VERACODE

Veracode Build Integration For AWS CodePipeline

Chris Campbell – Consultant Solution Architect

65 Network Drive, Burlington, Massachusetts 01803 339.674.2500 contact@veracode.com www.veracode.com

Table of Contents

Revision Control

- Version 1.0, 8th March 2019, first release

Introduction

Veracode offers a wide range of integrations 'out-of-the-box' for Continuous Integration/Continuous Delivery systems such as Jenkins, Azure DevOps, TeamCity etc. These integrations typically provide simple UI-driven configuration where possible that allows a development team to set up regular Static Analysis scanning of their applications directly within the application build pipeline. The delivery format for these integrations is via a 'plugin' that is installed into the CI/CD system.

For CI/CD systems where Veracode does not provide a plugin, a command line executable (AKA API wrapper, available as a Java .jar and a Windows .exe) is provided as an alternative integration option, allowing the same simple configuration as a plugin. This API wrapper takes parameters that define Veracode account credentials, the action to be performed (e.g. upload and scan), and any required configuration for the specific action. An example of running the Java version of the API wrapper in a bash terminal to perform an application scan follows.

```
java -jar /veracode/veracode-wrapper.jar -action uploadandscan -vid $VID -vkey
$VKEY -appname verademo-java -createprofile false -version $(date +%Y-%m-%d-%H:%M)
-filepath target/verademo.war -sandboxname aws-codebuild -createsandbox true
```

The Java API wrapper is published in the Maven Central Repository under the group ID com.veracode.vosp.api.wrappers. This version of the wrapper can be used within a CI/CD pipeline in a variety of ways. Options include downloading the jar to the CI/CD pipeline execution server in advance to be used by any future pipeline jobs, downloading the jar as a specific step within the pipeline for use in that job alone, or pre-installing the jar on a Docker image for use in systems that allow jobs to be run in a Docker container.

AWS CodePipeline is a CI/CD system that leverages the various software build and deployment tools provided by AWS. In particular, CodePipeline is capable of referencing AWS CodeBuild projects, which enable the use of Docker containers to perform actions within a CI/CD pipeline. CodePipeline has the

ability to share ‘artifacts’ between stages of the pipeline, via a pre-configured AWS S3 bucket, allowing the output of a software build to be provided to a CodeBuild Docker container. This configuration is the basis of how this document describes the steps required to integrate Veracode Static Analysis testing into CodePipeline. The final CodePipeline configuration is shown in figure 1.

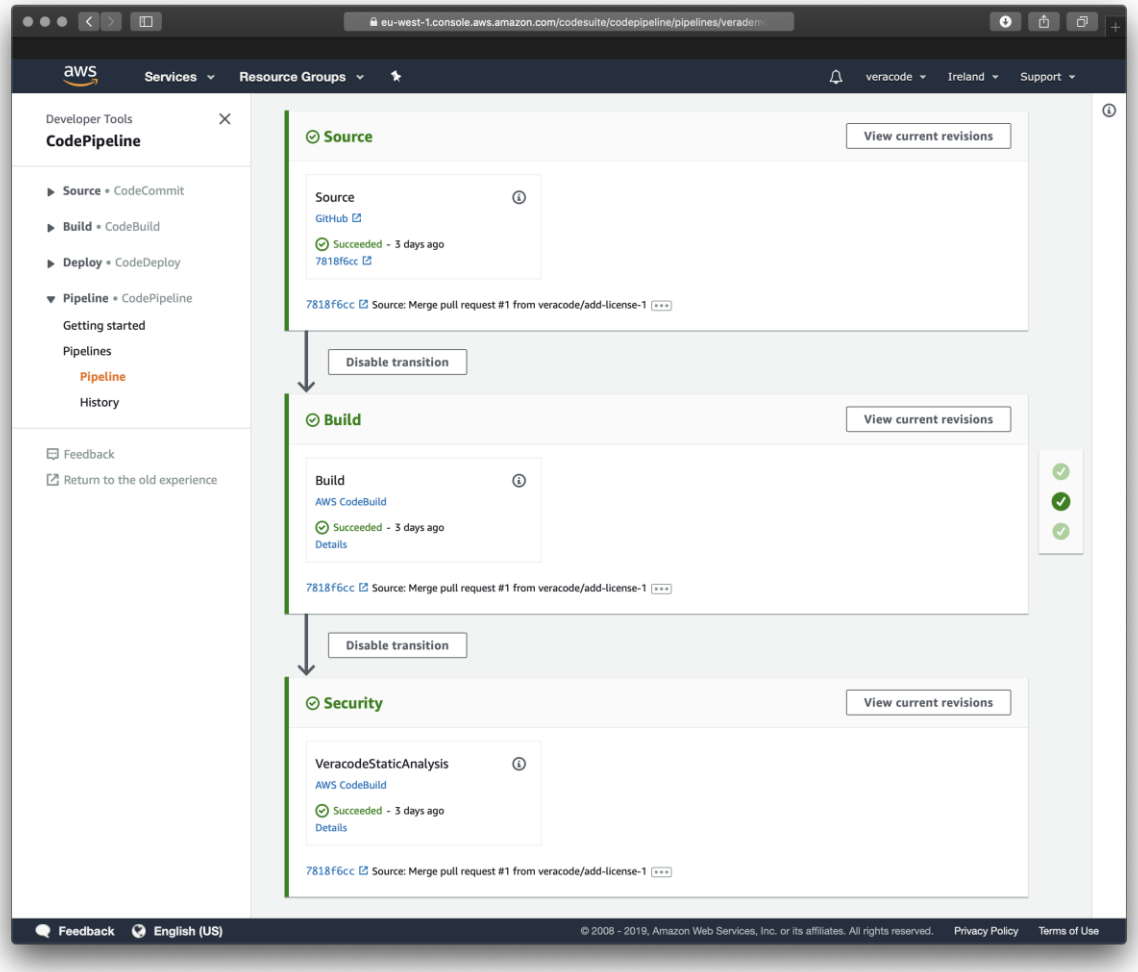

Figure 1- Final CodePipeline configuration

AWS CodePipeline Configuration

In this example the CodePipeline configuration is broken down into three stages. The first stage pulls source code from a GitHub repository for use within the pipeline. The is very little configuration work in this stage other than the URL of the repository and the branch that should be pulled.

