task, you use the Amazon Simple Queue Service (Amazon SQS) console to set up a queue as a Lambda event source and use the console features to perform basic testing with a dead-letter queue.

Set up the queue and its dead-letter queue

57. Navigate to the **Amazon SQS** console, and choose Create queue
58. In the **Details** section, name the queue `donut-queue` in the **Name** column.
59. Scroll to the **Dead-letter queue** - Optional section, expand the section and switch the radio button to **Enabled**.
60. In the **Choose queue** dropdown list, select **donut-dlq**.
61. Set **Maximum receives** to **2**.
62. Choose Create queue

Test the dead-letter queue

Note If you were to configure the Lambda function trigger before you run this test, Lambda would pick up and process the messages before you could interact with them on the console.

63. With your **donut-queue** details on the console, select Send and receive messages
64. Enter any simple message, and choose Send message

65. Modify each message body to distinguish them (for example, **test message 2** and **test message 3**), and add a couple more messages.

66. Scroll to the **Receive messages** section, and choose Poll for messages

Your messages will appear in the list. Note the receive count.

67. Choose Poll for messages again. The receive count increments.

68. Choose Poll for messages a third time.

The list of messages in the queue should exclude any messages with a maximum receive count higher than two. You can now find those messages in the dead-letter queue.

Configure and test the Lambda trigger on the donut-queue

69. Open the **donut-queue** queue, select the **Lambda triggers** tab, and choose Configure Lambda function trigger

70. From the dropdown list, choose **donutFunction**.

71. Choose Save

72. On the **donut-queue**, choose the **Send and receive messages** option again, and send another message.

73. Find the CloudWatch logs for the **donutFunction**, and verify that the function was invoked with the message from the Amazon SQS queue.

End lab

Follow these steps to close the console, end your lab, and evaluate your lab experience.

74. Return to the **AWS Management Console**.

75. At the upper-right corner of the page, choose **awsstudent@<AccountNumber>**, and then choose **Sign out**.

76. Choose End Lab.

77. Choose Submit.

78. (Optional):

- Select the applicable number of stars to rate your lab experience.
  - 1 star = Very dissatisfied
  - 2 stars = Dissatisfied
  - 3 stars = Neutral

  - 4 stars = Satisfied
  - 5 stars = Very satisfied
- Enter a comment.
- Choose **Submit**.

You can close the window if you don't want to provide feedback.

For more information about AWS Training and Certification, see *https://aws.amazon.com/training/*.

Your feedback is welcome and appreciated.

If you would like to share any feedback, suggestions, or corrections, please provide the details in our *AWS Training and Certification Contact Form*.

aws training and certification

Try-it-out Exercises: Day 2

Note: Do not include any personal, identifying, or confidential information into the lab environment. Information entered may be visible to others.

Corrections, feedback, or other questions? Contact us at *AWS Training and Certification*.

Overview

During the course, you use this environment to interact with the AWS Management Console and some of the services that the course discusses. The environment will be available to you throughout the day but will be reset for the following day's class.

Prerequisites

This lab requires:

- Access to a notebook computer with Wi-Fi and Microsoft Windows, macOS X, or Linux (Ubuntu, SUSE, or Red Hat)
- For Microsoft Windows users, administrator access to the computer
- An internet browser such as Chrome, Firefox, or Internet Explorer 9 (previous versions of Internet Explorer are not supported)
- A text editor

⚠ **Note** The lab environment is not accessible using an iPad or tablet device, but you can use these devices to access the lab guide.

Start lab

1. To launch the lab, at the top of the page, choose Start Lab.

This starts the process of provisioning the lab resources. An estimated amount of time to provision the lab resources is displayed. You must wait for the resources to be provisioned before continuing.

ⓘ If you are prompted for a token, use the one distributed to you (or credits you have purchased).

2. To open the lab, choose Open Console.

The **AWS Management Console** sign-in page opens in a new web browser tab.

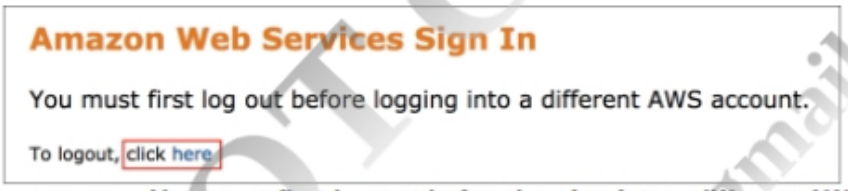
3. On the **Sign in as IAM user** page:

- For **IAM user name**, enter `awsstudent`.
- For **Password**, copy and paste the **Password** value listed to the left of these instructions.
- Choose Sign in.

⚠ **Do not change the Region unless instructed.**

Common sign-in errors

Error: You must first sign out



If you see the message, **You must first log out before logging into a different AWS account:**

- Choose the **click here** link.
- Close your **Amazon Web Services Sign In** web browser tab and return to your initial lab page.
- Choose Open Console again.

In some cases, certain pop-up or script blocker web browser extensions might prevent the **Start Lab** button from working as intended. If you experience an issue starting the lab:

- Add the lab domain name to your pop-up or script blocker's allow list or turn it off.
- Refresh the page and try again.

