# Part 4: Code Review, Deploy, & Configure an effective CI/CD Pipeline on AWS

*Author:* [*David P. Lopez*](http://www.DavidPLopez.com)

*Publisher:* [*UniqueSoftwareDevelopment*](https://www.uniquesoftwaredev.com)

This is a continuation of our multi-part series on building a simple web application on AWS using AWS Lambda and the ServerlessFramework. You can review the first, second, and third parts of this series starting with the setup of your local environment at:

* [Part 1: How To SetUp Your local Serverless Environment](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToSetUpYourLocalServerlessEnvironment.md)
* [Part 2: How To Configure Your Serverless Backend API](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToConfigureYourServerlessBackend.md)
* [Part 3: How To Configure Your Infrastructure As Code, Mock Services, & Unit Testing](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToConfigure.IAC.Mocks.UnitTests.md)

You can also clone a sample of the application we will be using in this tutorial here: [Serverless-Starter-Service](https://github.com/lopezdp/ServerlessStarterService)

Please refer to this repo as you follow along with this tutorial.

## Code Review Agility

We have definitely covered a lot of ground by now, and it may be best to take a minute and work through a *Code Review* of sorts, that we can use to absorb all of this information that we have reviewed and implemented together before merging our code and deploying our services into *production*. It is definitely a good idea to take this time to think about the workflow that we will put in place to manage our day to day development, and implementation activities as a team of engineers. [In my case it's just Wilson and I, remember](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToSetUpYourLocalServerlessEnvironment.md#setup-editor-and-install-linting)?

**Stay Agile OB-1**



Collaborating with other team members can be tricky, that is, it can be tricky if you don't have a plan in place to work in an organized manner!!! We recommend that the documentation for any project repository include a set of ContributionGuidelines.md that dictate how each member of your team shall interact with the organization's source code. There are numerous examples out in the [ether](https://www.google.com) that you can look to as a set of guiding principles, and yet, you know I have a suggestion for you anyway. As someone who has published OpenSource work for NASA as a contractor, I suggest you model any ContributionGuidelines that you make for you and your team after the [NASA OpenMCT](https://github.com/nasa/openmct/blob/master/CONTRIBUTING.md) project guildelines to start off. It is pretty straght forward and really gets to the point in terms of what a good OpenSource contribution policy should look like.

### Pull Request Check Lists Examples & Best Practices

In an Agile development environment, we just want a process that we can use to iterate over a predefined *Code Review* workflow that will help us implement and merge new updates to our source code efficiently, transparently, and with close to zero downtime in the *Wild*. When a team member writes and implements a set of features, there should be someone, again, in my case Wilson, who will review the code you have implemented on a topic-branch in git after you create a Pull Request for your project lead to review your code.

The *Code Review* process is an important part of your team workflow because it allows you to share the knowledge you gained from the implementation of the logic and functionality that defines the feature you will deploy. It also gives you a layer of quality assurance that enables your peers to contribute and provide insight into the feature you will deploy, and it allows new team members to learn from others on the team by taking ownership of a feature and implementing the logic the new feature needs so that it can be accepted by the project owner.

The Agile framework requires that every *Code Review* complete a *Pull Request* in phases. The first phase would require your *Tech Lead* to look at your implementation and any changes you made, and compare it to the code that you are refactoring.

The second phase of the review requires that the *Reviewer* add feedback and comments to your implementation to critique your work. The *Reviewer* should get you to question all scenarios and *edge-cases* that you have to consider before approving your Pull Request onto a dev or production branch.

You should complete and attach these checklists to any pull requests you create or file as an author or reviewer of a new feature in your project's repository. When you decide to merge a pull request please complete a checklist similar to the version of the checklists provided below:

#### Author Checklist

* [ ] Do the changes implement the correct feature?
* [ ] Did you implement and update your Unit Tests?
* [ ] Does a build from the Cmd Line Pass?
* [ ] Has the author tested changes?

#### Reviewer Checklist

* [ ] Do the changes implement the correct functionality?
* [ ] Are there a sufficient amount of Unit Tests included?
* [ ] Does the file's code-style and inline comments meet standards?
* [ ] Are the commit messages appropriate?

Collaborating with your team throughout the *Code Review* process is the most important part of the Pull Request workflow. When you create a Pull Request you initialize a collaborative review process that iterates between the updating and reviewing of your code. The process finishes with the *Reviewer* executing a merge of your code onto the next brange in the stage of deployment that your team defined.

**The Pull Request & Agile Code Review**

#### 

#### What have you done!?!?

If you have gotten this far I am impressed. You might still have a chance at completing your first production ready application soon enough. We still need to customize the logic in our starter template so that it matches the list of Lambda functions that [I promised you earlier that we would eventually get around to implementing](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToConfigureYourServerlessBackend.md#setup-serverless-framework-locally). We are not there yet, but we are really close. Let's take a second to put it all down into a sort of OnePager that we can use to refer back to these commands throughout future projects.

##### Serverless MicroService Implementation Best Practices

1. Setup local Serverless Environment:
   * Instal nvm and Node.js v10.14.1
   * Setup Editor & Install ESLint
   * Configure SublimeText3
2. Configure your Serverless Backend on AWS:
   * Decrease Application Latency with Warm Starts
   * Understand the AWS Lambda Runtime Environment
   * Register with AWS and configure your AWS-CLI
   * Setup the ServerlessFramework for local development
3. Infrastructure As Code, Mock Services, & Unit Testing:
   * Configuring your Infrastructure As Code
   * Configure an API Endpoint
   * Mocking Services before Deployment
   * Unit Testing & Code Coverage
   * Run Tests before Deployment

#### AWS Lambda Notes

There have been a few recent updates to the AWS Lambda service that you should be aware of. The Node.js AWS Lambda runtime environment now supports Node.js v.10.14.1. In your serverless.yml file you should declare the runtime you will use as runtime: nodejs10.x to make sure that you can deploy your Lambda function correctly and with the latest supported features.

Throughout the remaining articles in this tutorial series we will indicate the newest updates that we have discovered and have had to evolve with using a tag like this:

* UPDATE: *This will discuss the update in question*

#### UPDATE: .babelrc

Older projects running AWS Lambda v8.10 will have a .babelrc file with the following configuration settings that will no longer work:

{  
 "plugins": ["source-map-support", "transform-runtime"],  
 "presets": [  
 ["env", { "node": "8.10" }],  
 "stage-3"  
 ]  
}

**Tough Luck** right? Well we went through the headache of refactoring our projects so that we could report back to you and give you the easy way out of this mess in 30 minutes or less. Queue the [Domino's Pizza guarantee](https://www.dominos.co.in/hot-pizza-30-minutes-delivery-guarantee-at-dominos-get-pizza-hot)!

Either way you now have to replace you .babelrc file with the following set of configuration parameters:

{  
 "plugins": [],  
 "presets": [  
 ["env", {"node": "10.14.1"}],  
 "stage-3"  
 ],  
  
 "test": [  
 "jest"  
 ]  
}

Thank me later and send me some Bitcoin...

#### UPDATE: webpack.config.js

Going a bit further please ensure that the following setting are copied exactly the way they are specified below.

const slsw = require("serverless-webpack");  
const nodeExternals = require("webpack-node-externals");  
  
module.exports = {  
 entry: slsw.lib.entries,  
 target: "node",  
  
 // Generate source maps for the proper handling of error messages  
 devtool: "source-map",  
  
 // Since 'aws-sdk' is not compatible with webpack  
 // we must exclude all non dependencies  
 externals: [nodeExternals()],  
 mode: slsw.lib.webpack.isLocal ? "development" : "production",  
 optimization: {  
 // do not minimize code!  
 minimize: false  
 },  
 performance: {  
 // Turn off size warnings for entry points  
 hints: false  
 },  
 // Run babel on all .js files and skip those in node\_modules  
 module: {  
 rules: [  
 {  
 test: /\.js$/,  
 include: \_\_dirname,  
 exclude: /node\_modules/  
 }  
 ]  
 }  
};

#### Babel & Webpack. What you need to know

JavaScript over the years, has evolved into something that the programming community has labeled with a series of different and confusing acronyms, and mouths full of alphabet soup, to keep the uninitiated from braving the waters of the software engineering market.

Not really, but close. You see, what happens is that anytime you build something that goes into widespread use, you get these international organizations that get together and decide what is best for everyone in the world to adopt in terms of standards and best practices when working together to use a unified and codified language like JavaScript. In this case, that organization is ECMA International. ECMA is probably also owned by Sun Microsystems. Although this can all very easily turn into some kind of YouTube Conspiracy Video, due to the historic undertones that go back to the NetScape Navigator vs InternetExplorer Battles, this Planned Economy of sorts is good because it enforces a standard across the world so that all of us can together, experience the joys of writing OpenSource applications in what is really called ECMAScript!!!. The joy...

**Then why is it called ES6?**



Just take my word for it, JavaScript is really ES6... for now. The real issue for us is that ES6 is as awesome as the latest update to any codified language will ever be, and unfortunately for us, internet browsers are always a substiantial few releases behind the release schedule devised by the gods at Sun... I mean ECMA International.

This means that current browser technology is able to compile ES3 technology maybe ES5 in some cases which will never ever be Internet Explorer. Don't you even forget me when I say that you will forever kick yourself in the face in disgust when you realized that entire legions of your users will alway remain ardent users of **IE8**.

**womp-womp**

Anyway, this means that we need a tool that will help us *transpile* our new ES6 code into an older version of JavaScript that can be used by browser that at best will support ES5. This is the reason for our dependency on Babel and the .babelrc configuration settings above. Babel is what makes our handy ES6 logic dynamically viewable to the world in any of the silly web broswers that people use out there in the world.

Transpiling is an opportunity for JavaScript developers to say that they still have to compile their code. C++ developers will appreciate the beauty and humor in that one very efficient statement. Once all of ES6 code is compiled we will then need to use something called a module bundler. In our case we use Webpack which basically just takes chunks of your code and the libraries that it depends on to bundle them into one file that will be used in the browser.

And there you have it. That's why you need the webpack.config.js file set the way we have discussed above. Please take note or you probably won't be able to deploy anything. Don't get comfy yet, there's still a bit of banging your head on the walls to come.

#### UPDATE: package.json

Just to be safe I wanted to include the updated package.jsonfile we need to work with. Please copy the file below and don't be shy about changing the name in the Author field to your own if you have made it this far!

{  
 "name": "invoice-log-api",  
 "version": "1.7.3",  
 "description": "A serverless general ledger service.",  
 "main": "handler.js",  
 "scripts": {  
 "lint": "eslint .",  
 "test": "jest"  
 },  
 "author": "David P. Lopez",  
 "license": "MIT",  
 "repository": {  
 "type": "git",  
 "url": "https://github.com/lopezdp/invoice-log-api.git"  
 },  
 "devDependencies": {  
 "aws-sdk": "^2.350.0",  
 "babel-core": "^6.26.3",  
 "babel-eslint": "^10.0.1",  
 "babel-loader": "^8.0.6",  
 "babel-plugin-source-map-support": "^2.0.1",  
 "babel-plugin-transform-runtime": "^6.23.0",  
 "babel-preset-env": "^1.7.0",  
 "babel-preset-stage-3": "^6.24.1",  
 "eslint": "^5.15.0",  
 "eslint-config-standard": "^12.0.0",  
 "eslint-plugin-import": "^2.14.0",  
 "eslint-plugin-node": "^9.1.0",  
 "eslint-plugin-promise": "^4.0.1",  
 "eslint-plugin-react": "^7.12.4",  
 "eslint-plugin-standard": "^4.0.0",  
 "jest": "^24.8.0",  
 "serverless-offline": "^5.0.0",  
 "serverless-plugin-warmup": "^4.4.3-rc.1",  
 "serverless-webpack": "^5.1.0",  
 "webpack": "^4.16.2",  
 "webpack-node-externals": "^1.6.0"  
 },  
 "dependencies": {  
 "babel-runtime": "^6.26.0",  
 "source-map-support": "^0.5.12",  
 "stripe": "^6.17.0",  
 "uuid": "^3.3.2"  
 }  
}

## Deploy the ServerlessStarterService Template

We have finally gotten to the coolest part of this tutorial. The buildup is enormous, I know. Do not let your head explode on me yet, I Promise there is a point to all of this. We have already run npm install, run lint, and test on our [Serverless-Starter-Service](https://github.com/lopezdp/ServerlessStarterService) that we have cloned on our local machine; We also mocked the [Serverless-Starter-Service](https://github.com/lopezdp/ServerlessStarterService) locally with sls local invoke and our local environment responded with all of the appropriate Response: 200 OK [messages as expected](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToConfigure.IAC.Mocks.UnitTests.md#invoke-local-mock-example-with-serverlessstarterservice). Now it is time to deploy our *serverless + microservice* to see what we can do. **Are you ready?!?!**

Navigate into the [Serverless-Starter-Service](https://github.com/lopezdp/ServerlessStarterService) project directory on your local machine and execute the following command in your terminal:

* $ serverless deploy -v

Here is the result of the deployment; the output also includes the *Service Information* you need to consume the resources from the API you have just implemented on AWS!