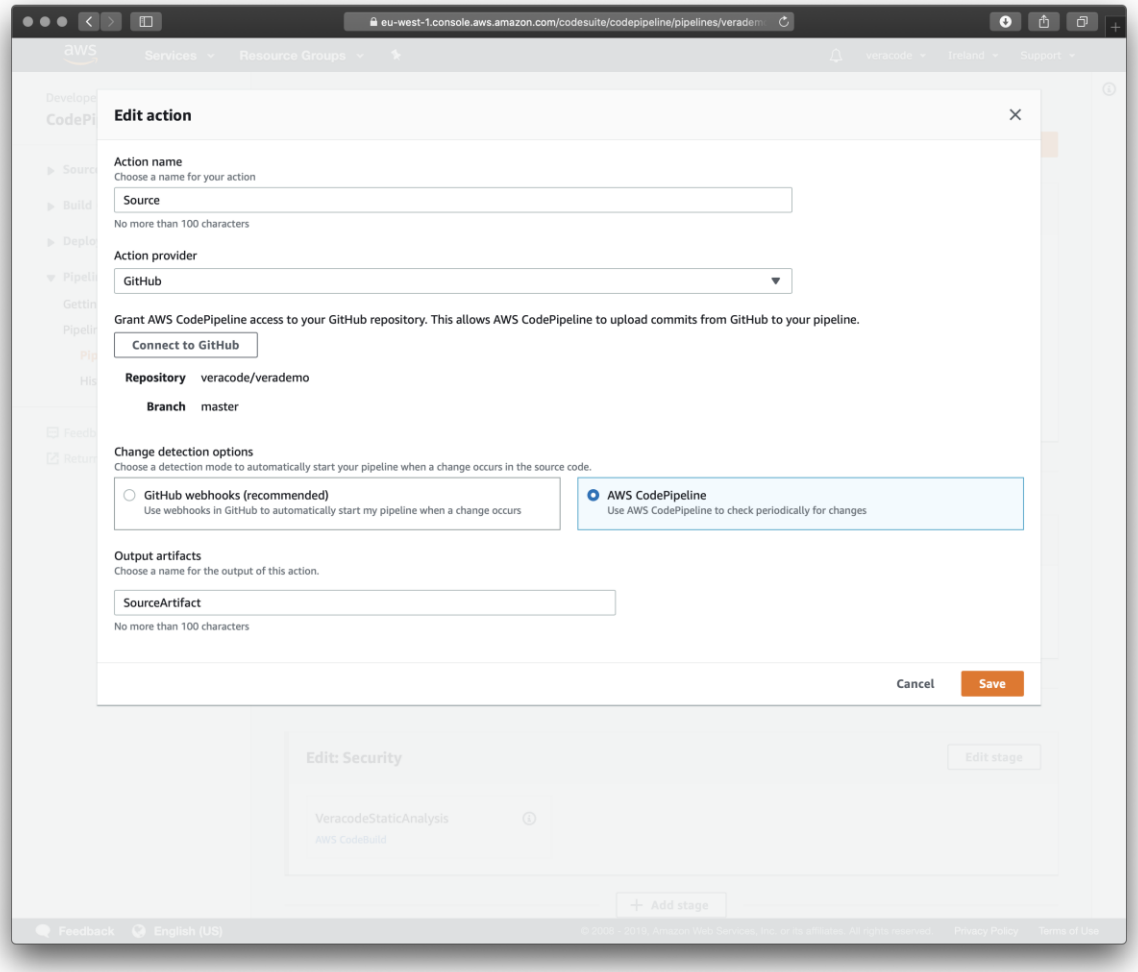

Figure 2 - Source code repository configuration

The second stage of the pipeline performs the application build. This will vary according to the specific needs of the application architecture and technology stack. In the example shown a Java web application, based on the Spring

framework, is built using a CodeBuild project that runs a Maven 'package' goal. The output of this build process is a deployable Java .war file that can be hosted on Apache Tomcat. This file is configured as an artifact to be stored in S3 and provided to later stages of the pipeline. In the CodePipeline only the name by which the artifact is referenced is configured, in this case 'BuildArtifact'. The specific file that this name reference points to is configured within the CodeBuild project shown later in this document.