Module 7

Build two serverless applications from examples

To get started in this module, you deploy applications that help to illustrate topics discussed later in this module and the next module. These examples are also available for use in your own account if you want to review or work with them later.

For both examples, you download and unzip each application using your AWS Cloud9 environment. The random error generator application has been scoped for use in this course, but you can get a copy of the application on which it is based from https://docs.aws.amazon.com/lambda/latest/dg/samples-errorprocessor.html. The serverless API application is available as a sample application in the AWS Lambda applications console.

4. In the **AWS Cloud9** environment, run the following commands to download the code, unzip it, and deploy both applications:

```
wget https://us-west-2-tcprod.s3-us-west-2.amazonaws.com/courses/ILT-TF-200-SVDVSS/v1.0.29/try-it-out-Day2/scripts/day-2-apps.zip
unzip day-2-apps.zip
cd day-2-apps
chmod +x *.sh
./resize.sh 20
./create-bucket.sh
./build-layer.sh
./deploy.sh
```

5. Run the **invoke** script:

```
./invoke.sh
```

Note The invoke script continues to run until you stop it. If you run the command by right-clicking the *invoke.sh* file in the file navigation window in AWS Cloud9, it opens in its own terminal window with an option to stop it. You come back and stop it after it has generated data for the course examples.

Try-it-out exercise: Lambda console

In your Cloud9 browser window, choose **AWS Cloud9** in the menu tab on the top and then select **Go To Your Dashboard** from the list. From the **Your environments** tab of your browser, navigate to the **Lambda** console. Use the details below to follow your instructor.

Review the resource-based policy

6. Open the **putItemFunction** function. Choose the **Configuration** tab, then choose **Permissions** in the left navigation pane, and scroll to the section titled **Resource-based policy** to review who or what services have permission to invoke your function.

Review the execution role

7. Choose the **Configuration** tab, then choose **Permissions** in the left navigation pane and find the role name in the **Execution role** section. Use the dropdown option in the **Resource summary** section to review the services and permissions that the function has through its execution role.

Review requirements for connecting to a VPC

8. Within the **Configuration** tab of your function, choose **VPC** in the left navigation pane to find the characteristics of the VPC that you would need to configure.

Add an OnFailure destination

9. Navigate to the **Function overview** section of your function.

10. Choose **Add destination**. Select the desired target (for example, an Amazon Simple Notification Service (SNS) topic).

Modify the memory setting

11. Choose the **Configuration** tab, and select **General configuration** in the left navigation pane to modify the memory for your function.

Modify timeout setting

12. In the same section, you can also modify the timeout.

Review Lambda function code

13. Choose the **Code** tab, and navigate to the **Code source** section. This section lets you directly review and edit functions depending on how they were written and deployed.

Review code to separate the business logic

14. Navigate to the **error-processor-randomerror** function. In the **Code source** section, select **index.js**, open the context (right-click) menu and choose **Open**.

The handler consists of a single line of code:

```
exports.handler = myFunction
```

All of the other logic happens in the **myFunction** function starting with the following line:

```
var myFunction = async function(event, context) {
```

Review where layers are added

15. The **error-processor-randomerror** function uses a layer. In the **Code** tab of the function, scroll to the **Layers** section. You can add a layer in two ways: Within the **Code** tab, use the **Add a layer** option, or go to the Lambda navigation menu on the left, and use the **Layers** option.

Enable CloudWatch Lambda Insights

16. While you have the **error-processor-randomerror** function open, choose the **Configuration** tab, select the **Monitoring and operation tools** section, and choose **Edit**. Under **CloudWatch Lambda Insights**, turn on **Enhanced monitoring** and **Save**.

Review environment variable example

17. The **error-processor-processor** function has an example of an environment variable set for the bucket used in the function. Open the function, choose the **Configuration** tab, select the **Environment variables** section, and you'll see the bucket variable. You can see the variable being used in the function in the following line:

```
const bucket = process.env.bucket
```

Publish a new version of a function

18. Choose the **Actions** dropdown, choose **Publish new version**, and then choose **Publish**.

19. Navigate back to the **error-processor-processor** function overview page. Choose the **Versions** tab. You can see **1** in the **Versions** section as an option.

Create an alias

20. Choose the **Actions** dropdown, and then choose **Create alias**. Name the alias, select the version to associate it with, and choose **Save**. Choose the **Aliases** tab, and you'll be able to see the alias as an option.

Configure the test event

21. Choose the **error-processor-randomerror** function. Choose the **Test** tab to create a test event.

22. Name the test event, and paste in the following code snippet:

```
{
  "max-depth": 50,
  "current-depth": 0,
  "error-rate": 0.05
}
```

Test the event

23. Choose **Test**. A panel above the main console sections displays your results.

Module 8

Try-it-out exercise: AWS Step Functions

24. Navigate to the **Step Functions** console. Use the details below to follow your instructor.

Start creation of a state machine

25. In the Step Functions console navigation menu, select **State machines**. Choose **Create state machine**. Select **Design your workflow visually**. Select **Standard** for the type. Choose **Next**.