### [Serverless-Starter-Service](https://github.com/lopezdp/ServerlessStarterService) Output

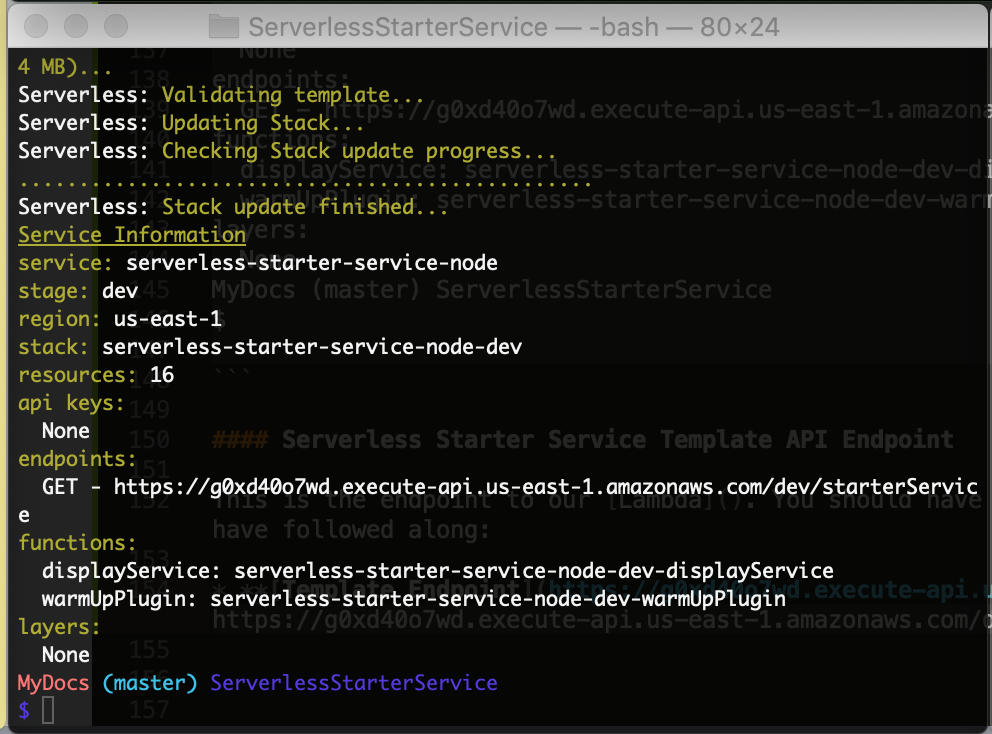
MyDocs (master) ServerlessStarterService  
$ serverless deploy  
Serverless: WarmUp: setting 1 lambdas to be warm  
Serverless: WarmUp: serverless-starter-service-node-dev-displayService  
Serverless: Bundling with Webpack...  
Time: 1008ms  
Built at: 2019-03-29 19:19:19  
 Asset Size Chunks Chunk Names  
 \_warmup/index.js 8.94 KiB 0 [emitted] \_warmup/index  
\_warmup/index.js.map 7.42 KiB 0 [emitted] \_warmup/index  
 handler.js 6.84 KiB 1 [emitted] handler  
 handler.js.map 5.82 KiB 1 [emitted] handler  
Entrypoint handler = handler.js handler.js.map  
Entrypoint \_warmup/index = \_warmup/index.js \_warmup/index.js.map  
[0] external "babel-runtime/regenerator" 42 bytes {0} {1} [built]  
[1] external "babel-runtime/helpers/asyncToGenerator" 42 bytes {0} {1} [built]  
[2] external "babel-runtime/core-js/promise" 42 bytes {0} {1} [built]  
[3] external "source-map-support/register" 42 bytes {0} {1} [built]  
[4] ./handler.js 2.58 KiB {1} [built]  
[5] external "babel-runtime/core-js/json/stringify" 42 bytes {1} [built]  
[6] external "babel-runtime/helpers/objectWithoutProperties" 42 bytes {1} [built]  
[7] ./\_warmup/index.js 4.75 KiB {0} [built]  
[8] external "aws-sdk" 42 bytes {0} [built]  
Serverless: Package lock found - Using locked versions  
Serverless: Packing external modules: babel-runtime@^6.26.0, source-map-support@^0.4.18  
Serverless: Packaging service...  
Serverless: Creating Stack...  
Serverless: Checking Stack create progress...  
.....  
Serverless: Stack create finished...  
Serverless: Uploading CloudFormation file to S3...  
Serverless: Uploading artifacts...  
Serverless: Uploading service serverless-starter-service-node.zip file to S3 (1.4 MB)...  
Serverless: Validating template...  
Serverless: Updating Stack...  
Serverless: Checking Stack update progress...  
................................................  
Serverless: Stack update finished...  
Service Information  
service: serverless-starter-service-node  
stage: dev  
region: us-east-1  
stack: serverless-starter-service-node-dev  
resources: 16  
api keys:  
 None  
endpoints:  
 GET - https://g0xd40o7wd.execute-api.us-east-1.amazonaws.com/dev/starterService  
functions:  
 displayService: serverless-starter-service-node-dev-displayService  
 warmUpPlugin: serverless-starter-service-node-dev-warmUpPlugin  
layers:  
 None  
MyDocs (master) ServerlessStarterService  
$

#### Serverless Starter Service Template API Endpoint

This is the endpoint to our Lambda. You should have something similar that will produce a similar result if you have followed along:

* [**Template Endpoint**](https://g0xd40o7wd.execute-api.us-east-1.amazonaws.com/dev/starterService)**:** https://g0xd40o7wd.execute-api.us-east-1.amazonaws.com/dev/starterService

**Deployment Output to terminal**



Triggering the new resource we deployed on API Gateway from the address bar in our browser to our new [**Template Endpoint**](https://g0xd40o7wd.execute-api.us-east-1.amazonaws.com/dev/starterService) at https://g0xd40o7wd.execute-api.us-east-1.amazonaws.com/dev/starterService will produce the following output to our device screen:

{  
 "message":"You are now Serverless on AWS! Your serverless lambda has executed as it should! (with a delay)",  
 "input": {  
 "resource":"/starterService",  
 "path":"/starterService",  
 "httpMethod":"GET",  
 "headers": {  
 "Accept":"text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8",  
 "Accept-Encoding":"br, gzip, deflate",  
 "Accept-Language":"en-us",  
 "CloudFront-Forwarded-Proto":"https",  
 "CloudFront-Is-Desktop-Viewer":"true",  
 "CloudFront-Is-Mobile-Viewer":"false",  
 "CloudFront-Is-SmartTV-Viewer":"false",  
 "CloudFront-Is-Tablet-Viewer":"false",  
 "CloudFront-Viewer-Country":"US",  
 "Host":"g0xd40o7wd.execute-api.us-east-1.amazonaws.com",  
 "User-Agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_14\_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.1 Safari/605.1.15",  
 "Via":"2.0 6ba5553fa41dafcdc0e74d152f3a7a75.cloudfront.net (CloudFront)",  
 "X-Amz-Cf-Id":"20\_\_-h2k2APyiG8\_1wFfAVbJm--W1nsOjH1m0la\_Emdaft0DxqzW7A==",  
 "X-Amzn-Trace-Id":"Root=1-5c9eac1d-58cadfb397aea186074bd6ab",  
 "X-Forwarded-For":"134.56.130.56, 54.239.140.19",  
 "X-Forwarded-Port":"443",  
 "X-Forwarded-Proto":"https"  
 },  
 "multiValueHeaders": {  
 "Accept":["text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8"],  
 "Accept-Encoding":["br, gzip, deflate"],  
 "Accept-Language":["en-us"],"CloudFront-Forwarded-Proto":["https"],"CloudFront-Is-Desktop-Viewer":["true"],  
 "CloudFront-Is-Mobile-Viewer":[  
 "false"  
 ],  
 "CloudFront-Is-SmartTV-Viewer":[  
 "false"  
 ],  
 "CloudFront-Is-Tablet-Viewer":[  
 "false"  
 ],  
 "CloudFront-Viewer-Country":[  
 "US"  
 ],  
 "Host":[  
 "g0xd40o7wd.execute-api.us-east-1.amazonaws.com"  
 ],  
 "User-Agent":[  
 "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_14\_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.1 Safari/605.1.15"  
 ],  
 "Via":[  
 "2.0 6ba5553fa41dafcdc0e74d152f3a7a75.cloudfront.net (CloudFront)"  
 ],  
 "X-Amz-Cf-Id":[  
 "20\_\_-h2k2APyiG8\_1wFfAVbJm--W1nsOjH1m0la\_Emdaft0DxqzW7A=="  
 ],  
 "X-Amzn-Trace-Id":[  
 "Root=1-5c9eac1d-58cadfb397aea186074bd6ab"  
 ],  
 "X-Forwarded-For":[  
 "134.56.130.56, 54.239.140.19"  
 ],  
 "X-Forwarded-Port":[  
 "443"  
 ],  
 "X-Forwarded-Proto":[  
 "https"  
 ]  
 },  
 "queryStringParameters":null,  
 "multiValueQueryStringParameters":null,  
 "pathParameters":null,  
 "stageVariables":null,  
 "requestContext": {  
 "resourceId":"bzr3wo",  
 "resourcePath":"/starterService",  
 "httpMethod":"GET",  
 "extendedRequestId":"XU\_UiEVSIAMFnyw=",  
 "requestTime":"29/Mar/2019:23:37:01 +0000",  
 "path":"/dev/starterService",  
 "accountId":"968256005255",  
 "protocol":"HTTP/1.1",  
 "stage":"dev",  
 "domainPrefix":"g0xd40o7wd",  
 "requestTimeEpoch":1553902621048,  
 "requestId":"8cef7894-527b-11e9-b360-f339559a98bd",  
 "identity": {  
 "cognitoIdentityPoolId":null,  
 "accountId":null,  
 "cognitoIdentityId":null,  
 "caller":null,  
 "sourceIp":"134.56.130.56",  
 "accessKey":null,  
 "cognitoAuthenticationType":null,  
 "cognitoAuthenticationProvider":null,  
 "userArn":null,  
 "userAgent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_14\_4) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.1 Safari/605.1.15",  
 "user":null  
 },  
 "domainName":"g0xd40o7wd.execute-api.us-east-1.amazonaws.com",  
 "apiId":"g0xd40o7wd"  
 },  
 "body":null,  
 "isBase64Encoded":false  
 }  
}

With the completion of this review and with the deployment of our ServerlessStarterTemplate, we are ready to proceed with the configuration of a proper *Continuous Integration & Continuous Deployment* pipeline that will allow us to abstract the deployment of our services a bit so that we can focus on implementing new features instead of maintaining infrastructure.

#### You have successfuly deployed your first serverless + microservice on AWS with the ServerlessFramework!

## Continuous Integration & Continuous Deployment Fundamentals

Moving forward with our project, and this tutorial, we can now take some time to discuss and understand the principles, practices, and benefits of adopting a *DevOps* mentality. We will also study and review concepts in Continuous Integration and Continuous Delivery and we will really start getting comfortable deploying enterprise ready software to the AWS Cloud. Just to make sure you are ready, we will review and get you comfortable with commiting your code to a *Version Control* repository on something like GitHub, and I'll show you how to setup a continuous integration server and integrate it with AWS *DevOps* tools like CodeBuild and CodePipeline.

"Opportunity is missed by most people because it is dressed in overalls and looks like work." - Thomas A. Edison

DevOps definitely conjures up the feeling of mysticism and confusion amongst those who discuss it in the cryptographic and vitriolic corners of the *Dark Web* and *Tor Browser* drum circles. If there was anything that you can call *The Force*, it would definitely be DevOps.

DevOps is really a *benevolent force* for cultural good that promotes collaborative working relationships between development teams, and their operational counterparts to work together to deploy and deliver software and infrastructure at a pace that allows the business units to cash in on the monetization of new features. The biggest benefit to this philosophy, is that the business teams can be sure that their efforts enforce the reliability and stability of the production environment holistically, because all of the key stakeholders are involved in the success of the product.

Implementing a DevOps mindset allows you to completely automate your deployment process, testing and validating every new feature that you implement using a consistent process-framework, starting with every developer on your team, all the way until the feature finds itself in use by your users, in *production*. This process is designed to eliminate IT silos of information across distributed teams of developers and operations teams. The idea is to breakdown the barriers that prevent teams from accessing information transparently, so that you can deploy infrastructure programatically, using a standard resource template, that allows you and your team to focus on developing new features and launching your product faster.

The idea is to apply software development practices like quality control, testing, and code reviews to infrastructure and feature deployment that can be rolled into *production* with little intervention and minimal risk. Transparency is prioritized so that every team member has a clear view at every stage of the development and deployment process from its implemetation by the dev team, all the way to the operations team that monitors and measures your application's resources and infrastructure deployed in *production*.

### Understanding Continuous Integration (CI)

Continuous Integration started with this evolution of collaborative ideas that we now call DevOps. Ine perfect world, you want you and your team of engineers implementing and *integrating* new features into your platform, *continuously*, i.e. **Continuous Integration**.

As you *continuously integrate* new features into your platform that will [*make the world a better place*](https://vimeo.com/98720197); hopefully, you and your group of *code monkeys*, have defined some kind of git commit style guideinto those needles [ContributionGuidelines.md that I told you about earlier](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToReviewServiceToConfigureCICDpipeline.md#code-review-agility). What you need to be doing is continuouslly using $ git commit to $ git push origin new-feature-branch-1 new features that you implement into some god forsaken form of central version control repository that you and your team use to share and review code.

When you or someone on your team implements a new change to the source code that you have saved on your central, and version controlled repository, that is hopefully on something *Free*, like GitHub, the changes you $ git push must complete a *build process* that includes an *automated unit testing phase* that we used jest.js to implement in a [previous discussion in this tutorial series](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToConfigure.IAC.Mocks.UnitTests.md#serverless-unit-testing--code-coverage) for our use case. The architecture that decided to implement is suppossed to give you immediate feedack about any changes that you introduce into the source code files that you save on GitHub.

Being able to obtain a response with instant feedback from the implementation paradim that we have tried to show you throughout this tutorial series, enables you and the rest of your team to fix and correct any mistakes, errors, or bugs that you find along the way, quickly, so that you and your team may continue moving forward and iterating over the development of your product, so you can take it to market **ASAP**! The whole point of **Continuous Integration** is to optimize the $ git merge topic-branch of new features into your source code so that you can stay focused on launching new products that help your business and operations teams measure bottom line growth. When you deliver quality software very fast,your business will make more money and will be able to afford to pay you you salary. Play ball because [there is no crying in Baseball](https://www.youtube.com/watch?v=cx2Sps9aMcY)!!!

**Continuous Integration (CI) Workflow**

#### 

#### Development

Visually, on the development side of the team, this would look like the image above with a developer making a change to a business feature deployed into production. The developer would commit the new code changes to the project's GitHub repository onto the master branch. The master branch is where all of the changes that the engineers on the team make to their topic branches flow into, once each Pull Request is reviewed and merged into *production*.

The code changes that your engineers will $ git merge into *production* after the Code Reviews are complete for each feature implementation, will automatically trigger a system **Build**. A system **Build** that your *Continuous Integration* pipeline triggers automatically when your **CI Server** detects your newest changes, will verify that your code will compile and execute successfully at runtime. Our Unit Tests that we implement will execute at this time, and will run against the new code changes that our **CI Server** detects when merged onto the master branch by your team. The goal used as a *Best Practice* in these scenarious is to have our **CI Servers** complete the *Build and Test* process quickly so that you and your development team get the immediate feedback you need to quickly iterate your way to a solution. The name of the game is speed, and launch your product ASAP.

#### 

#### Operations

When the idea of *Continuous Integration* devolved into this paradigm that we now call DevOps, it was originally only put into practice by engineering and devlopment teams within their Agile framework to deliver quality code faster. *Database Administrators*, *IT Operators*, and *Network Admins* didn't pay any attention to these models, and they moved on provisioning and autoscaling servers and load balancers on their own, in secret. Over time however, as the DevOps mentality and its transparent culture of collaboration amongst teams spread throughout the business realm, more teams adopted its philosophy. At some point, someone wrote a book about all of the headaches these companies were facing with these distributed teams of information silos, and one day, when CEO's were convinced of its ability to improve their bottom line, all of these software development patterns and practices percolated through the industry which led infrastructure and operations teams straight to their adoption.