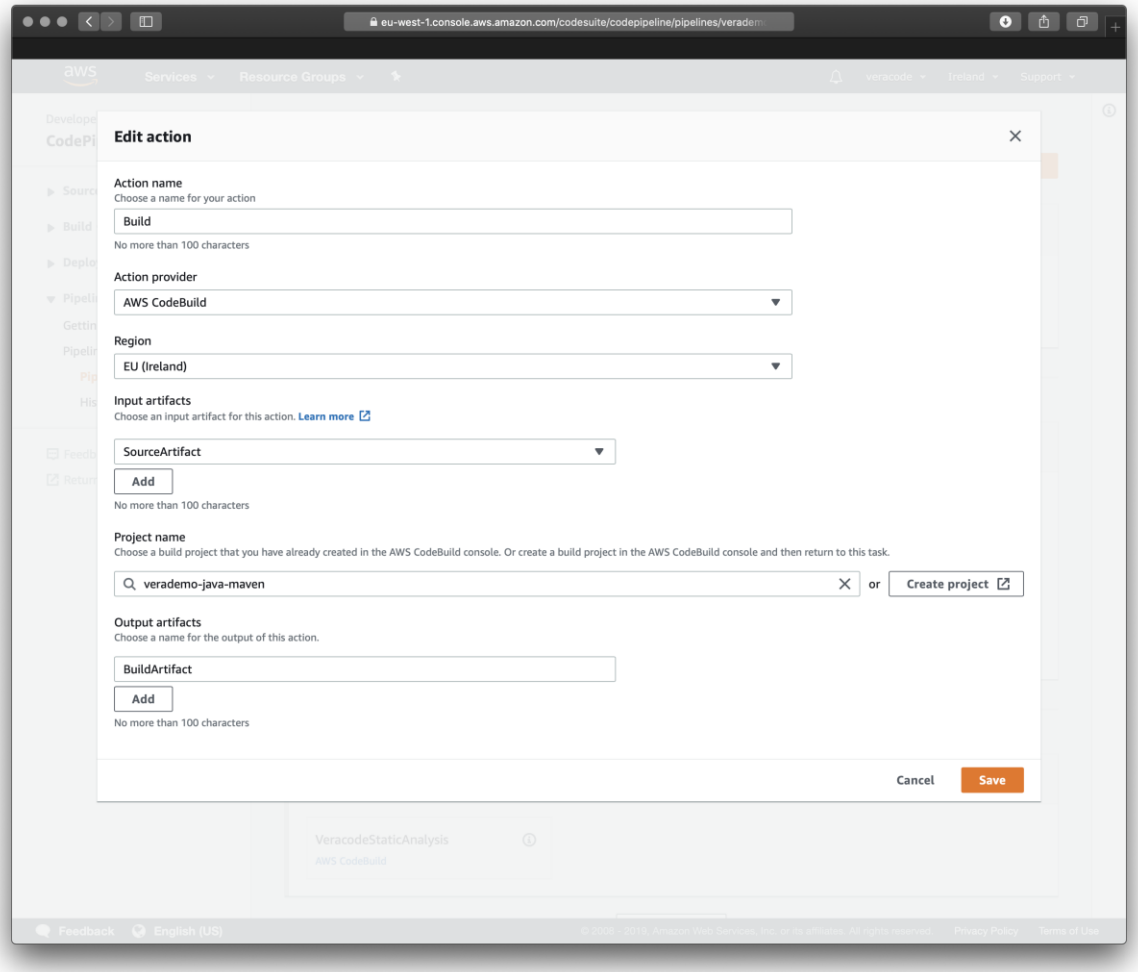

Figure 3 - Application build stage configuration

The third stage of the pipeline is where the Veracode Static Analysis is performed, via the use of a second CodeBuild project, this time using a Docker container that provides the API wrapper. In this stage the input artifact is the

output artifact from the second stage. There is no output artifact for this stage as there are no files created during the Static Analysis upload process.

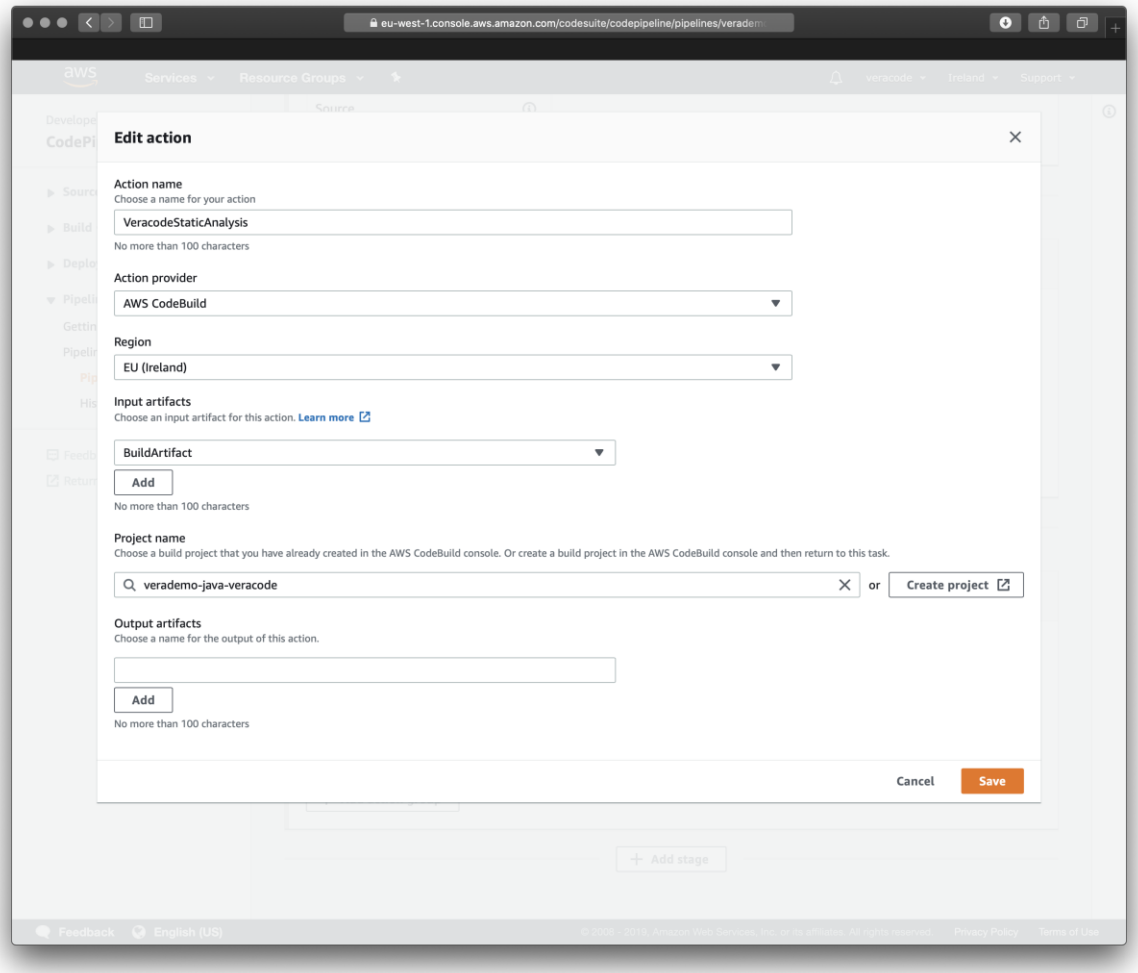

Figure 4 - Veracode Static Analysis configuration

AWS CodeBuild Configuration

This example contains two CodeBuild projects, the first project compiles the application using a Docker container that provides Maven. The specifics of this CodeBuild configuration are out of scope for this document, with the exception of the artifact configuration as shown in figure 5. The build output is targeted via the use of a 'files' node in the buildspec YAML. This is the file that will be referenced by the 'BuildArtifact' name in the CodePipeline configuration, and passed to stage three of the pipeline.