Preview code for a task state

26. In the **Design workflow** page, choose **AWS Lambda: Invoke** from the list of **Actions**, and drop it in the workflow state where it says *Drag first state here*. Choose **Definition** on the right side of the pane to preview the code that you would use to incorporate this as a step in your state machine.

Preview the code for a Choice state

27. Choose **Delete** in the design pane to clear the existing selection.

28. In the **Flow** section, select the **Choice state** template and drop it in the workflow state where it says *Drag first state here*. Review the visualization in the design pane and the code populated in the **Definition** section.

Preview the code for a Parallel state

29. Choose **Delete** in the design pane to clear the existing selection.

30. Select the **Parallel** template in the **Flow** section and drop it in the workflow state where it says *Drag first state here*. Review the visualization in the design pane and the code populated in the **Definition** section.

Preview the code for a Wait state

31. Choose **Delete** in the design pane to clear the existing selection.

32. In the **Flow** section, select the **Wait state** template and drop it in the workflow state where it says *Drag first state here*. Review the visualization in the design pane and the code populated in the **Definition** section.

Preview the code to wait for a batch job to complete

33. Choose **Delete** in the design pane to clear the existing selection.

34. From the list of **Actions**, find and choose **AWS Batch**.

35. From the list of **AWS Batch** actions, select **AWS Batch: SubmitJob** and drop it in the workflow state where it says *Drag first state here*. Review the code in the **Definition** section in the right navigation pane. Choose **Form** and toggle the check box for **Wait for task to complete - optional**. Review the resulting code changes in the **Definition** section. The *sync* API call is added to the *Resource* invocation.

Preview the code to use a task token with callback

36. Choose **Delete** in the design pane to clear the existing selection.

37. Select **Amazon SNS: Publish** from the list of **Actions**. Review the code in the **Definition** section in the right navigation pane. In the **Form** section, select the check box for **Wait for callback - optional**.

38. Choose **Done** to close the **Wait for callback** window. Review the code changes in the **Definition** section. The *waitforTaskToken* API call is added to the *Resource* invocation.

Preview the code for a Map state

39. Choose **Delete** in the design pane to clear the existing selection.

40. In the **Flow** section, select the **Map state** and drop it in the workflow state where it says *Drag first state here.*. Review the visualization in the design pane and the code populated in the **Definition** section.

Select Retry on errors

41. In the **Form** section, choose the **Error handling** tab. Select the **Add new retrier** button. You'll find the array of retriers in the **Errors** drop down list. You can add one or more retriers and view the code in the **Definition** section.

Select Catch errors

42. Under **Catch errors**, select the **Add new catcher** button. You'll find the catch field and its array of catchers and, beneath that, the states that are used for fallback from the catchers. You can view the code in the **Definition** section.

43. Choose **Cancel** to return to the **State machines** page.

Troubleshoot a Step Functions express workflow

44. In the Step Functions console, select the **CheckName-Express** express state machine.

45. Choose Start execution

46. In the **Input** field, enter the payload below:

```
{ "application": { "name": "evil Spock" } }
```

47. Choose Start execution

This produces an error. With express workflows, you can review CloudWatch logs to troubleshoot the error. (In the **Results** panel, use the CloudWatch Logs link that tells you that the state machine has failed.)

To take advantage of the visual results available in standard workflows, copy the express state machine definition to a new standard state machine.

48. On the **CheckName-Express** state machine, from the **Actions** menu, choose **Copy to new**.

49. Change the type to **Standard**, and choose Next
50. Choose Next and then choose Next again.
51. Name the new state machine `CheckName-Standard`
52. In the **Permissions** section, choose **Choose an existing role**. Do not modify the role that is presented. Choose Create state machine
53. Choose Start execution again, and use the same payload in the input field:

```
{ "application": { "name": "evil Spock" } }
```

54. When the standard state machine fails, use the **Graph inspector** to isolate where the failure occurred.
55. Choose the **Denied** task, and look at the details associated with the failure.

Module 9

Configure Amazon API Gateway logging and AWS X-Ray tracing and generate sample traffic

Enable CloudWatch Logs and X-Ray for the REST API created with your serverless API backend application

ⓘ If you ended this lab at any point during the day, repeat the first two steps in the Module 7 section to deploy and invoke the serverless apps again before proceeding with the following steps.

56. Navigate to the **API Gateway** console and select the **serverless-api** REST API.
57. In the left navigation pane of the lab page, copy the **APIGatewayInvocationRole** value.
58. At the bottom of the left navigation pane of the API Gateway console, choose **Settings**.
59. In the **CloudWatch log role ARN** box, paste the **APIGatewayInvocationRole** value you copied from the lab page.
60. Choose Save
61. Choose **Stages**, and then choose the **Prod** stage to open the Prod stage editor. Choose the **Logs/Tracing** tab.
62. Select the **Enable CloudWatch Logs** check box. For the **log level**, choose **INFO**. (For production APIs, you would typically use the **ERROR** level rather than logging everything, but for this example, use the **INFO** option to produce more detailed logs.)
63. Select the **Enable X-Ray Tracing** check box.