Graphically, this means that you and your operations engineers can write all of your [infrastructure, as code](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToConfigure.IAC.Mocks.UnitTests.md#configuring-infrastructure-as-code), in a declarative YAML or json-formatted template and commit it to the same GitHub repository that your engineering and development teams are using to commit code to impolement new features. Following in the footsteps on the development teams, we then proceed to have our **CI Server** pull any of our infrastructure changes that we merge onto our *Version Control Repositories* to automate our build and unit testing processes. In the case of infrastructure we won't be compiling and code or triggering any unit tests. Instead, we will use the *build process* to spin up any *Linux Images (AMIs)* or resources that we need to have deployed to the cloud. Furthermore, instead of testing code, we will use the test phase to validate our YAML-formatted templates used by [CloudFormation] and to run infrastructure tests to be sure our application is listening on the expected network ports and that our enpoints are returning the expected http responses.

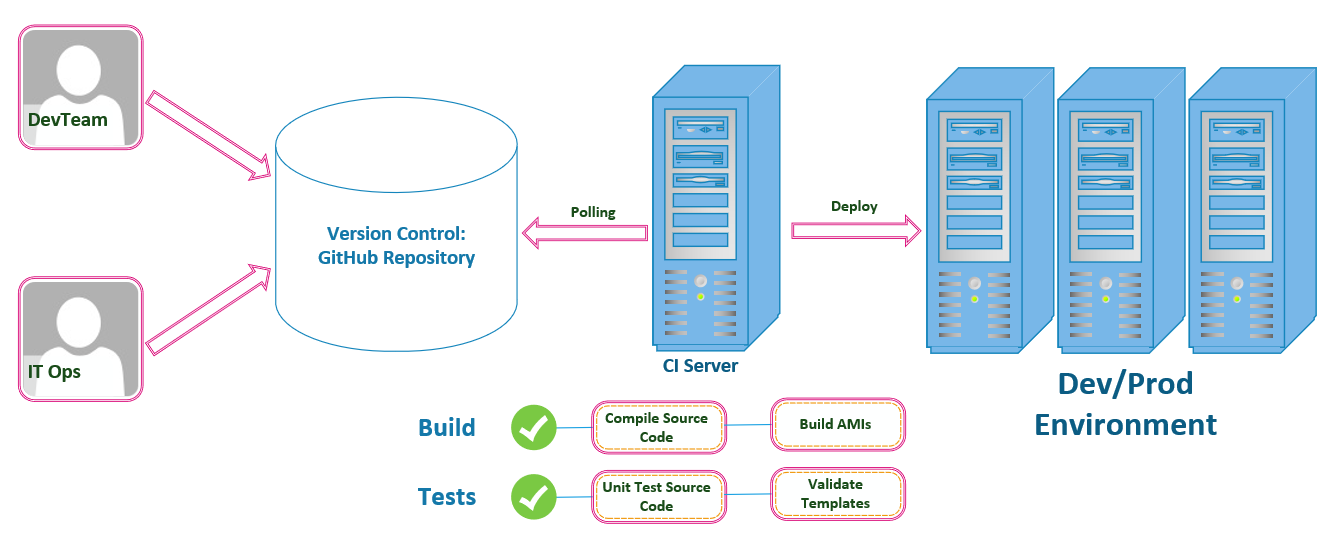
The goal here is the same as in development, to get feedback quickly; we want to get your infrastructure engineer immediate information pertaining to the status of the resources deployed. Your operations teams need to be able to anticipate any issues and respond to feedback immediately to correct any issues that may appear in *production*.

### Understanding Continuous Delivery (CD)

*Continuous Delivery* simply builds on the concept and process of *Continuous Integration* and is intended to automate the entire release of the application all the way on through the *production* environment. To execute this continuous integration and delivery lifecycle your team has to commit to practicing the DevOps culture transparently and in a consistent and Agile manner.

Development and operations teams *shall* implement the same model for versioning both application and infrastructure implemented as code to allow your *Continuous Integration (CI) Servers* to run automated *builds* and *tests*, that will trigger the deployment of new versions of our application to development branches before we decide to promote them into production ourselves. This brings up an important distinction in the different variations that we can choose to run our CI/CD pipelines which we will discuss shortly. For now please take a second to review a simple example of what a CI/CD Pipeline will look like:

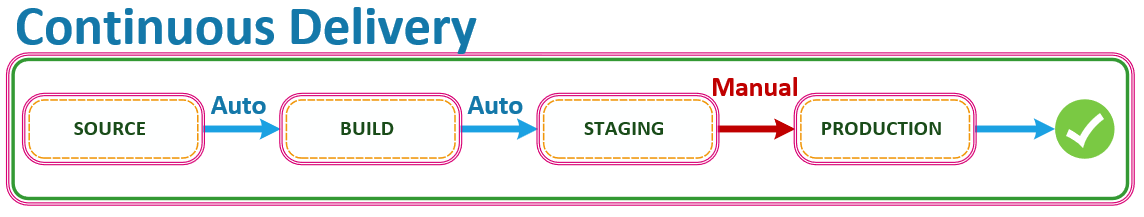
**This is a CI/CD Pipeline**



We will need to follow a workflow similar to the pipeline described above, on both the development and operations sides of our implementation teams. Both of our teams will have to have the discipline to follow a standard set of ContributionGuidelines.md as we have discussed in previous parts in this series to holds team members accountable and committed to pushing all of their changes to our GutHub repositories that we are using as our Version Control Framework. With you and the rest of the team committing changes to the repository that stores our project's source code, our *Continuous Integration (CI) Server* will initiate the *builds* and *tests* that will then trigger a deplpoyment to either a new or existing environment in *production*. This is our CI/CD Pipeline as shown in the image above.

The image does in fact simplify what we will be implementing over the course of this tutorial. We will also touch upon a few more complex examples throughout the rest of this tutorial. You can also customize your own CI/CD Pipeline to suit your own project's needs, to include as many stages as you may require to suitably deploy your application with minimal risk of failure.

**Continuous DELIVERY Stages**

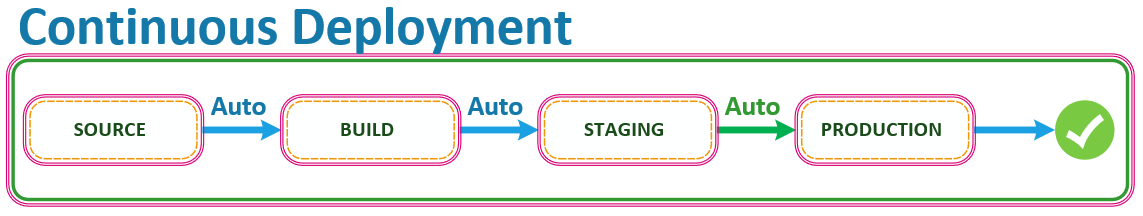


Looking at this implementation from the high level view of the Systems Architect, the above is what a CI/CD Pipeline variation looks like. As we have discussed, the first step in the process includes a source stage where you and your team will commit your changes to the source and any new feature implementations to your respositories on GitHub.

The next stage will run a build process that will compile your source code and spin up any AMI's, Lambda's, or other infrastructure that your Operations team has declared as code, and which your application needs to function at scale. This stage will also perform and run unit tests, and infrastructure template validations on the resources declared as code that you will deploy with CloudFormation. Unit Tests are executed within this build step to mitigate against any errors or bugs that are introduced by any changes to the source at this stage of the process. Your Unit Tests **SHALL** trigger a deployment into the staging environment next INSTEAD OF deploying into *production* automatically. When the *Final Release* of the application to the *production* environment is **NOT** executed *automatically*, then this is known as **Continuous DELIVERY**. Typically there is a **Business Rule** or **Manual Activity** completed before the final decision is made to release, and **promote** the new version into *production*.

Furthermore, you can add a stage that is run after Unit Tests are completed on your application's source code files that *may* load tests and execute them against your infrastructure to make sure that your application's performance is acceptable. The objective of all of this is to make sure that everything must be validated and shown to work as designed at each stage before it moves on to the next step of the delivery or deployment process.

**Continuous DEPLOYMENT Stages**



Here is where things get tricky and where everyone confuses the diferences between **Continuous Delivery** vs **Continuous Deployment**. Be careful, there are always decisions to make, and this is one of those times when a decision must be made. In having to choose a path, this is the exact point where the road begins to diverge for many DevOps Teams. In a Continuous DEPLOYMENT pipeline on AWS, the last step of moving into *production* is **automatic**. As long as each stage of the pipeline on AWS was successful, the code that you commit to GitHub will **ALWAYS** go through the pipeline and into *production* **AUTOMATICALLY**, assuming that all stages have completed and pased all testing successfuly.

In summary, in a Continuous **DEPLOYMENT** pipeline on AWS, the goal is to completely **automate** everything from *end-to-end*. The *commits* that we push to our GitHub repositories will *automatically* trigger a build phase and any appropriate *Unit Testing* on your appliction's source code and *infrastructure* as code. The workflow will culminate with the deployment of the application into either a *development*, or staging environment, where will eventually be pushing our application into *production* from, onl after having completed a MANUAL process or BUSINESS RULE validation with the project's principals.

In our specific use case as we work through the implementation of the PayMyInvoice application throughout the rest of this tutorial, we will be eliminating the manual task that provisions and configures our infrastructure, while making sure to define all of our application and infrastructure source code declaratively, to optimize our workflow with an implementation of a *Continuous Deployment* pipeline on AWS for our implementation of a *CI/CD* workflow for our application.

By pushing all of our application and infrastructure source code to a repository where we $ git commit all of our changes, you and your team will have the benefit and ability to see exactly what team member has changed and added to the source code. This transparency and versioning controlled by a central repository also allows you to rollback to previous versions of you software as need and in case of emergency.

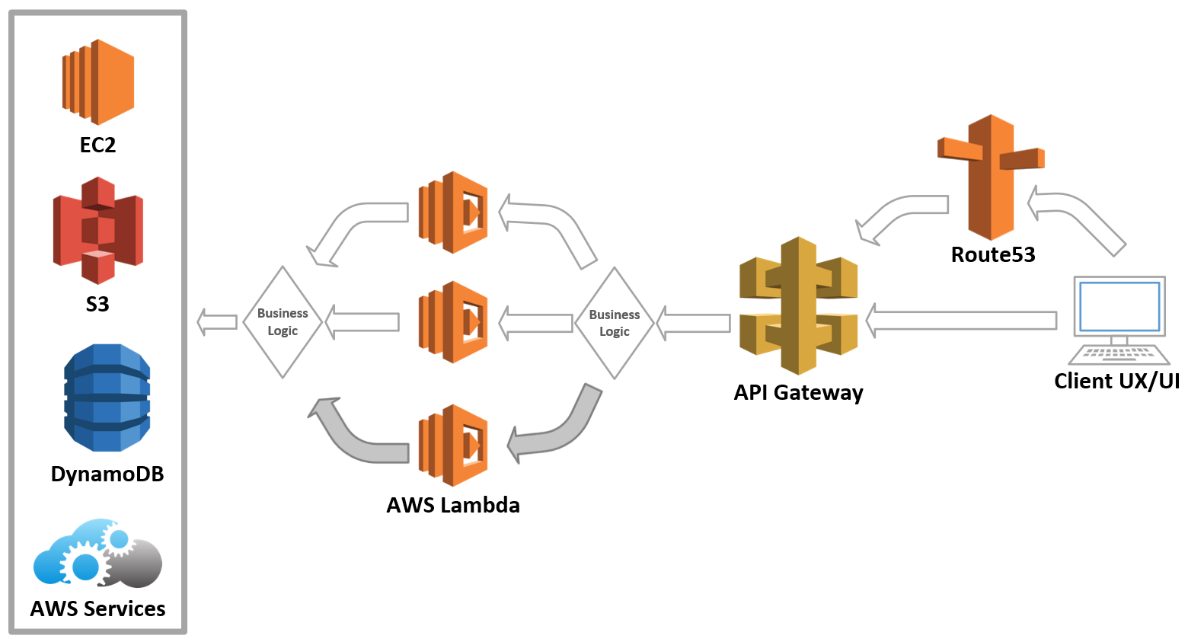
The implied goal of all of the processes is to *unify the software delivery* process so that our application and its infrastructure are treated as one object that we can run through our end-to-end automated testing build phase to validate all of our application, infrastructure, and configuration logic and provisioning is correct and to the project's requirements and specifications.

AWS CodeBuild and AWS CodePipeline are a few tools we will be using to implment our *CI/CD* pipeline. CodeBuild allows us to easily **AUTOMATE** our build and **DEPLOYMENT** to rapidly release new features and services. CodePipeline is a Continuous Delivery service that lets us model and visualize our solftware releases.

## Implementing Continuous Integration & Continuous Deployment on AWS

Now that you understand the fundamentals of **CI/CD** and how to foster a collaborative *DevOps* culture that will help your team work in a more *Agile* fashion, let's move on and take a look at how to implement these practices using AWS *DevOps* tools and services. Lets take a second to quickly review the basic architecture of a serverless application running on the AWS Lambda *Compute* service. Our *Continuous Deployment* pipeline that we will implement using AWS CodePipeline will automate the deployment of resources and application logic across this stack.

### Basic Serverless Architecture



Moving from right to left, and starting with the client, our functions running on Lambda will respond to the user's requests triggered by API Gateway through the http methods configured for each resource. Our *Infrastructure As Code* template will trigger CloudFormation to configure endpoints on API Gateway that will appoint a specific backend Lambda function to respond to requests made by the client. Moving along the pipeline we will implement, you will then configure Lambda to work with specific system resources on AWS like DynamoDB, Cognito, or your own *Custom Code* to be able to properly respond to the requests handled by API Gateway. The users of our application can access the API Gateway endpoints that you provision using either a direct URL or through a domain name that your register and configure through Route53.