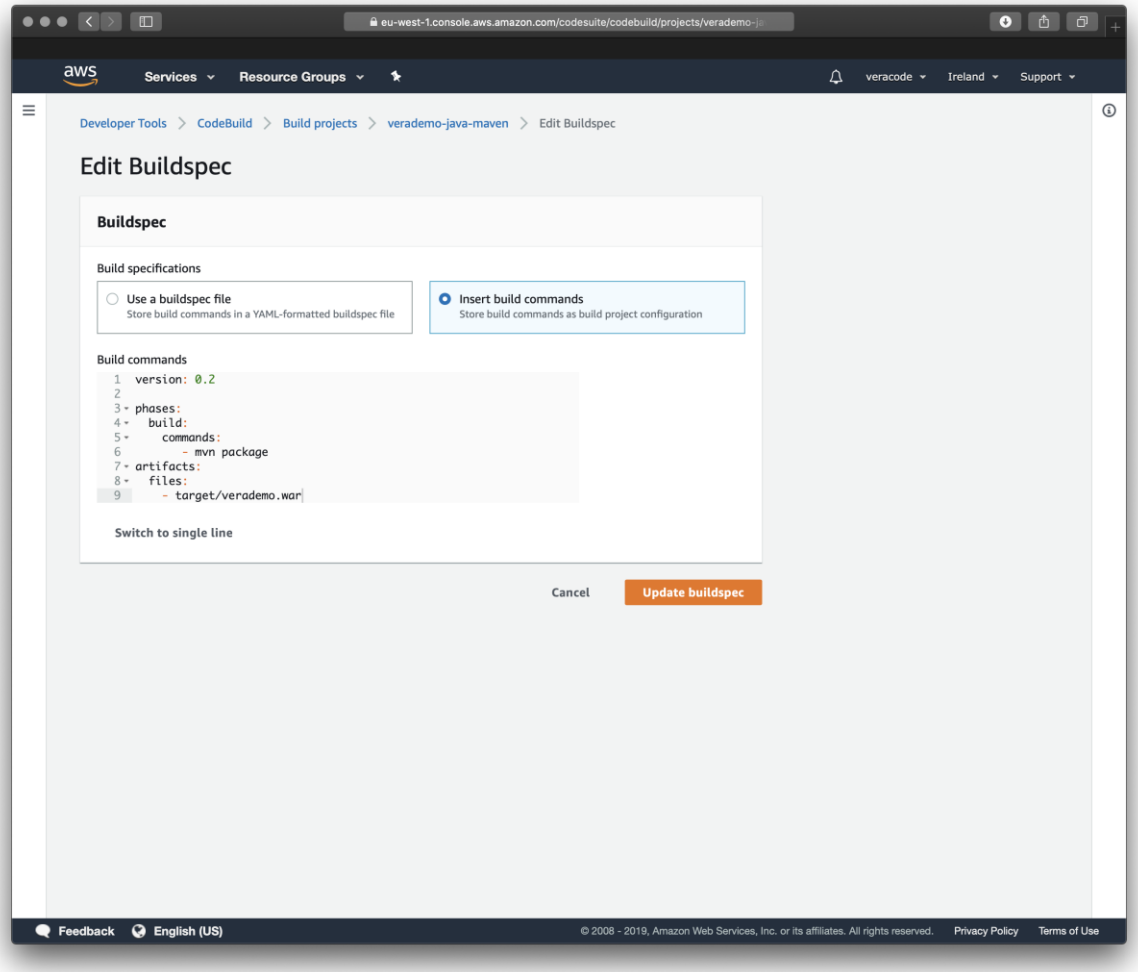

Figure 5 - Application build CodeBuild project

The second CodeBuild project runs a Docker container that has the API wrapper pre-installed. In this example the image used for the container is [ctcampbellcom/veracode-tools](). This image is provided as an example only and should not be relied upon for production usage. The Dockerfile within the GitHub repository can easily be copied and a built Docker image hosted on Docker Hub or an internally hosted registry.

The project name should clearly state its purpose for easier future usage.

