STACS SOLUTIONS ENGINEERING

Optimizing AWS CodePipeline For AWS ECS Fargate to enable Continuous Delivery

STACS
Blockchain for Finance

PREFACE

As a fintech startup, we need to reliably deliver new product features and bug fixes to our clients rapidly and in a scalable manner while fully maximizing our technical talents and resources within the Hashstacs Solutions Engineering team.

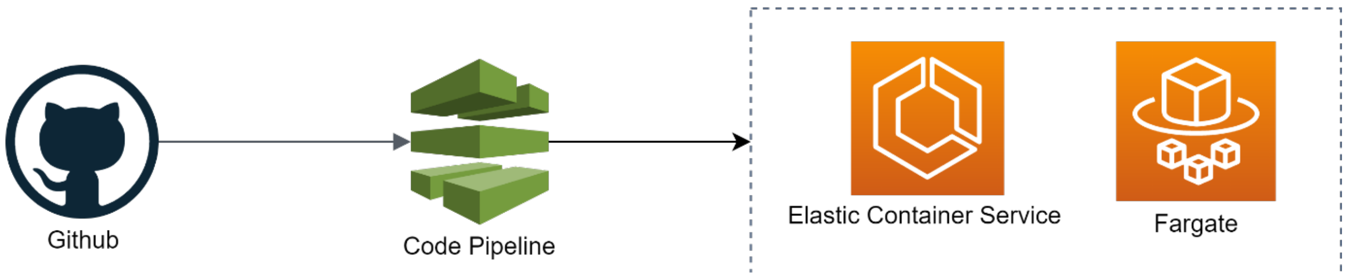
While we already have Continuous Integration in place to ensure we spend as little time as possible on fixing code conflicts, we go one step further to implement Continuous Delivery as it allows us to achieve a low Turn-Around-Time (TAT) for responding to bug fixes and software patches. To that end, we have automated our entire deployment process that completes the full software cycle from source code to full deployment on multiple environments.

CONTENTS

1 Introduction

Leveraging on the AWS Cloud and its many resources, AWS CodePipeline is essential to enable us to achieve a fully automated CI/CD process. This in turn allows our technical team to dedicate their time and effort to the software engineering aspects rather than devops or deployment especially since AWS CodePipeline is a 1-off setup (and less when we start exploring Cloud Formation with CodePipeline) for all project implementations.

The final deployment for our applications is as containerized applications on AWS Elastic Container Service (ECS) which is a fully managed container orchestration service that has enabled us to scale with performance.
We also decided to use AWS Fargate to manage the containers on ECS for us since Fargate is a managed serverless compute engine for containers to further reduce dedicated manpower costs for managing deployment resources and costs.



To track the many versions of our containerized application we use AWS Elastic Container Registry (ECR) to store and manage our containers.

In this article we would like to share our experience on using Code Pipeline and how we were able to cut our initial deployment time on our first Code Pipeline end to end model from 45 minutes down to about 5 minutes, a 900% increase in deployment speed.