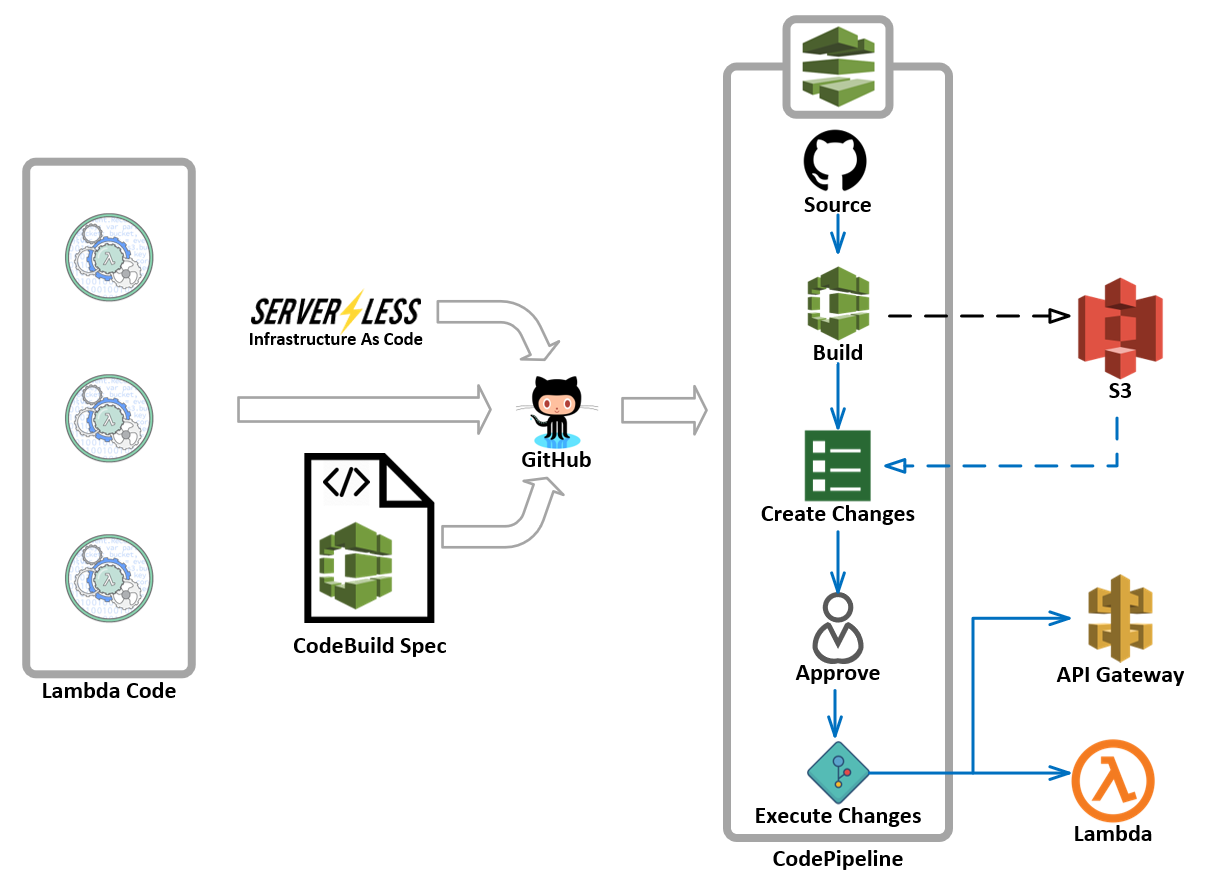
Our [Serverless-Starter-Service](https://github.com/lopezdp/ServerlessStarterService) mimics this workflow and we will make use of the endpoint created for us on API Gateway when we deployed with the ServerlessFramework's $ serverless deploy command from our terminal.

This is the endpoint to our Lambda. You should have something similar that will produce a similar result if you have followed along:

* [**Template Endpoint**](https://g0xd40o7wd.execute-api.us-east-1.amazonaws.com/dev/starterService)**:** https://g0xd40o7wd.execute-api.us-east-1.amazonaws.com/dev/starterService

Next we will describe the exact implementation steps on AWS for our **CI/CD Pipeline**.

### Continuous Deployment Workflow on AWS



Now that we have covered the fundamentals of *CI/CD*, the image above will help us understand the exact nature of the *CI/CD* workflow that we will implement on AWS. We will implement CodeDeploy and CodePipeline to complete our *Continuous Deplpyment* pipeline as shown above. Our *Pipeline* will start with a commit to our *Central Version Control* repository by you or anyone on your development team. Shown above, our *CI/CD* Pipeline will start when you trigger an event started by a $ git commit of any new source code to our GitHub repository with the **Lambda Code** you see in the image above.

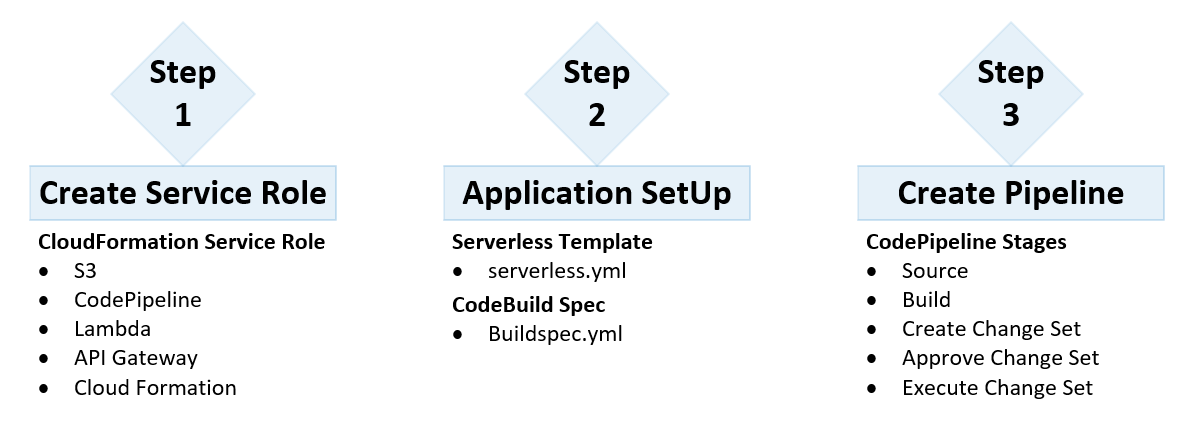
Aside from any new **Lambda Code**, or application code that we commit to our *central version control* repository (repo from now on), there are two additional files that we **shall** include for the setup of our Pipeline. You already know what our serverless.yml file does and how it works with CloudFormation to deploy new **Infrastructure As Code**, and now we are includeing a second file that you must call the: buildspec.yml. Your new buildspec.yml file is the input and configuration instructions we need to send to AWS to setup our CodeBuild project that will convert our serverless.yml file into the CloudFormation template that we need so that AWS can deploy the resources that our applications *need to breathe*!

When your buildspec.yml and your serverless.yml files are configured properly, and you $ git push them to your GitHub repo, you will trigger a Pipeline that will pull the *application code* and *infrastrastructure as code* from our GitHub repo that will store your *application logic* on an S3 bucket and begin to trigger an AWS CodeBuild project that will convert your serverless.yml file into a CloudFormation template. The phases of the Pipeline that we will implement look like this:

1. The CloudFormation template generated by our new CodeBuild project are the resources that we defined and the details for our business logic that we need to access with API Gateway. This **Build** phase that will pull from **Source**, will output an S3 url that will include the implementation of our *application code*.
2. Next, our implementation of CodePipeline will call CloudFormation and it will generate a CloudFormation ChangeSet that will *Create the Changes* that we want to deploy. Our serverless.yml created the CloudFormation template and the *build* files that we have in S3 are what we will use to generate this ChangeSet.
3. For the sake of this description, we will say that the next step will require a Manual Approval step as would be the case in a *Continuous Delivery* pipeline. A Manual Approval however is optional, and in a *Continuous Deployment* paradigm this step will be an automatic approval determined by the conditions that we will implement. Once this step is *Approved* however, our CodePipeline will **execute** the CloudFormation ChangeSet which will deploy the infrastructure and the new application logic that we want to implement and deploy for our applications.

As we $ git push new updates or features to our *serverless + microservice* application, our Pipeline will respond to events in our GitHub repo that correspond to changes in our *application* or *infrastructure as code*, and will automatically execute the new *build* and make the appropriate changes to our application immediately. Diving in a bit deeper, below are the steps we will need to accomplish to complete the configuration and setup of our Pipeline.

### Serverless Application Continuous Deployment Configuration Steps on AWS



As shown above, the first thing that we need to get done is to create an IAM service-role for CloudFormation that **must** include permissions to S3, CodePipeline, Lambda, API Gateway, and CloudFormation. This service-role is needed because CloudFormation needs our permission to create resources on our behalf. Without the protection of these service-roles your AWS account will be hacked by *Russian Bots* and your account will consist of bots that will take over the world without your help.

Next we will configure our *serverless + microservice*, its *application code* and *infrastructure as code* that we configure with our serverless.yml template that is used to launch and deploy our resources with CloudFormation. Included in this step is the implementation and configuration of our buildspec.yml which is our AWS CodeBuild Specification file.

The last step is to create our AWS CodePipeline which we can define with as many stages as we need. In the example we are using a Source, Build, Create ChangeSet, Approve ChangeSet, and an Execute ChangeSet series of stages for our implementations of our *CI/CD* pipeline for our [Serverless-Starter-Service](https://github.com/lopezdp/ServerlessStarterService).

### Implementation of CodeBuild and CodePipeline

We are going to deploy our application on AWS because we like it. We love being *Locked-In* and we love paying *Bezos* his cut. This might be a long tutorial but its worth it when you realize how easy it is to use SublimeText3 with AWS on a Mac over VisualStudio on your 10-pound PC.

Do yourself a favor and follow along. This is your *lottery-ticket*!

### Using the [PayMyInvoice](https://github.com/lopezdp/PayMyInvoice) App to Deploy our Pipeline

We will use the **Project Directory** that we defined in [Part 2, Setting Up Serverless Locally](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToConfigureYourServerlessBackend.md#setup-serverless-framework-locally) that we called PayMyInvoice, and we will start with the service that we named $ ~/PayMyInvoice/services/invoice-log-api. The serverless.yml file located in the project is of the same structure that discussed to implement our *Infrastructure As Code*. I will rely on the fact that you have been listening carefully throughout this entire tutorial, and that I will not have to review the elements of this file which is used by CloudFormation to deploy the AWS Services you need to run your application at scale. I will move right along into what I would argue is the most important file that we need to implement our *CI/CD* Pipeline effectively, the buildspec.yml.

#### Implement a buildspec.yml for AWS CodeBuild

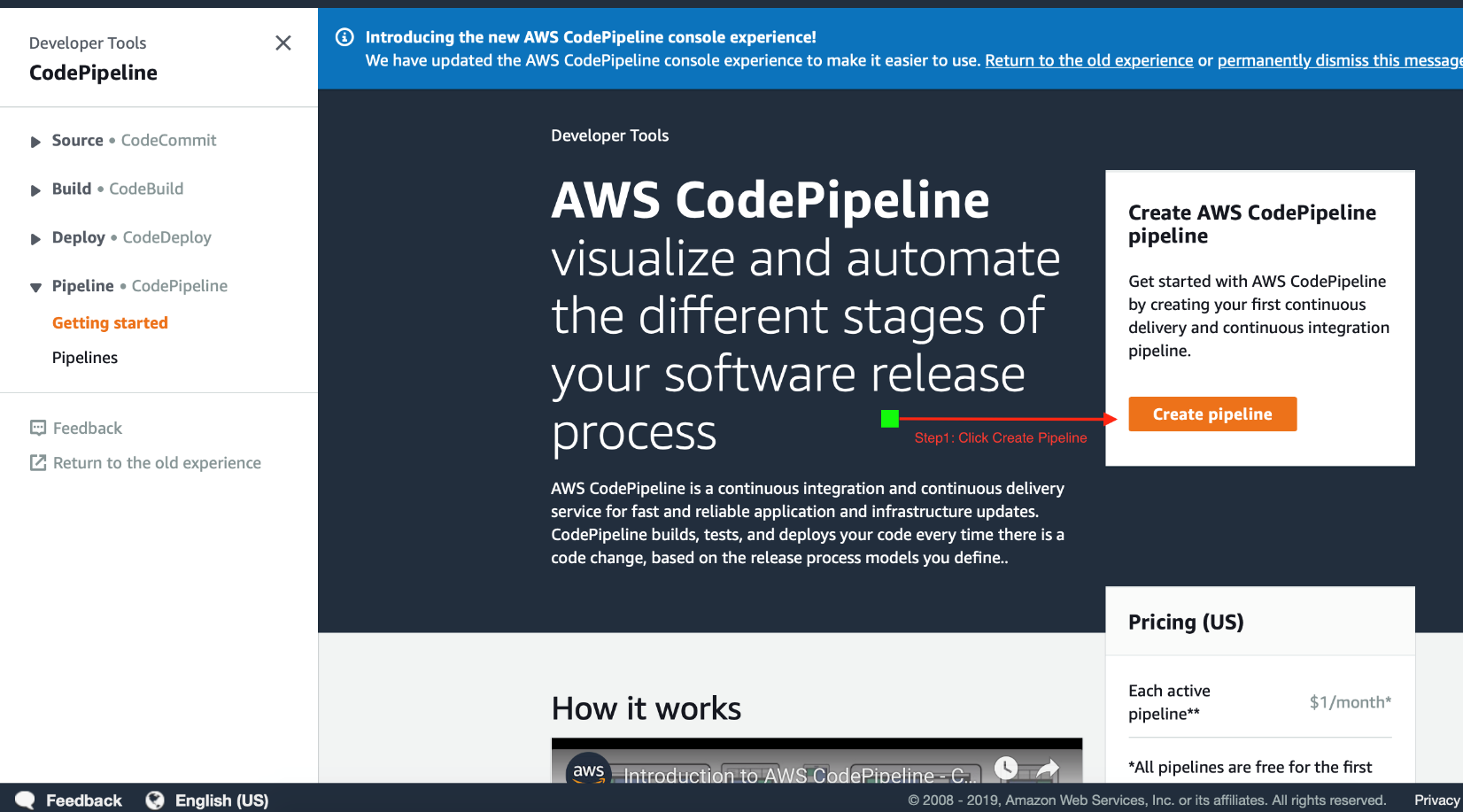
Go ahead and run $ touch buildspec.yml in your terminal from the ~/PayMyInvoice/services/invoice-log-api directory and add the following to the file and save in the root directory of the service to deploy:

# CI/CD buildspec-dev.yml file for invoice-log service on dev  
  
version: 0.2  
phases:  
 install:  
 runtime-versions:  
 nodejs: 10  
 commands:  
 - cd services/invoice-log-api  
 - npm install -g serverless  
 - npm install  
 build:  
 commands:  
 - serverless deploy -v --stage dev | tee deploy.out  
 #post\_build:  
 #commands:  
 #- npm test

#### Create a New AWS CodePipeline

Here you will need to navigate to the [CodePipeline Console](https://console.aws.amazon.com/codesuite/codepipeline/home), and proceed by clicking on the Create Pipeline button shown on the Dashboard.