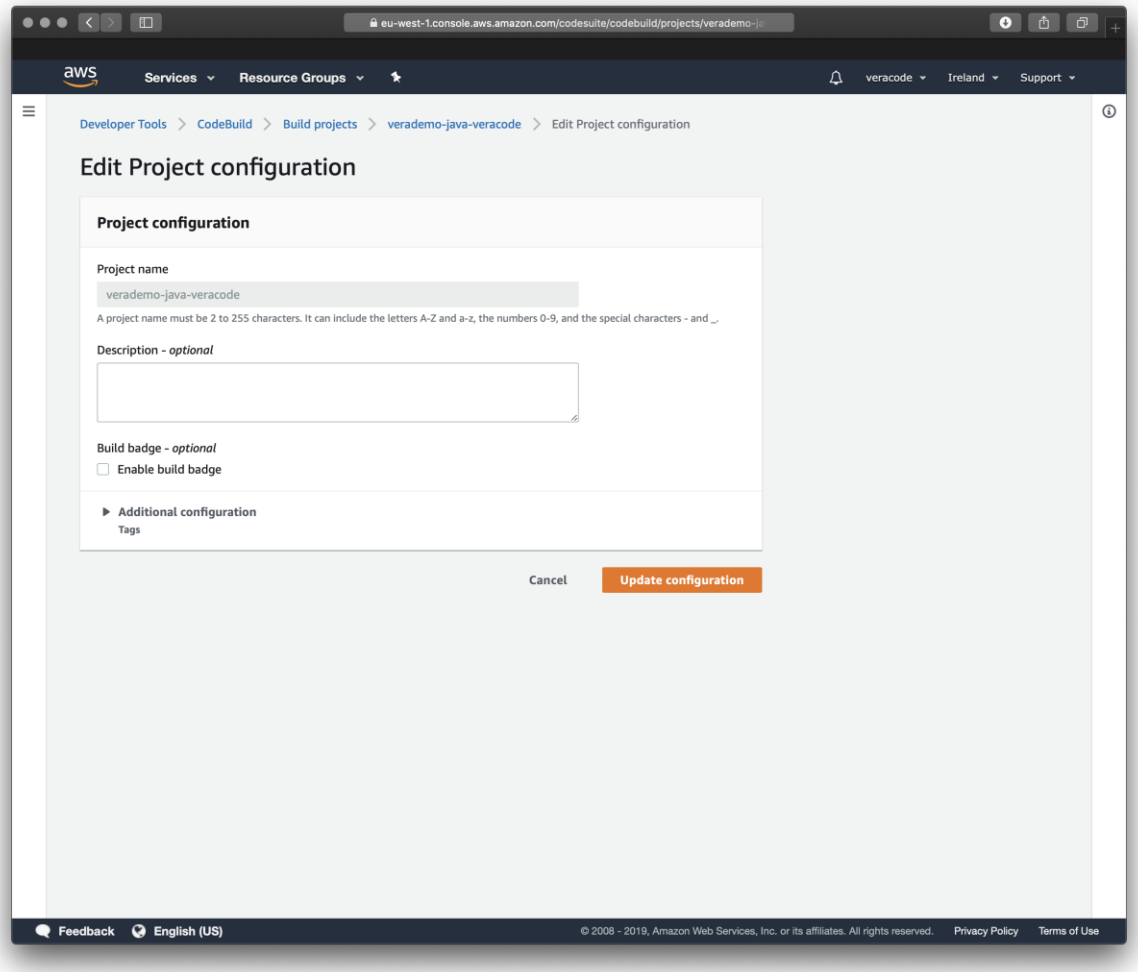

Figure 6 - CodeBuild project name configuration

The project source provider should be set to AWS CodePipeline. This will be set automatically if you have created the CodeBuild project using the 'new project' workflow whilst configuring the CodePipeline stages.

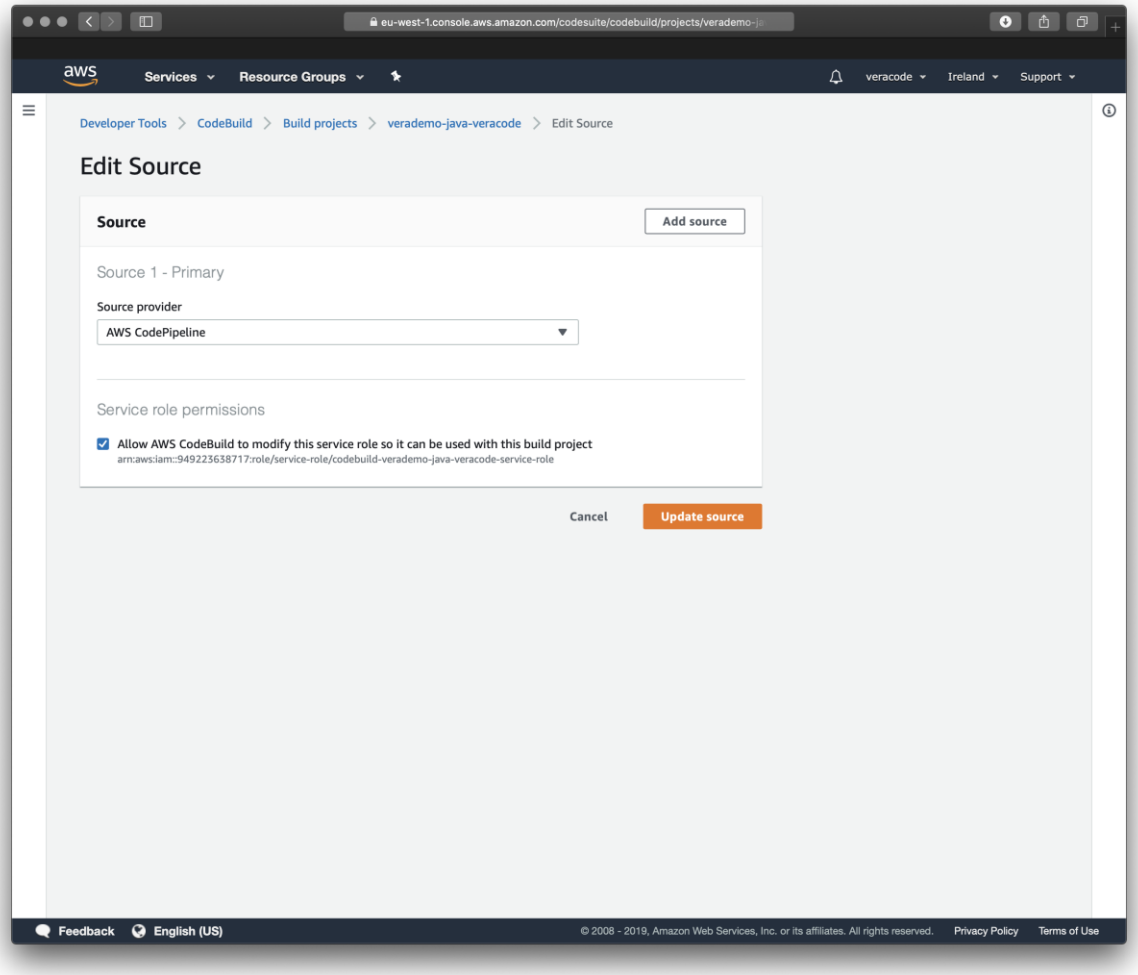

Figure 7 - CodeBuild project source configuration

The project environment configures two important parts of the overall project. The first is the Docker image as described earlier. This can be an image hosted within AWS via ECS or on Docker Hub.