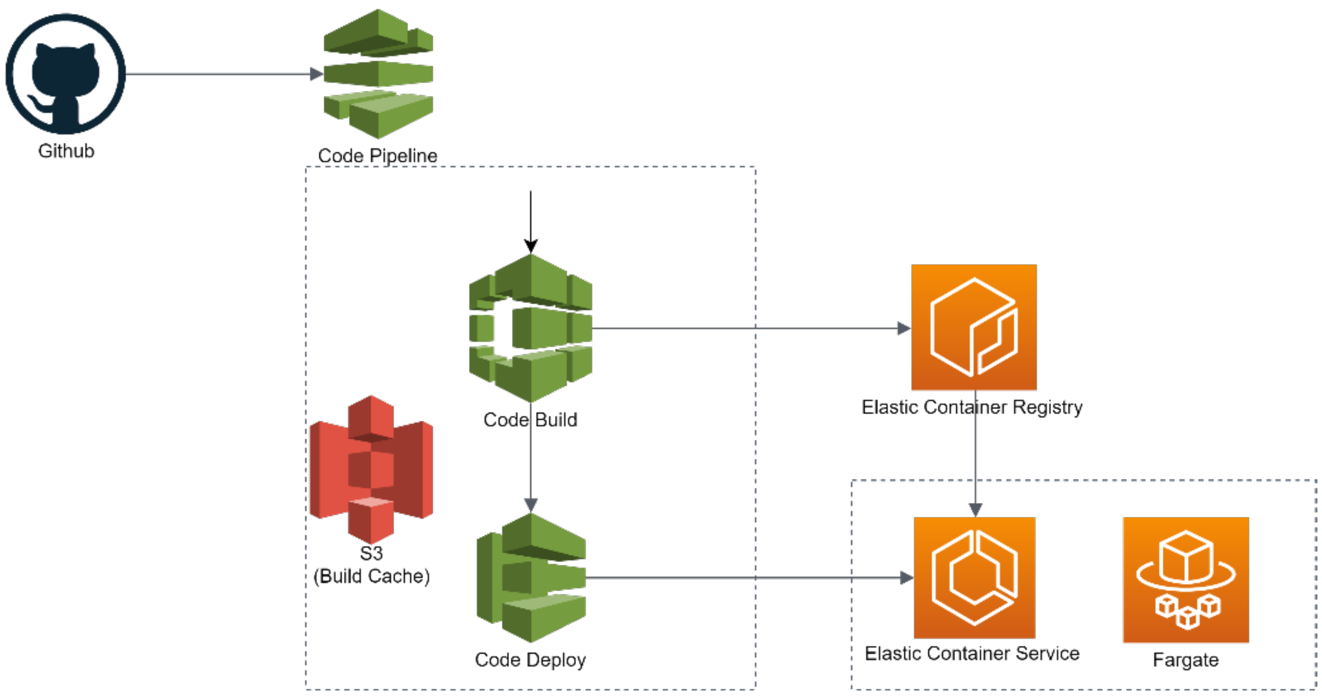
We will be sharing our experience of Deployment with ECS and Fargate as well in upcoming articles.

2 Initial CI/CD Architecture

Our application is built in Java which we must package into a Jar file for final deployment.

Since we are using containers, the Jar file is packaged into a Docker image and stored in ECR where ECS would then pull and deploy on Fargate.

The initial CI/CD Pipeline we had designed had the following architecture:

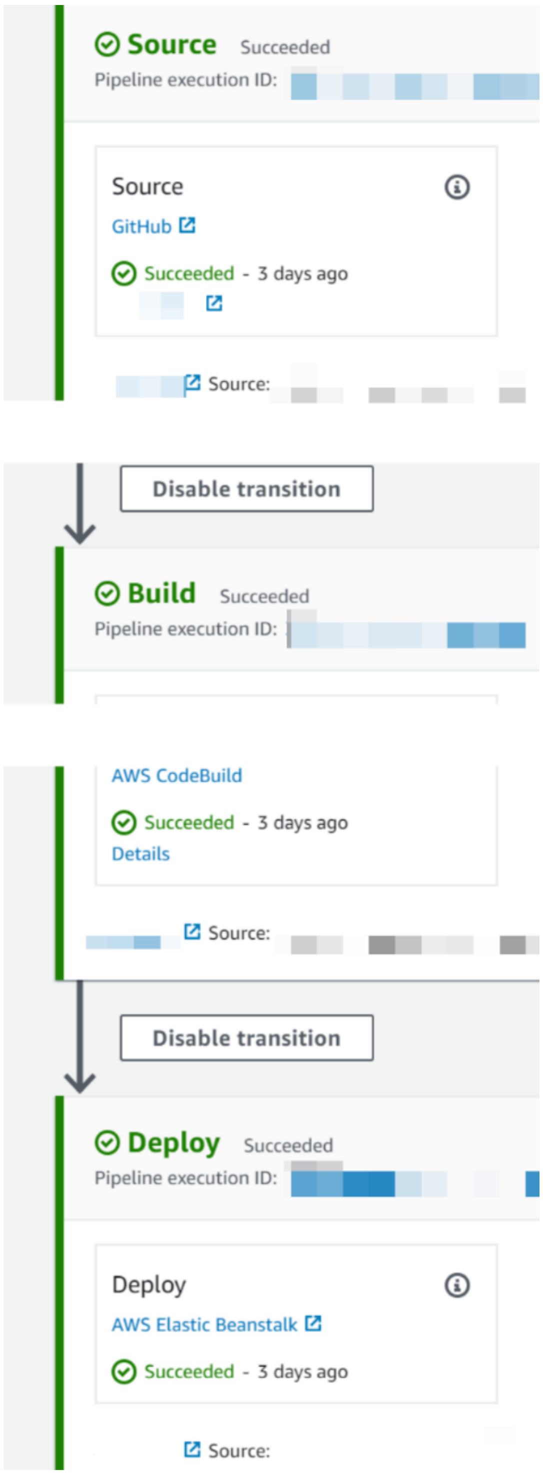


Code committed to Github is pulled by AWS CodePipeline via hooks.

This in turn triggers CodeBuild to build the source code into the final Docker Image stored in ECR. Once ready, Code Deploy deploys our application service to ECS on Fargate.

The initial build optimization made use of S3 as a build cache which we setup following instructions from here.

Thus, the initial CodePipeline had 3 stages: Source, Build and Deploy as shown in this screenshot:



Source Succeeded
Pipeline execution ID: [redacted]

Source
GitHub
Succeeded - 3 days ago
[redacted]

[redacted] Source: [redacted]

Disable transition

Build Succeeded
Pipeline execution ID: [redacted]

AWS CodeBuild
Succeeded - 3 days ago
Details

[redacted] Source: [redacted]

Disable transition

Deploy Succeeded
Pipeline execution ID: [redacted]

Deploy
AWS Elastic Beanstalk
Succeeded - 3 days ago

Source:

The Source stage simply extracts the source code from Github (we have not explored AWS CodeCommit) for the next step, Build.

2.1 Build Stage

For the Build stage, we had to factor in the many different steps that lead from the initial source code to the final output artifact, which is the Docker image to be uploaded to ECR.

In the Dockerfile, we ran Maven build commands that created the final jar file and the execution of the jar file.

```
FROM maven:3.5-jdk-8-alpine as build
WORKDIR /app
COPY . /app
RUN mvn install

FROM openjdk:8-jre-alpine
WORKDIR /app
COPY --from=build /app/target/app-1.0.0-SNAPSHOT.jar /app
ENTRYPOINT ["sh","-c"]
CMD ["java -jar app-1.0.0-SNAPSHOT.jar"]
```

For the buildspec.yml file, we followed the sample code provided by AWS here, where we built the Docker image using the Dockerfile above before sending it to ECR.

```
version: 0.2
phases:
  pre_build:
    commands:
      - echo Logging in to Amazon ECR...
      - $(aws ecr get-login --no-include-email --region
$AWS_DEFAULT_REGION)
  build:
    commands:
      - echo Build started on `date`
      - echo Building the Docker image...
      - docker build -t $IMAGE_REPO_NAME:$IMAGE_TAG .
      - docker tag $IMAGE_REPO_NAME:$IMAGE_TAG
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/$IMAGE_REPO_NAME:
$IMAGE_TAG
  post_build:
    commands:
      - echo Build completed on `date`
      - echo Pushing the Docker image...
      - docker push
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/$IMAGE_REPO_NAME:
$IMAGE_TAG
```

2.2 Deployment Speed

This first CodeBuild setup took 45 minutes for every code push despite setting up the build cache in Code Build.

We had initially imagined that a single Build stage would be sufficient for our use case, but the performance was disappointing.

The next alternative we thought up was to setup a slightly more complex, but more decoupled CodePipeline process.