**Step 1**

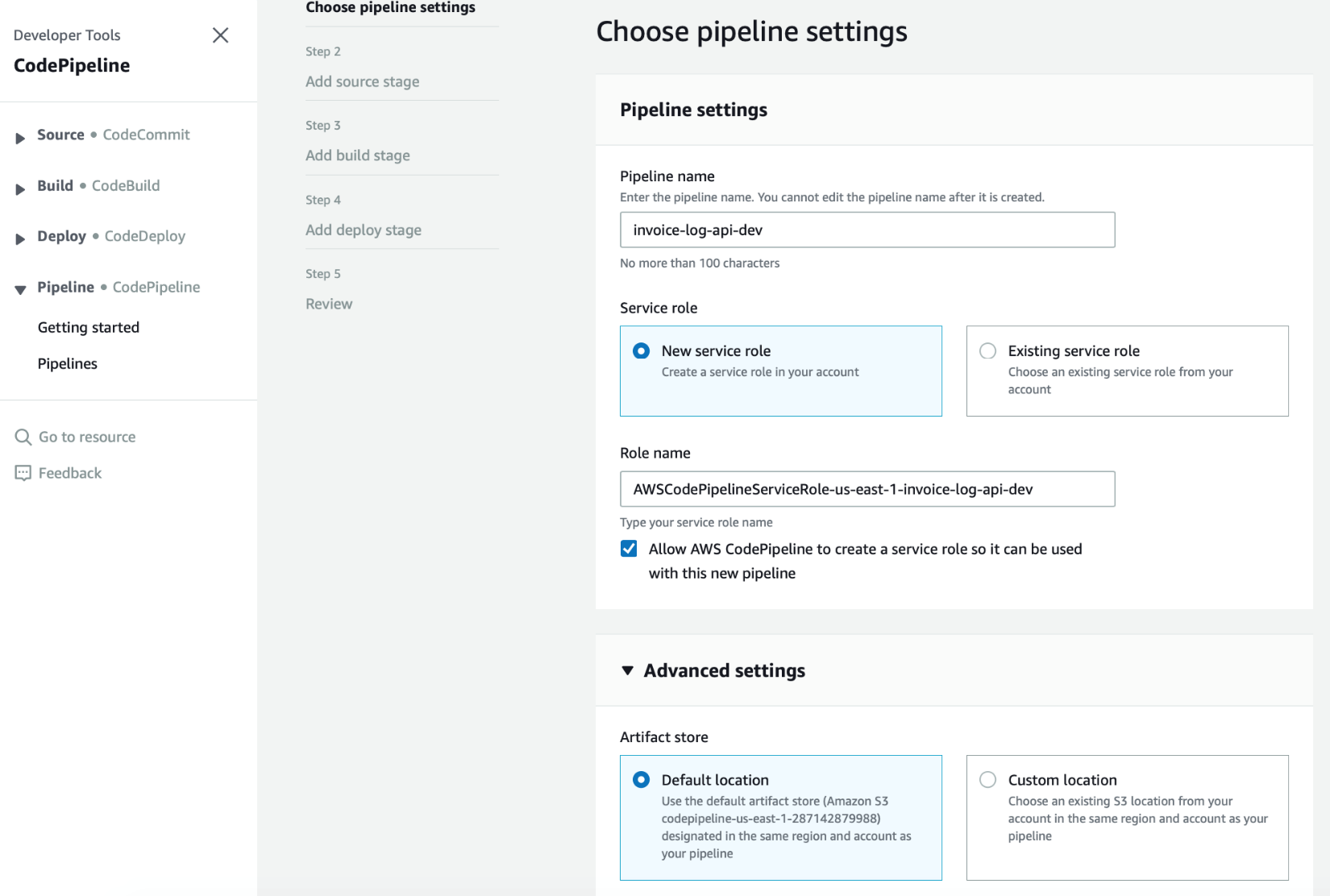


**Step 2**

Give your new CodePipeline a name to remember in the Pipeline Name field. In our use case we will name our Pipeline after the service that we will deploy on *CI/CD*. We will name this Pipeline: invoice-log-api-dev.

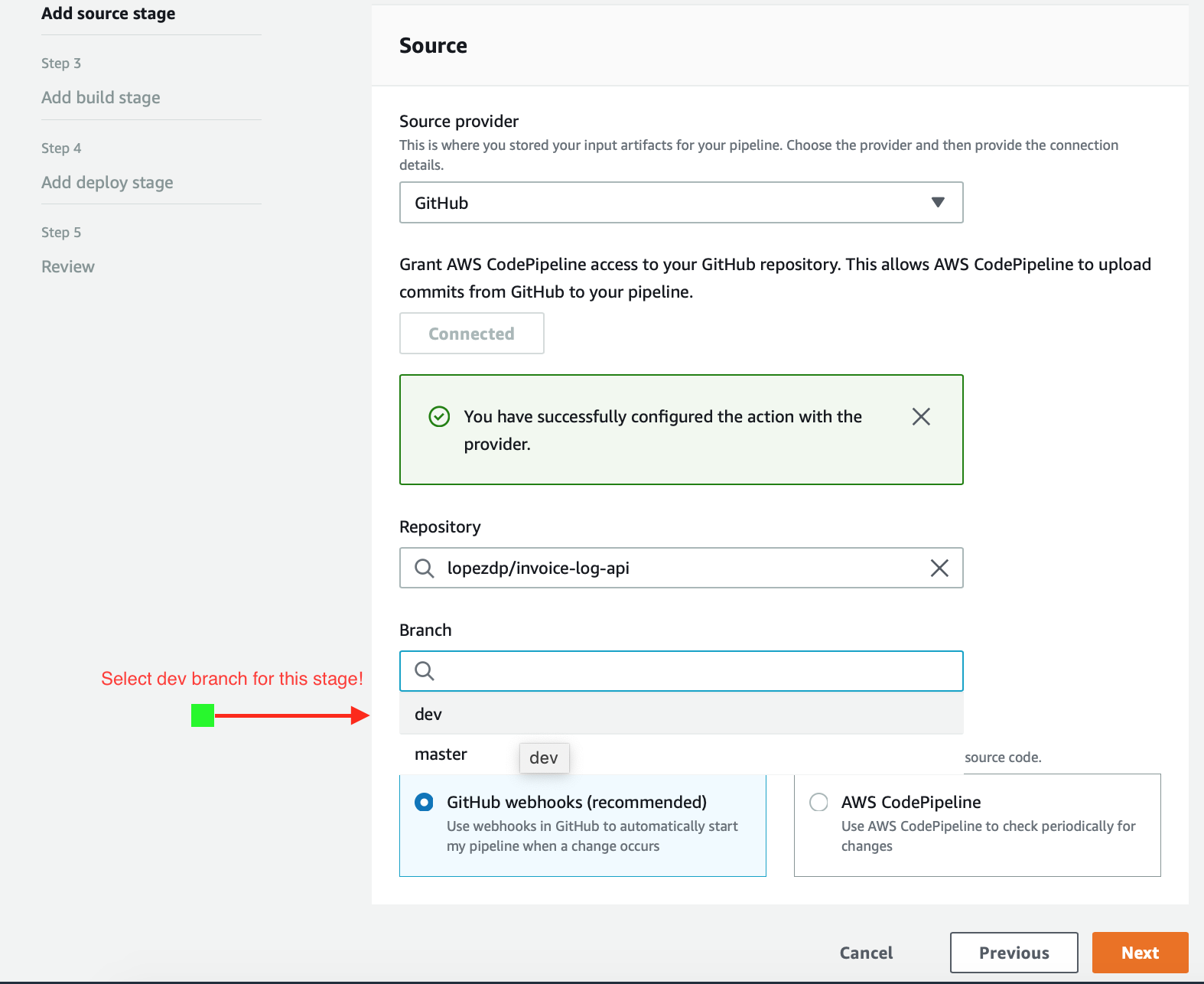
Click on the **New Service Role** selection and just let AWS Pipeline fill out the Role Name for you while letting it create the service role for you. You can also rename the Role name to something that makes more sense to you to better remember which roles you have allocated for the resources on your AWS account.

Lastly, leave the default selection for the *Artifact Store*. Keep It Simple, Stupid. **Click Next**!



**Step 3**

Here we need to define the Source repository where our application resides and where CodePipeline will have to listen in on for changes made by you and the rest of your development team. Choose GitHub as your SourceProvider then continue on by clicking the Connect to GitHub button shown in the image below.



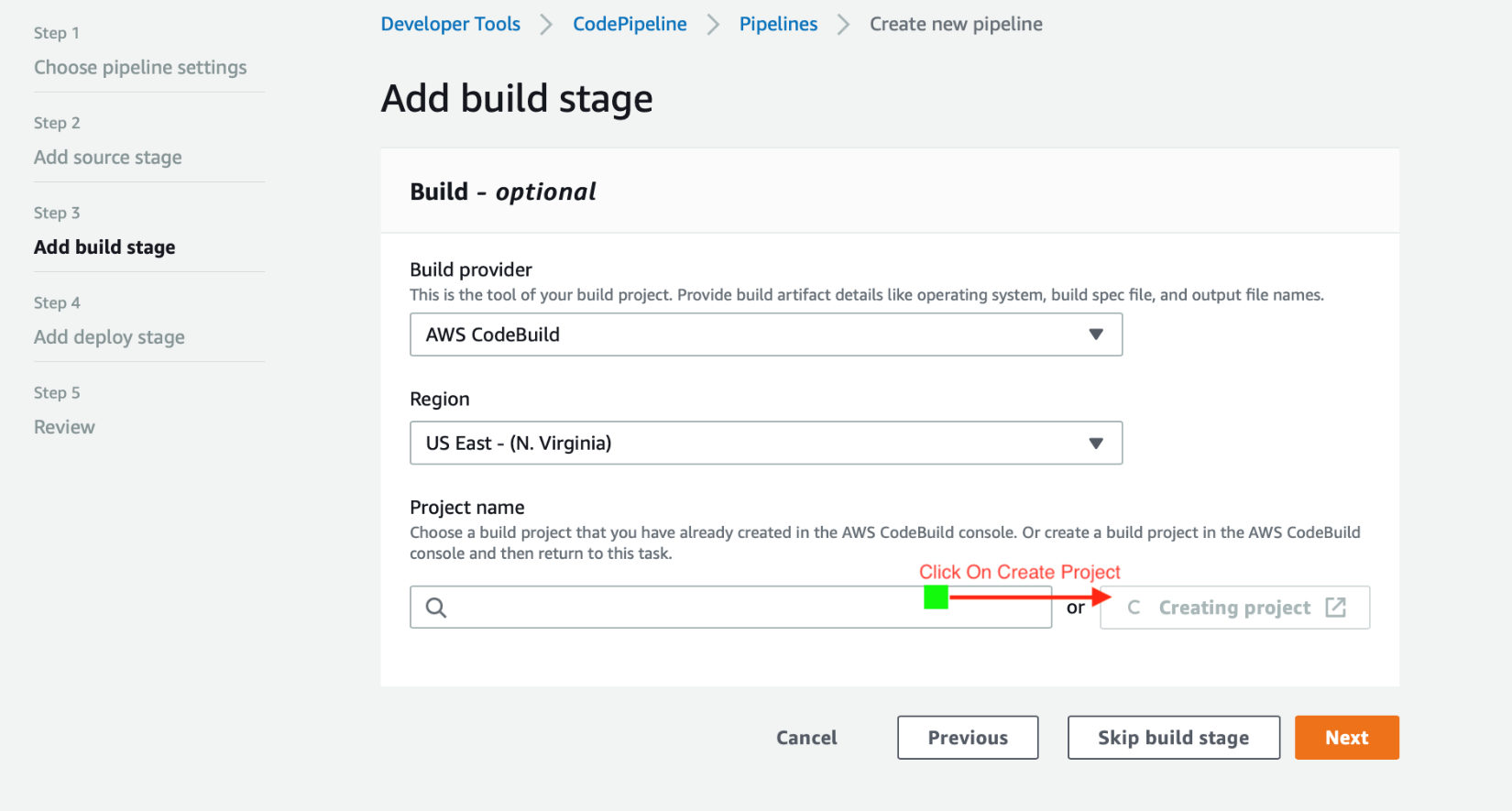
When you proceed to the Connect dialog, this will trigger an authenticated connection between GitHub and your new CodePipeline. Once connected, select the repo that you want to connect: invoice-log-api-dev. Next, select the branch that you want Pipeline to listen to, in our case this will be the dev branch.

Make sure you select the dev branch for this stage of our pipeline! Reserve the master branch for everything you push onto prod later!

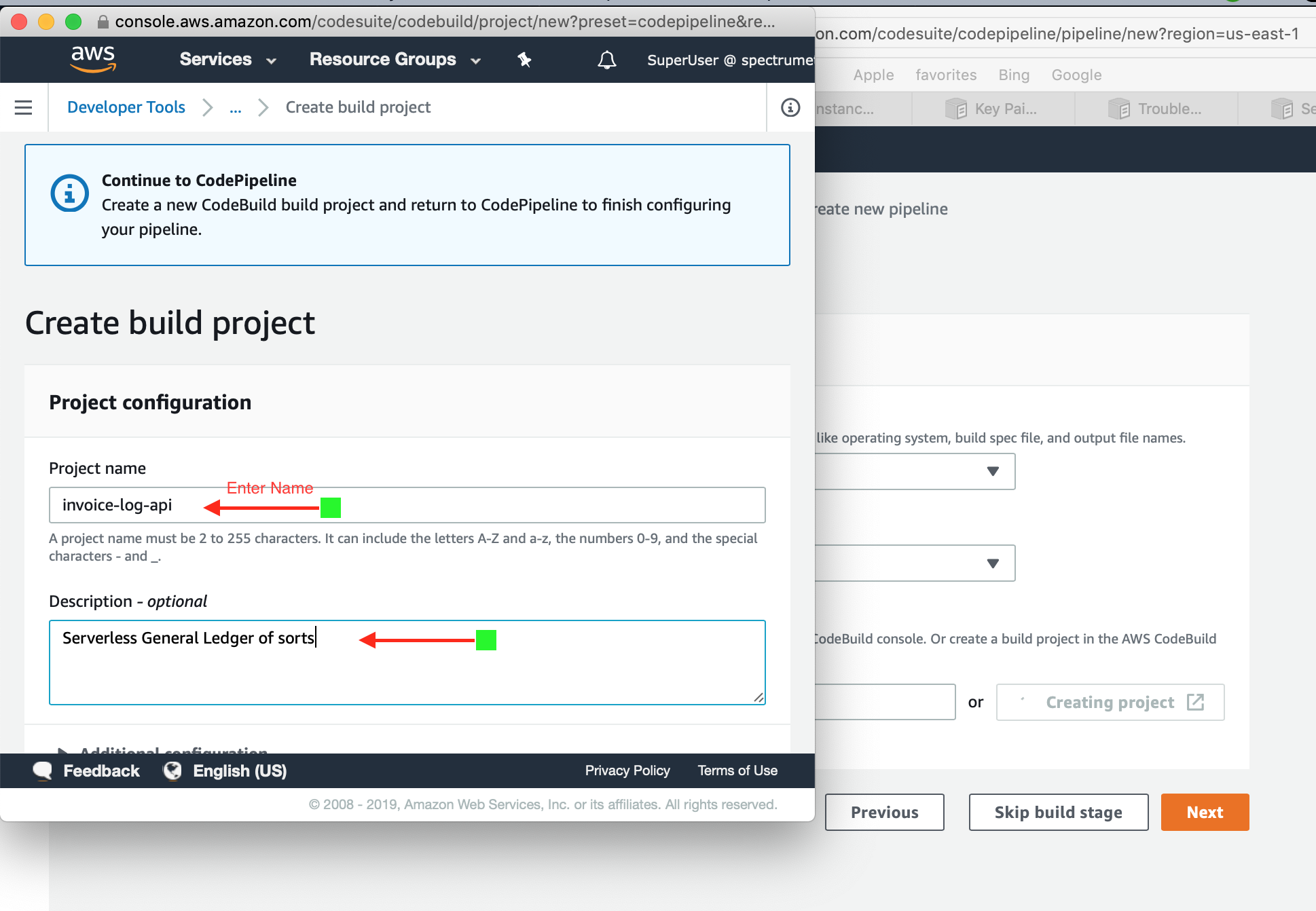
Use GitHub Webhooks to start our Pipeline and execute a build when changes occur. **Click Next** to complete this step.