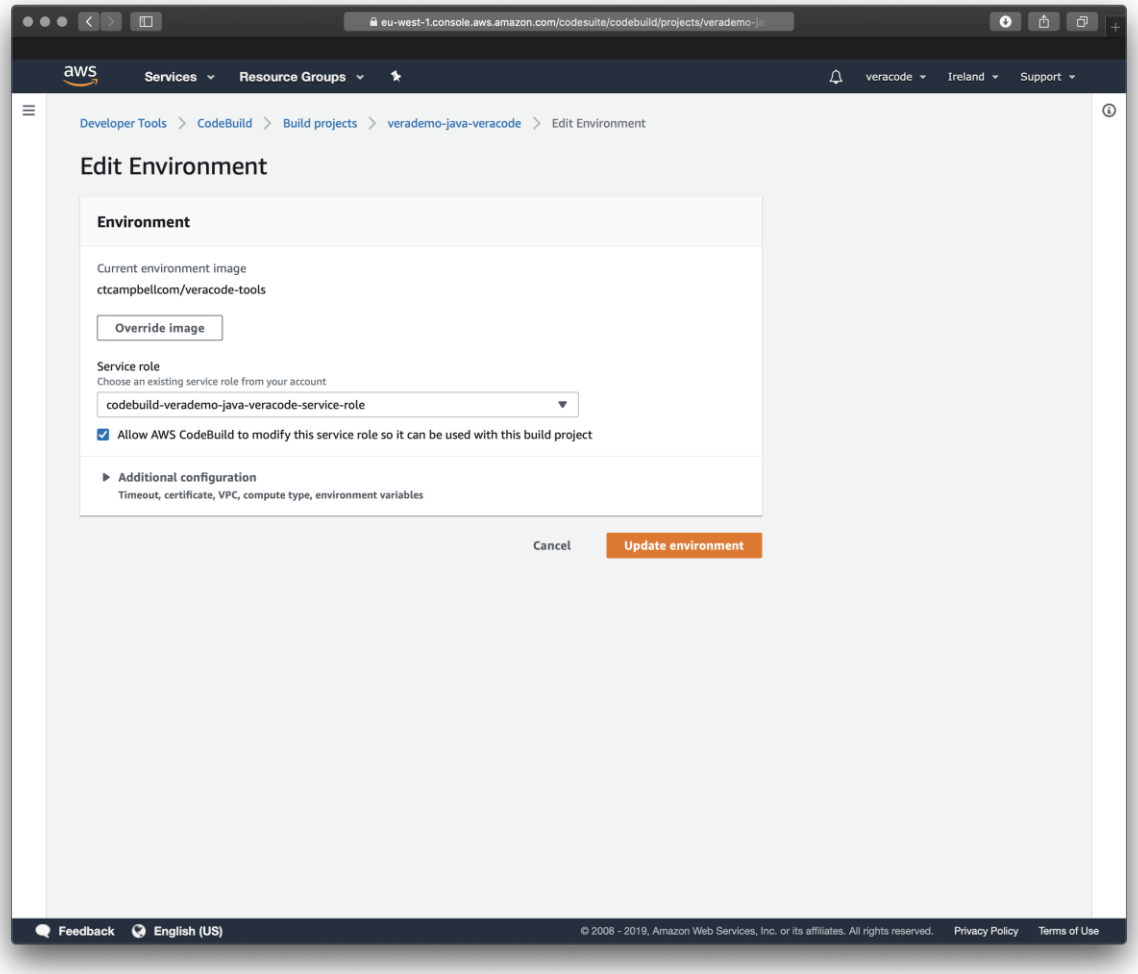

Figure 8 - CodeBuild project Docker image configuration

Expanding out the 'Additional configuration' section of the environment settings allows configuration of the environment variables that will be passed into the running Docker container when the CodeBuild project is run by CodePipeline. This is the recommended place to configure Veracode API credentials as this allows the use of the AWS Parameter Store, a secure key/value storage system. The variable names shown in figure 9, VID and VKEY are referenced later on in the project 'buildspec' configuration.