**Step 4**

Now it is time to select the Build Provider we will use to build our source that CodePipeline will use to pull from our GitHub repo. We love the *mythical concept of AWS Lock-In*, and we have decided to use AWS CodeBuild as our provider. Select CodeBuild as your *Build Provider*, and enter the *Region* that is best for your location and constraints.

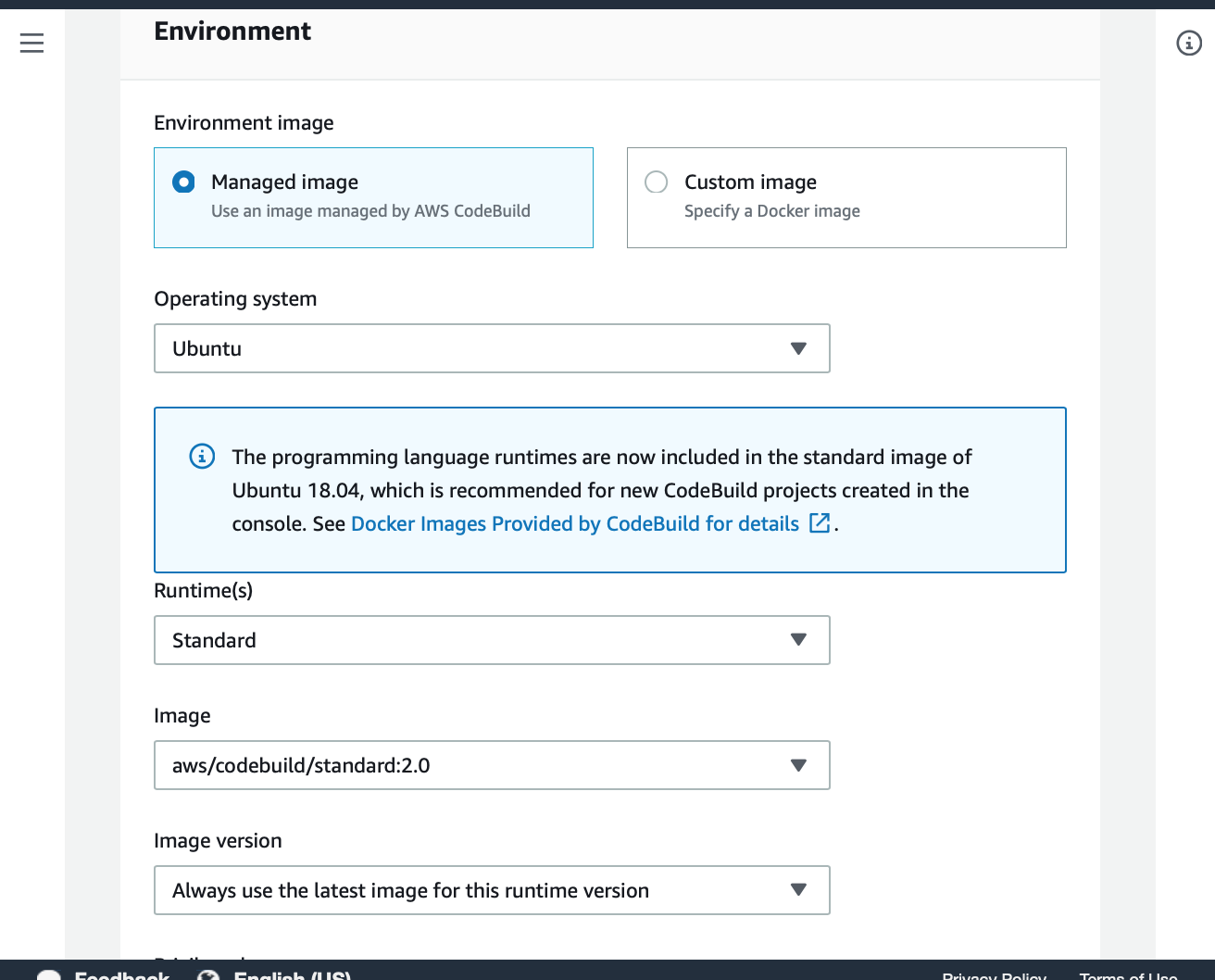


Next, go ahead and enter the *Project Name* that you have defined in CodeBuild, **or** click on **Create Project** as shown in the image. You will be taken to another window to complete the *Project* setup step. We are naming our build *invoice-log-api-dev*. This is the dev pipeline so be sure to label it accordingly!



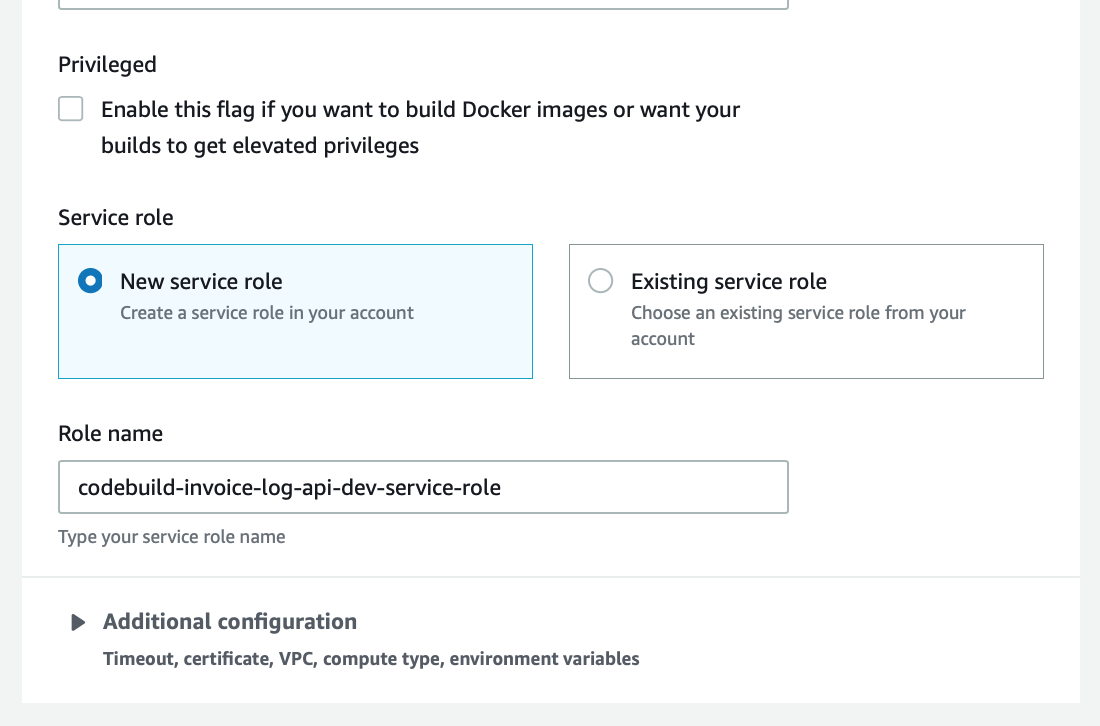
We have to manage our environment and specify the build properties we need to have setup for us by CodePipeline. You need to select an **Environment Image** that asks you to select **Use an image managed by AWS CodeBuild**, or a custom image. Select the default choice and go with **Use an image managed by AWS CodeBuild**. Your **Operating System** must be specified as an **Ubuntu** OS and your **Runtime** environment must be specified as **Standard**.

Next we must select an **Image** that we will use on the AWS Cloud to build our application. In our specific case because we like to keep our systems as updated as we possibly can be we are choosing to go with the aws/codebuild/standard:2.0 image selection. We will choose to keep our image version updated for this runtime version to complete this step. This is an important configuration step because it will require that all of our custom buildspec.yml files use the version: 0.2 declaration at the start of the file.

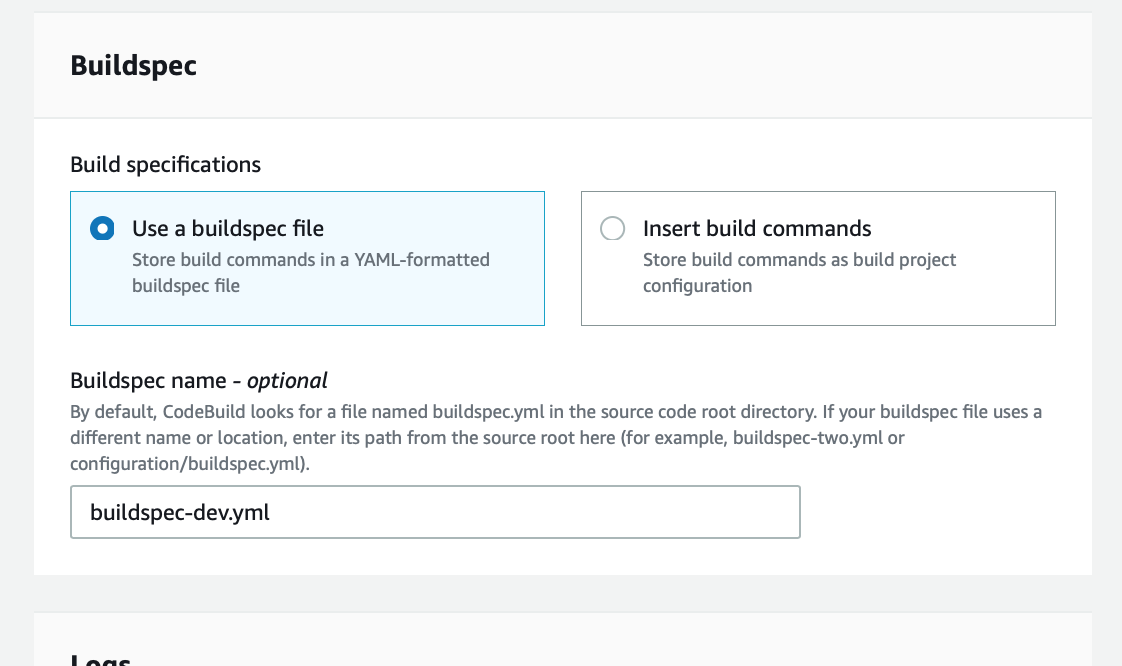


Next proceed to create a **New Service Role** as shown in the image below and choose a unique name for this role that you can remember later and make secure changes to as needed.

Please keep the Role name handy because we will need to find it within the AWS IAM service roles available so tjhat we can attach a few more permissions that this CodeBuild project will need to continuously deploy changes through out our pipeline. Make sure to allow AWS CodeBuild to create a service role so it can be used with this new Code Build Project on CodePipeline.

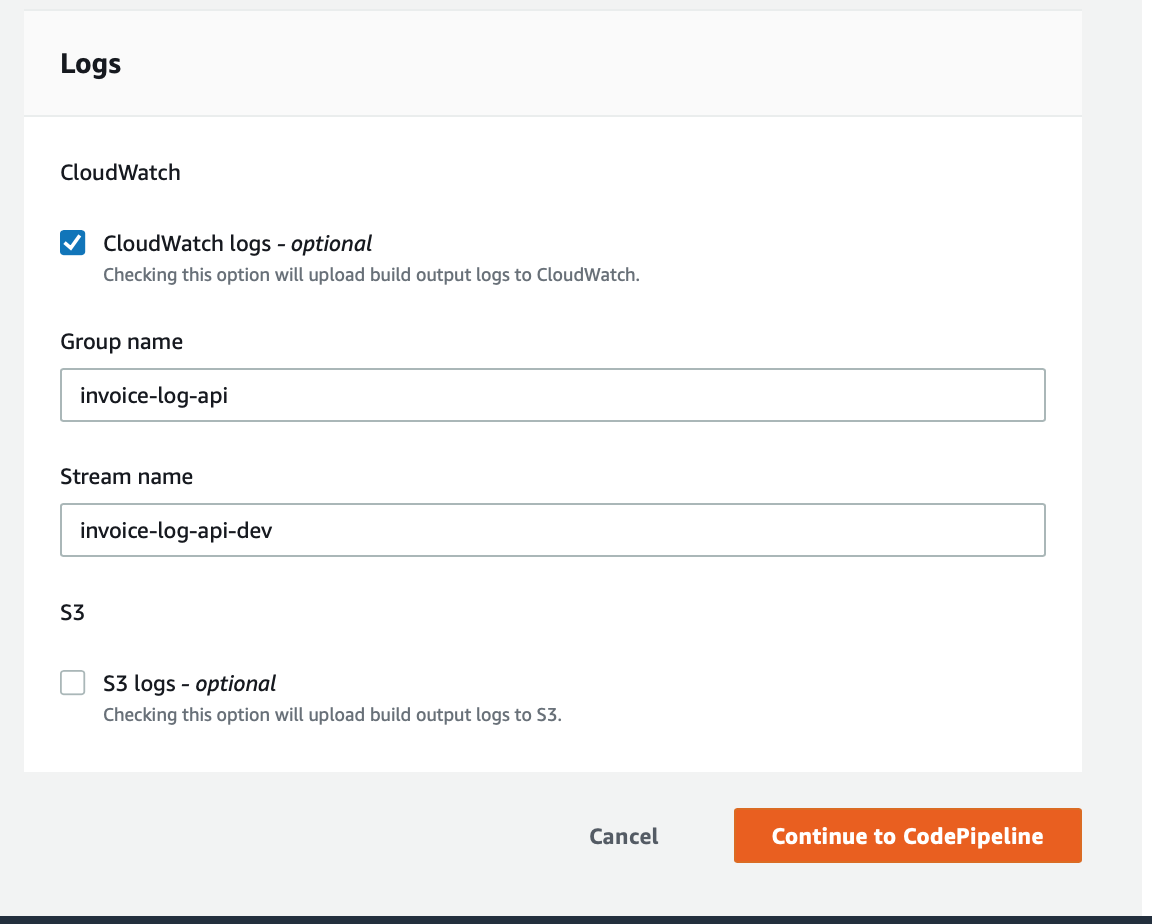


Make sure you tell Pipeline to **Use the buildspec-dev.yml in the source code root directory** when describing the environment build properties and configuration instructions. For our use-case, each pipeline that we implement will represent a build stage of our project that will map to a build phase on the front end that maps to a subdomain in dev, test, or prod, for example.

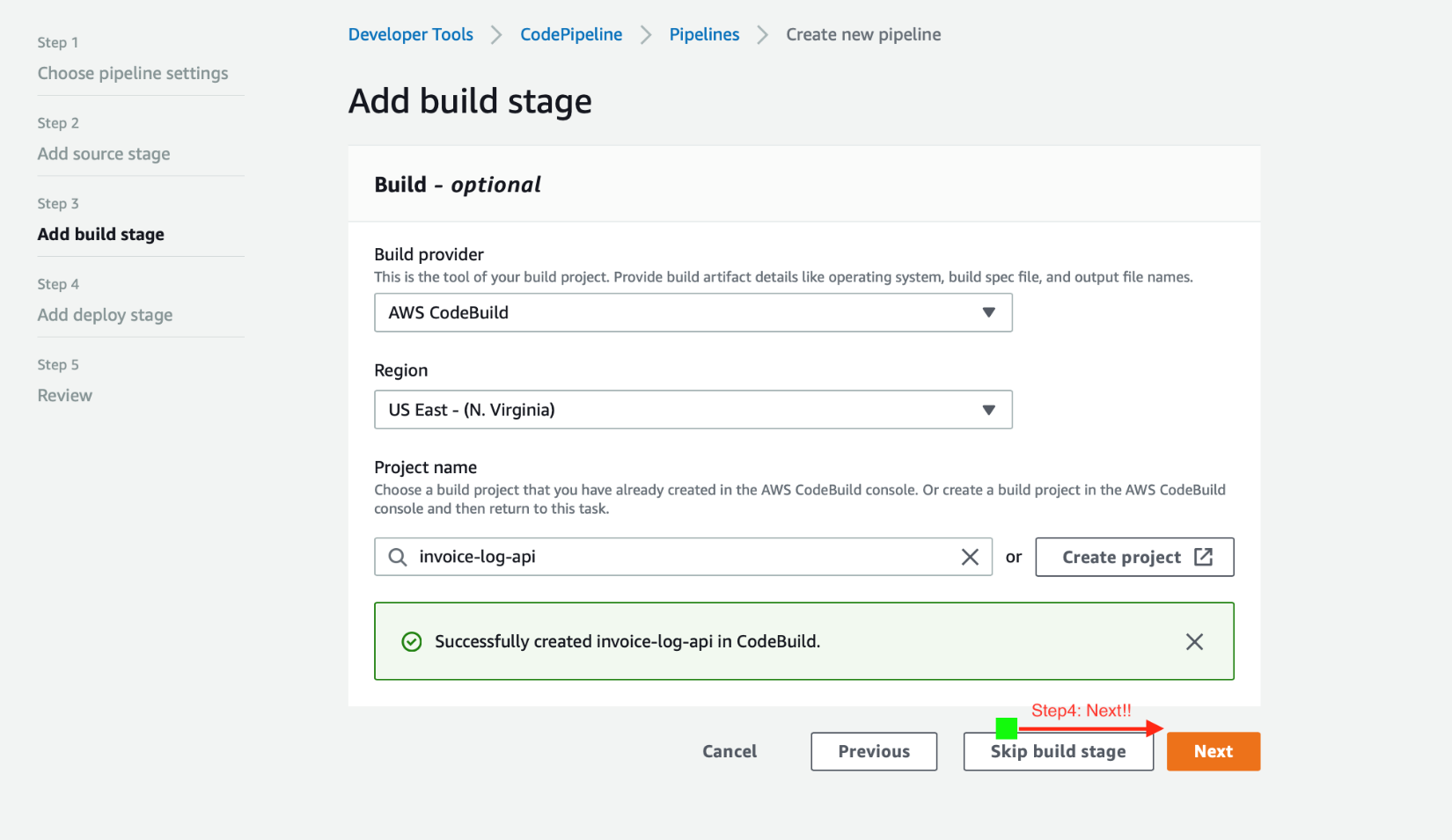


Select and use CloudWatch logs and give the group name the name of your service and differentiate the streams by the build phase which you plan to deploy each service on to make it easier to debug later. In our case our group name is invoice-log-api and of course our stream is uniquely identified by dev in this first launch of the pipeline.

Click **Continue to CodePipeline** and let's deploy this project.

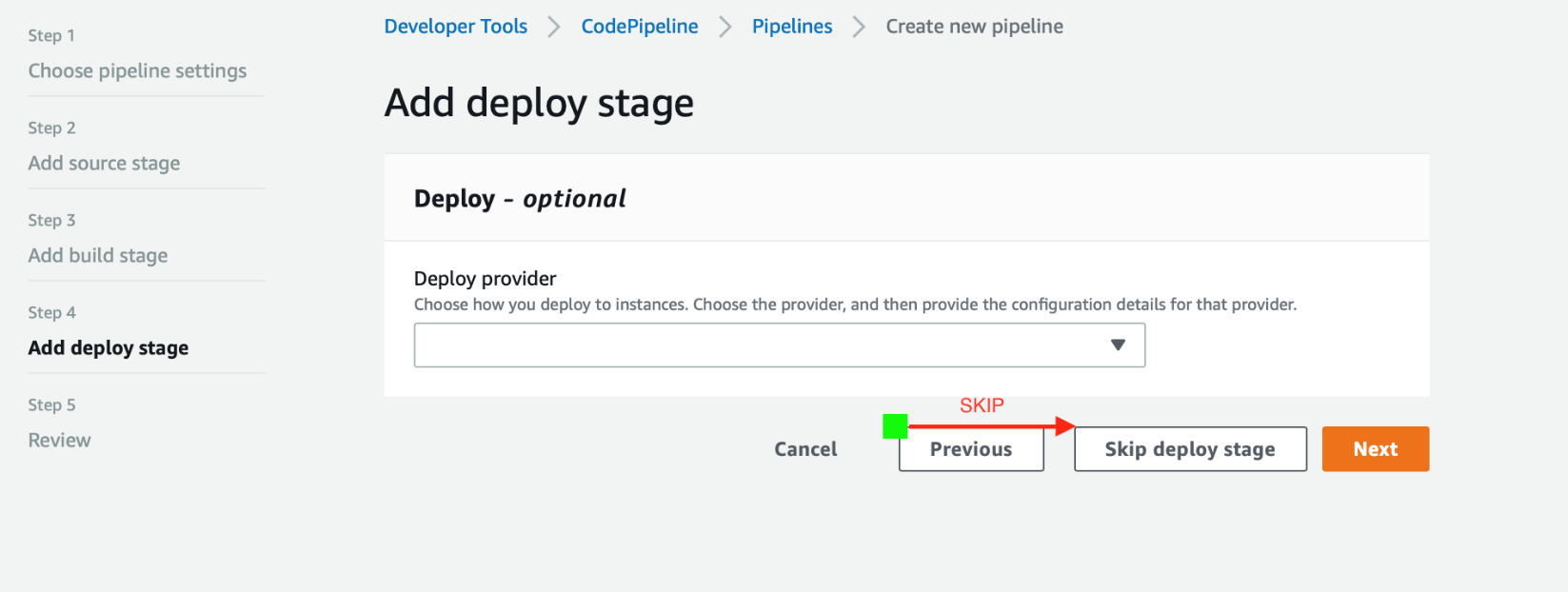


You should now see a message that says that you have *Successfully created invoice-log-api-dev in CodeBuild*. Go ahead and click the *Next* button to continue.



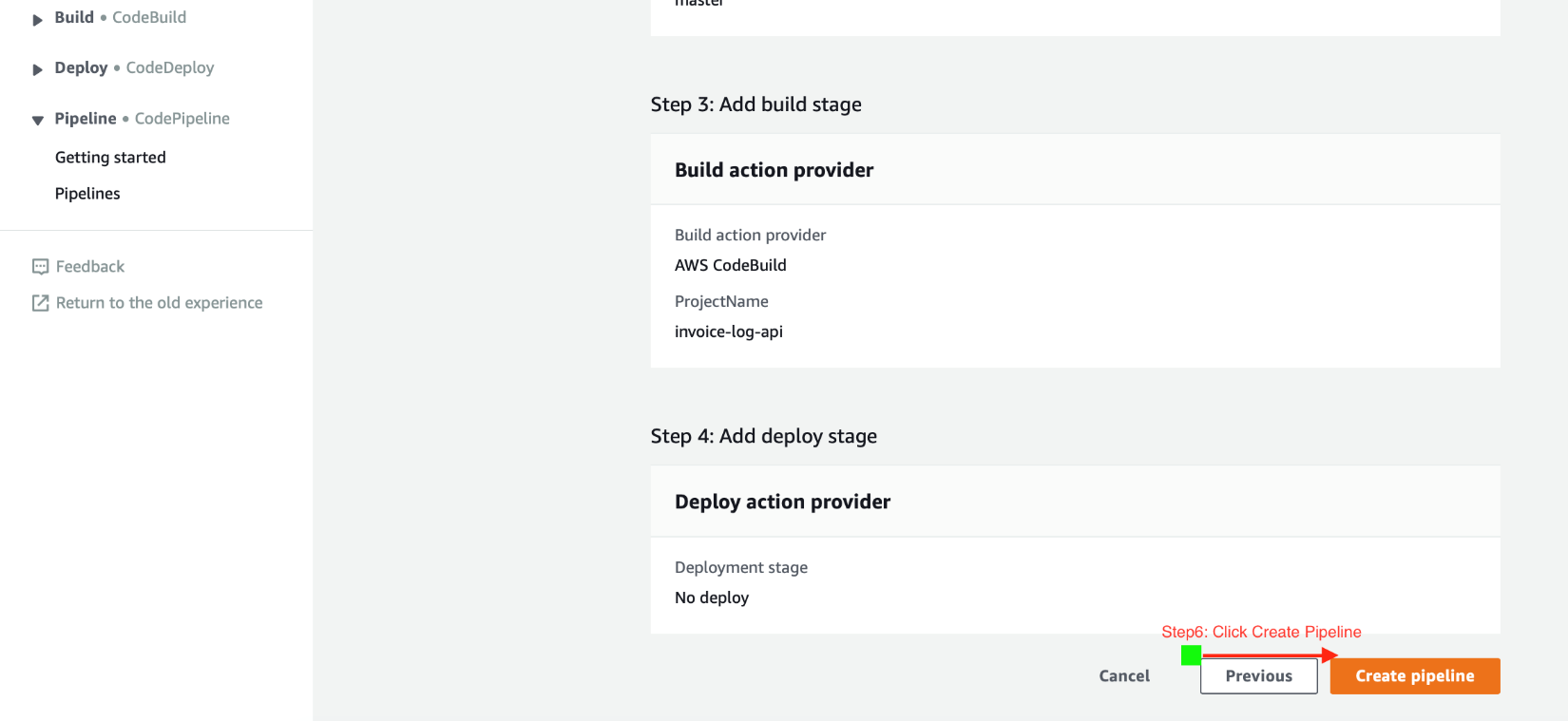
**Step 5**

We can skip the **Deployment Provider** for now because we are not using AWS CodeDeploy in this CodePipeline with the Serverless Framework.



**Step 6**

You can now review all of the setting you have configured for your CodePipeline. If everything looks *kosher*, go on and click on the **Create Pipeline** button as shown in the image below.



Your Build is now complete for the implementation of our **CI/CD** Pipeline on AWS with the ServerlessFramework.

#JediStatus

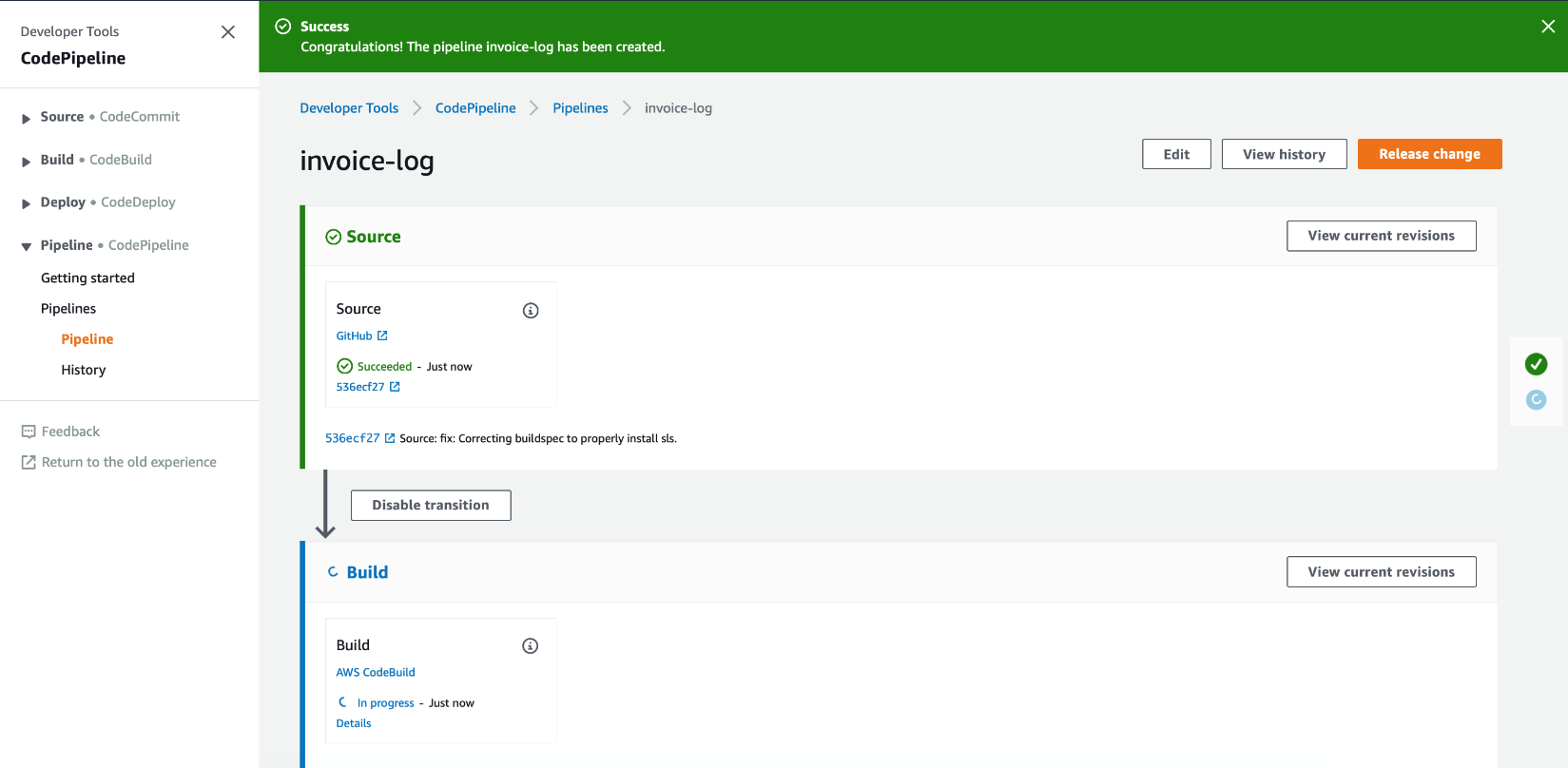
All we are doing with CodePipeline and CodeBuild in this pyrrhic exercise to *Continuously Integrate and Deploy* our application is to build a container that contains our Source code that CodePipeline will pull from out GitHub repo to run the commands declared in the implementation of our buildspec.yml. The commands we need to execute to complete our build stage are as follows:

# CI/CD buildspec.yml file for invoice-log service on dev  
version: 0.2  
phases:  
 install:  
 runtime-versions:  
 nodejs: 10  
 commands:  
 - npm install -g serverless  
 - npm install  
 build:  
 commands:  
 - serverless deploy -v --stage dev | tee deploy.out  
 #post\_build:  
 #commands:  
 #- npm test

The buildspec.yml will install the dependencies declared in our package.json file with the $ npm install command. During the build stage of our Pipeline, we will $ serverless deploy our Lambda services using the npm packages in our local copy of the node\_modules directory. We will execute our *Unit Tests* that we implemented with Jestjs.io during the post\_build stage using the same scripts that we have defined in our package.json file. In this case we will execute $ npm test before moving on to the next phase of our Pipeline.