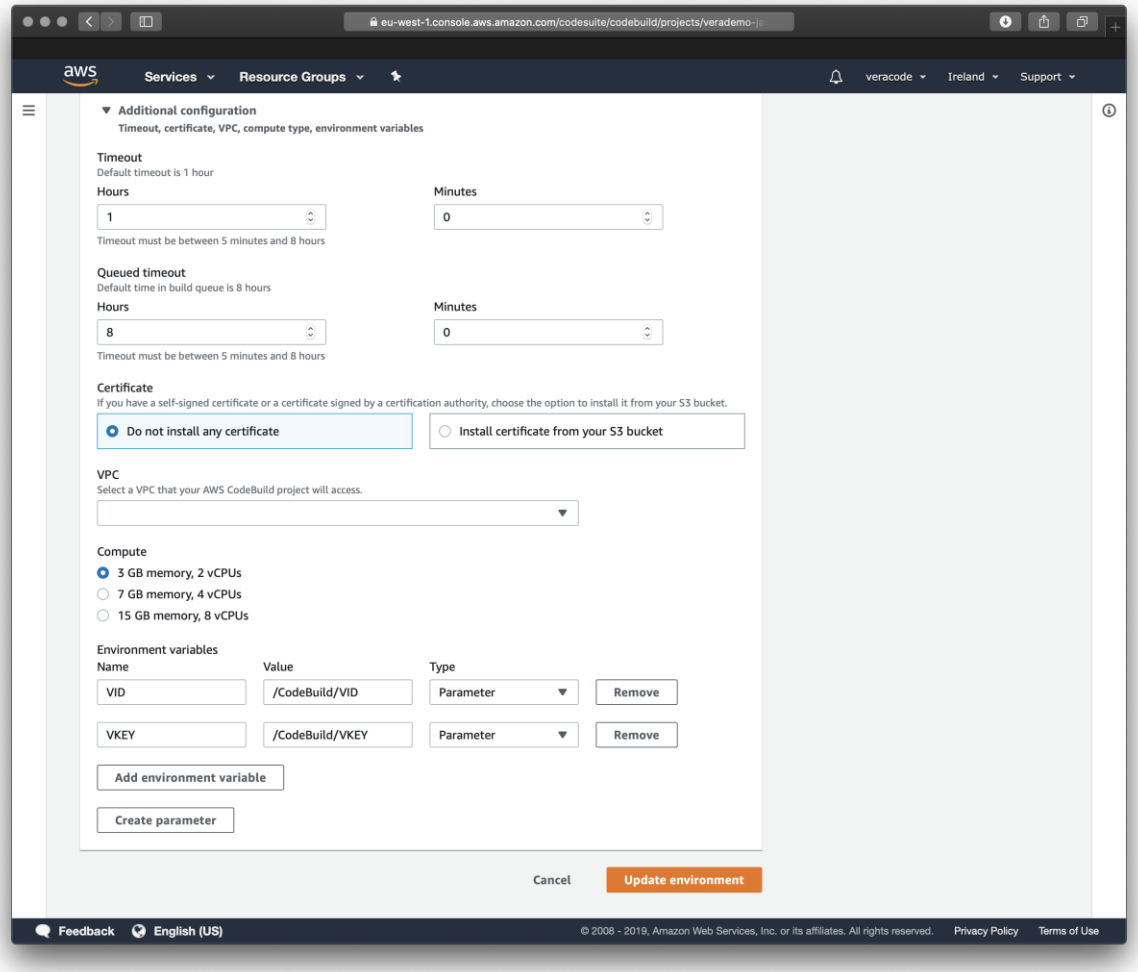

Figure 9 - CodeBuild project environment variable configuration

The project buildspec defines the commands that will be run within the Docker container. The container has both the configured environment variables and the CodePipeline build stage output artifact injected into it. This makes the process of configuring a Veracode Static Analysis scan as simple as calling the pre-installed API wrapper using the command example shown in the introduction section of this document. Details of available options for the API wrapper can be found on the Veracode Help Center.

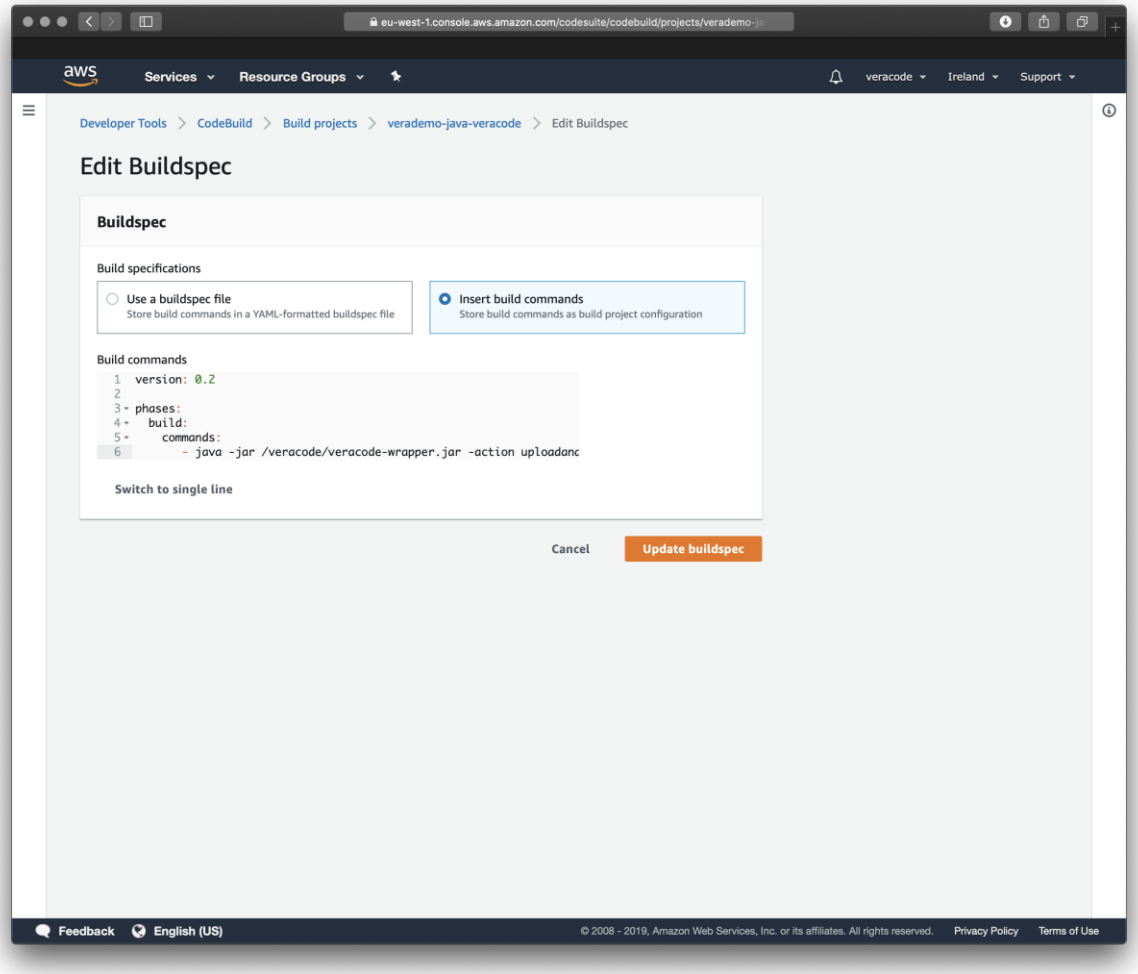

Figure 10 - CodeBuild project buildspec configuration

For troubleshooting purposes, it is necessary to configure the CodeBuild project to retain logs. This will make the output of the API wrapper visible in the history of the project. These logs can be stored in AWS CloudWatch or S3.