## CI/CD CodePipeline Validation and Execution

Below is what your CodePipeline will look like once it is created. As you can see below, the Pipeline workflow will pull from the Source code located at your GitHub repository. The first time you create your Pipeline AWS will create an s3 bucket for you if you followed along with me. When you make changes in the future, CodePipeline will use this same s3 Bucket as the artifact store that it will use to pull the Source code to from your repo on GitHub. There is a lot going on here and I will review this article a few times to make sure I don't miss a critical step that will leave you in RPM Hell](<http://wiki.c2.com/?RpmHell>).

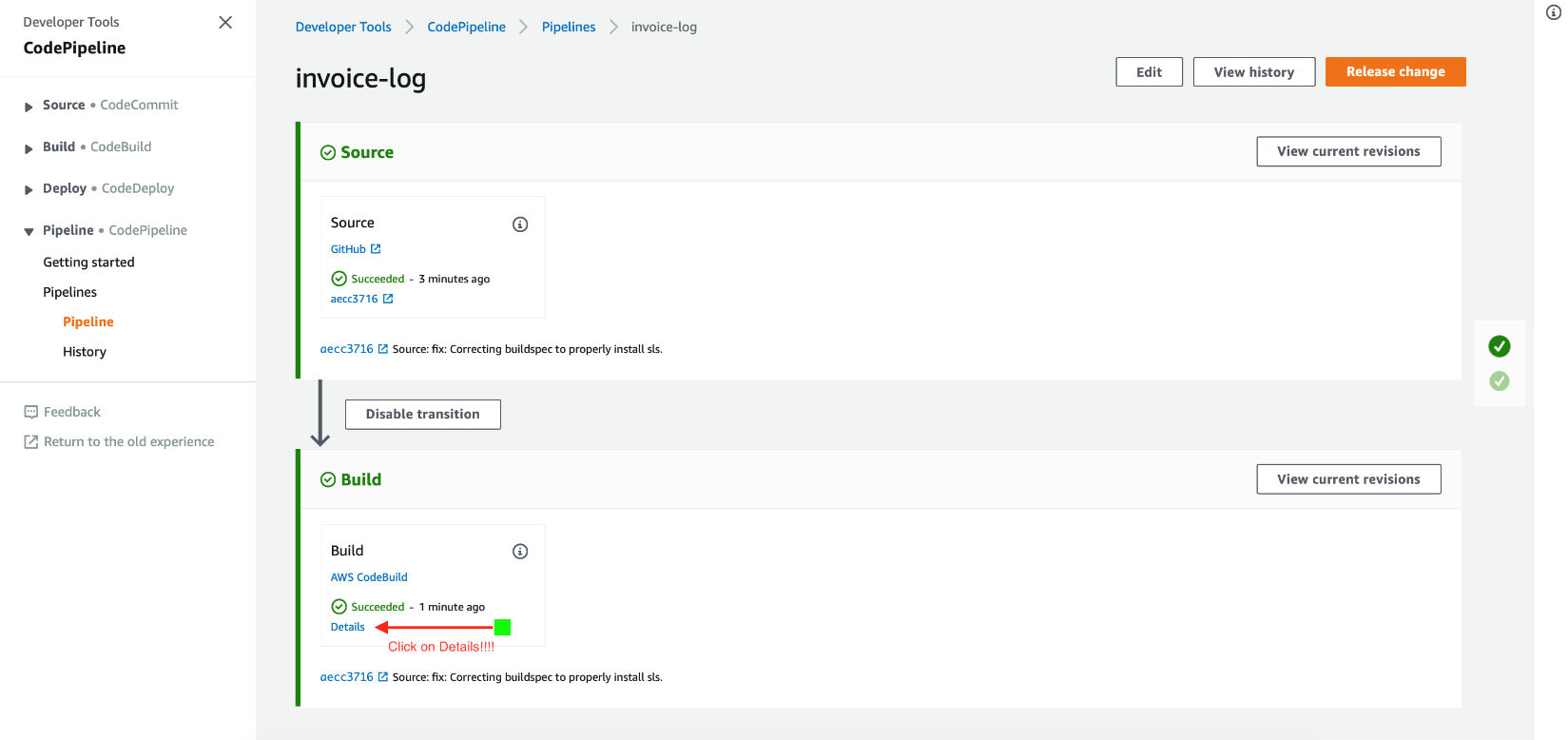


Once our Pipeline pulls from source, it will proccess each of the build phases that we defined in our buildspec-dev.yml file. During the install phase that we defined, CodeBuild will run $ npm install on our package.json file and it will also run $ npm install -g serverless and prepare our application's environment very much the same way we completed that task for our local machines in Part 1 of this series.. In this case we don't have to include ESLint or jsfmt because the container running our Lambda just doesn't care about the beauty of our code.

During the build phase, CodeBuild will literally build out our application using the ServerlessFramework tools we had it install in the install phase and it will deploy our Lambda's and backend services the same exact way we did it on our local machines... just, autonomously. Fear the Ai my friend, fear, the Ai.

In the post\_build phase, well we simply decided to run our unit tests. This was just to show you a very simple example and to highlight how flexible these stages can be. You can define these stages and run unit testing at any point along the workflow to deploy your services only when your tests come back positively. We've commented out testing and left some homework for you to do to see if you can take it a step further by implementing a prod environment also with Jest.

**Home Free!!**



You might think that you're home free at this point, and here's where the software Gods and Gremlins laugh at you dear sir:



sYour pipeline will initially show up as being successful... but you'll need to dig deep into the build details to review the build logs.

More than likely you'll find that you have gotten a permissions error because we never went back to give that CodeBuild Role the permission it needed to actually build our project!

**I pity the fool!**

Serverless: Packaging service...   
Serverless: Remove /codebuild/output/src7045676547/src/.webpack   
   
 Serverless Error ---------------------------------------   
   
 ServerlessError: User: arn:aws:sts::xxxxxxxxxxxxxx:assumed-role/codebuild-invoice-log-api-dev-service-role/AWSCodeBuild-0234rrfg-524a-4a32-9e00-adddddddd02 is not authorized to perform: cloudformation:DescribeStacks on resource: arn:aws:cloudformation:us-east-1:xxxxxxxxxxxxxx:stack/invoice-log-api-dev/05fa6f20-9afe-11e9-8fe6-12a1111111115d0   
   
 Get Support --------------------------------------------   
 Docs: docs.serverless.com   
 Bugs: github.com/serverless/serverless/issues   
 Issues: forum.serverless.com   
   
 Your Environment Information ---------------------------   
 OS: linux   
 Node Version: 10.16.0   
 Serverless Version: 1.46.1   
   
   
[Container] 2019/06/30 08:25:47 Phase complete: BUILD State: SUCCEEDED   
[Container] 2019/06/30 08:25:47 Phase context status code: Message:   
[Container] 2019/06/30 08:25:47 Entering phase POST\_BUILD   
[Container] 2019/06/30 08:25:47 Phase complete: POST\_BUILD State: SUCCEEDED   
[Container] 2019/06/30 08:25:47 Phase context status code: Message:   
 ```  
  
What happens here is that each of the build phases complete successfully, even with the error response and our commented out testing, and it tells CodePipeline that all of our scripts are complete. In another article we'll dig deeper into scripting more of these commands so that you can take more of the control from your command line and put it onto the cloud!  
  
When you look at the build logs however, you see that the Serverless Framework was not able to successfully deploy the Lambda functions we wrote using CodeBuild and CodePipeline due to not having the correct permissions.  
  
What we need to do is to click on the Services tab in the AWS navigation bar and head on over to IAM so that we can adjust the policies that this role associated to this CodeBuild project has access to.   
  
Inside of IAM you will want to click on Roles and find the Role that was created by CodeBuild when you created your project earlier. In our case the Role name was `codebuild-invoice-log-api-dev-service-role` and you can more easily find it by looking at the Trusted entities column to the right, and looking for an \*\*AWS Service: codebuild\*\* tag to more easily sort through the list.  
  
Click on the Role needed and click on \*\*Attach Policies\*\* to select \*AdministratorAccess\*. When selected proceed to give this policy for this code build project by clicking \*\*Attach Policy\*\*.   
  
When this is done you need to proceed to the CodePipeline Console and select the pipeline in question. When you are in the pipeline UI click on the \*\*Release Change\*\* Button on the top right so that you can trigger the pipeline to run the entire process from source to build and deployment on your `dev` branch of this new serverless + microservice backend.  
  
You know a successfull test by the following result in the build logs:

Serverless: Stack update finished...  
Service Information  
service: invoice-log-api  
stage: dev  
region: us-east-1  
stack: invoice-log-api-dev  
resources: 45  
api keys:  
 None  
endpoints:  
 POST - <https://tusgdsg6qg.execute-api.us-east-1.amazonaws.com/dev/invoices>  
 GET - [https://tusgdsg6qg.execute-api.us-east-1.amazonaws.com/dev/invoices/{id}](https://tusgdsg6qg.execute-api.us-east-1.amazonaws.com/dev/invoices/%7bid%7d)  
 DELETE - [https://tusgdsg6qg.execute-api.us-east-1.amazonaws.com/dev/invoices/{id}](https://tusgdsg6qg.execute-api.us-east-1.amazonaws.com/dev/invoices/%7bid%7d)  
 PUT - [https://tusgdsg6qg.execute-api.us-east-1.amazonaws.com/dev/invoices/{id}](https://tusgdsg6qg.execute-api.us-east-1.amazonaws.com/dev/invoices/%7bid%7d)  
 GET - <https://tusgdsg6qg.execute-api.us-east-1.amazonaws.com/dev/invoices>  
functions:  
 createInvoice: invoice-log-api-dev-createInvoice  
 getInvoice: invoice-log-api-dev-getInvoice  
 deleteInvoice: invoice-log-api-dev-deleteInvoice  
 updateInvoice: invoice-log-api-dev-updateInvoice  
 listInvoices: invoice-log-api-dev-listInvoices  
 warmUpPlugin: invoice-log-api-dev-warmup-plugin  
layers:  
 None

Stack Outputs  
UserPoolClientId: 52setdhgnfmhhfgsdhdy54639lol  
AttachmentsBucketName: invoice-log-api-dev-attachmentsbucket-tusgdsg6qg  
UserPoolId: us-east-1\_IJsfdgvbj  
UpdateInvoiceLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:96wragrqgareg255:function:invoice-log-api-dev-updateInvoice:2  
WarmUpPluginLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:96wragrqgareg255:function:invoice-log-api-dev-warmup-plugin:2  
IdentityPoolId: us-east-1:5e240536-4ce6-4422-8bfd-63e79d177510  
ListInvoicesLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:96wragrqgareg255:function:invoice-log-api-dev-listInvoices:2  
CreateInvoiceLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:96wragrqgareg255:function:invoice-log-api-dev-createInvoice:2  
DeleteInvoiceLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:96wragrqgareg255:function:invoice-log-api-dev-deleteInvoice:2  
GetInvoiceLambdaFunctionQualifiedArn: arn:aws:lambda:us-east-1:96wragrqgareg255:function:invoice-log-api-dev-getInvoice:2  
ServiceEndpoint: <https://96wragrqgareg255.execute-api.us-east-1.amazonaws.com/dev>  
ServerlessDeploymentBucketName: invoice-log-api-dev-serverlessdeploymentbucket-khwragveqrgafscv

Serverless: Removing old service artifacts from S3...  
Serverless Enterprise: Run serverless login and deploy again to explore, monitor, secure your serverless project for free.

[Container] 2019/06/30 09:18:13 Phase complete: BUILD State: SUCCEEDED  
[Container] 2019/06/30 09:18:13 Phase context status code: Message:  
[Container] 2019/06/30 09:18:13 Entering phase POST\_BUILD  
[Container] 2019/06/30 09:18:13 Phase complete: POST\_BUILD State: SUCCEEDED  
[Container] 2019/06/30 09:18:13 Phase context status code: Message:  
```

*I have changed the resource ID's to prevent scoundrels from killing the funds I have on my AWS account!*

This is the result of a successfull build and deployment to a stage and what it will look like once all build phases complete and all of your automated constraints to deplpoy your services are met.

### Troubleshooting

None of this comes without its issues and perpetual errors of course. Click on the details above, whether your Pipeline succeeded or failed, so that we can analyze a few common propblems that you may encounter out in the *wild*.

### You configured your serverless CICD on AWS CodePipeline. Good Luck!

## Part 5: Build a PayPal clone. Make money off your friends with the PayMyInvoice App!

* [Part 5: How To Implement and Deploy your own Serverless + MicroService to AWS](https://github.com/lopezdp/TechnicalArticles/blob/master/HowToBuildAServerlessMicroService.md) - *Not Published.*