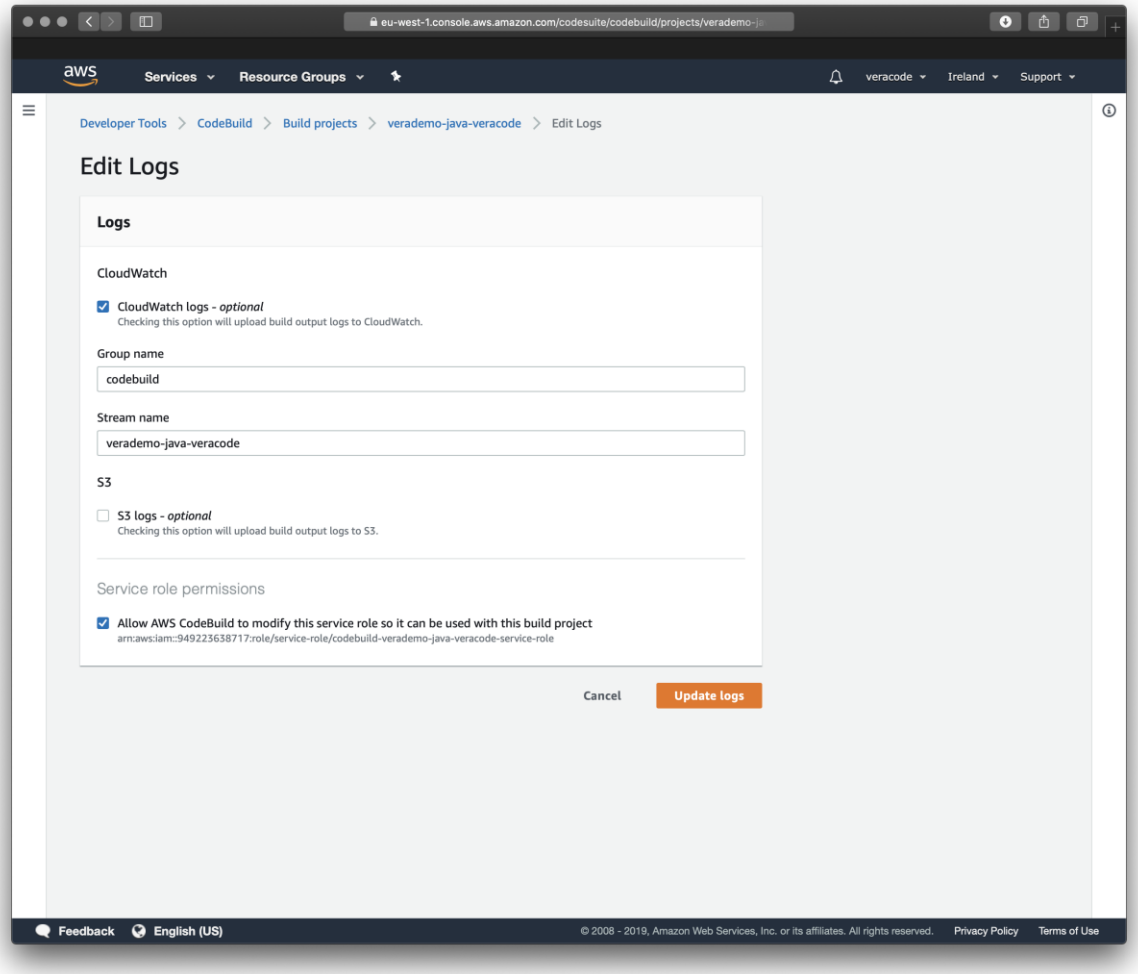

Figure 11 - CodeBuild logging configuration

Once fully configured the entire CodePipeline can be run by using the 'Release change' workflow and output similar to figure 12 should be shown.

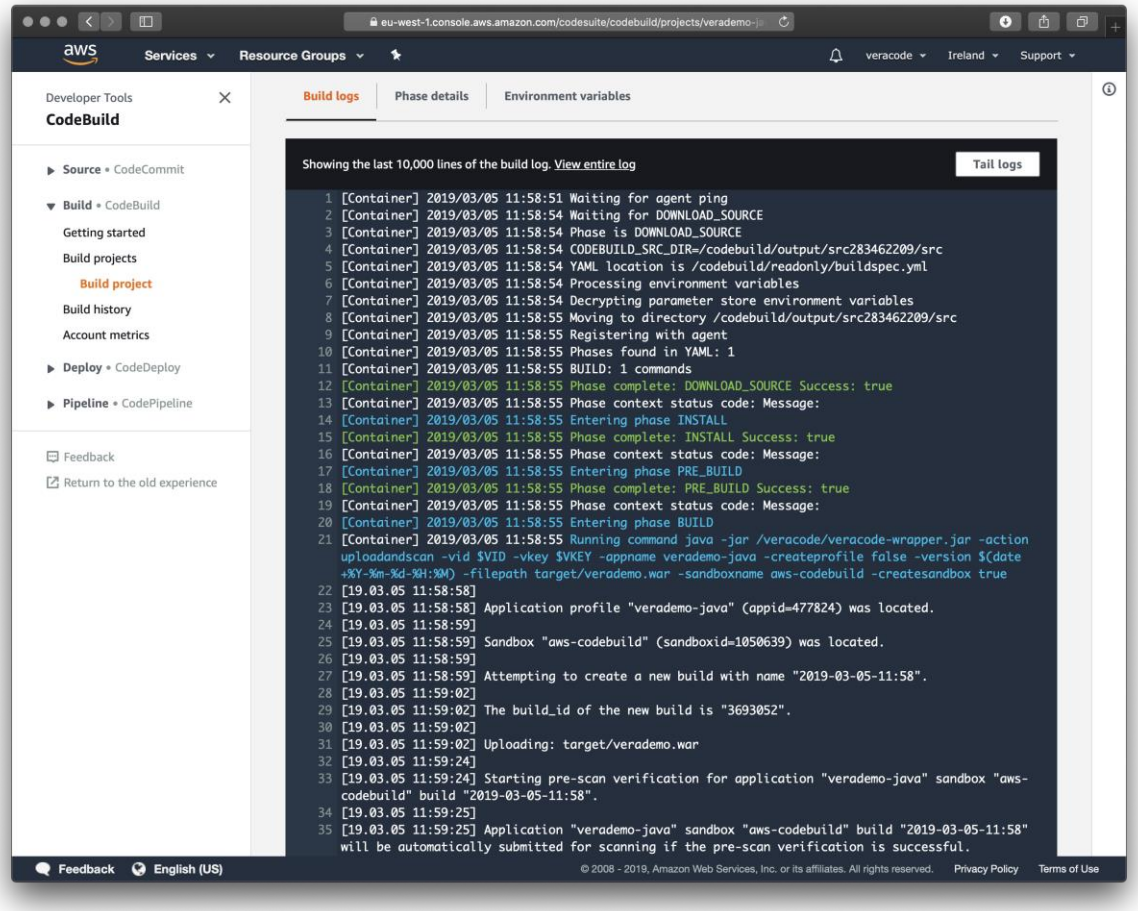

Figure 12 - Output of successful CodePipeline execution