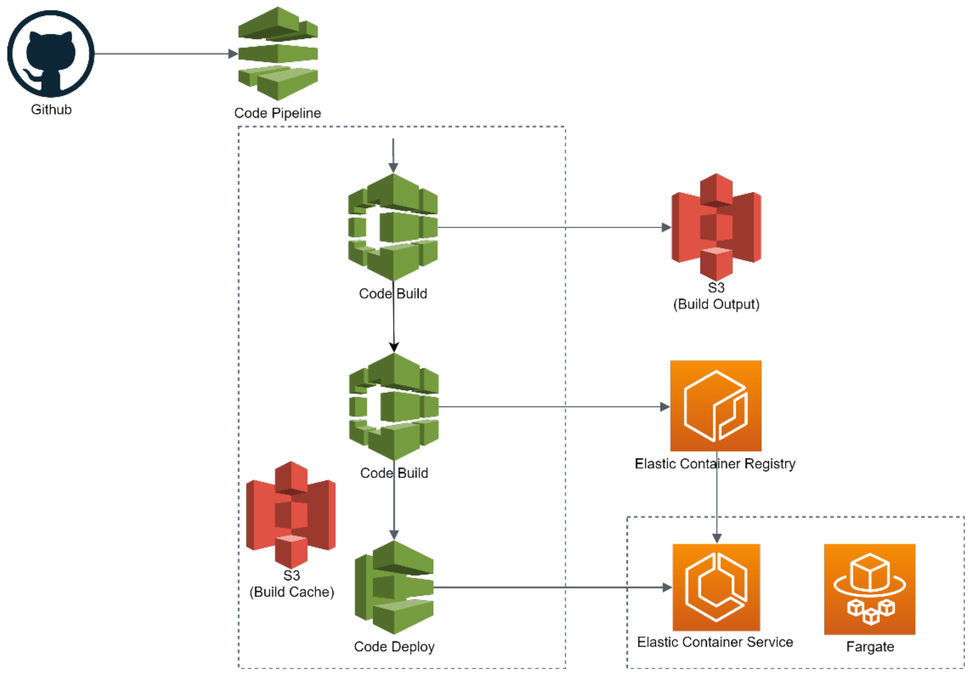
Breaking down the most complex portion of our initial design of the Build stage, we realized that we could effectively split it into 2 Stages and test for its effectiveness:

1. Packaging of the final Jar file
2. Creating the final Docker image from the previous Jar file

We set about in the next section setting up this second CodePipeline workflow.

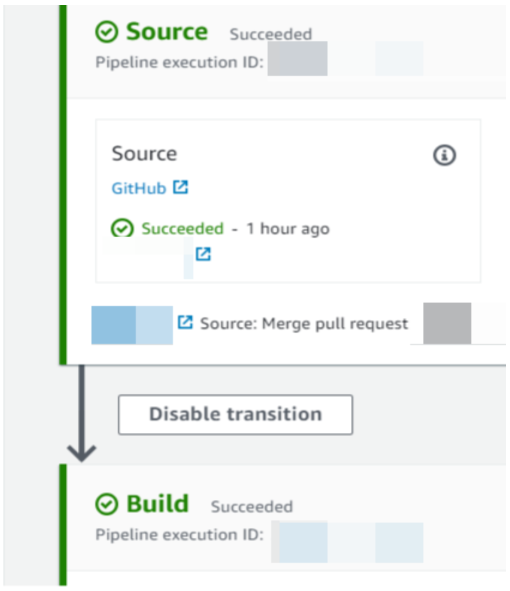
3 Improved CI/CD Architecture

This second design had the following modified architecture:



We added an additional CodeBuild Stage with a corresponding S3 bucket for storage of its build output artifacts which handled the first task of Jar creation while the second CodeBuild Stage handled the task of creating the Docker Image.

This second CodePipeline setup looks like the following with 4 stages now: Source, Build, DockerBuild and Deploy:



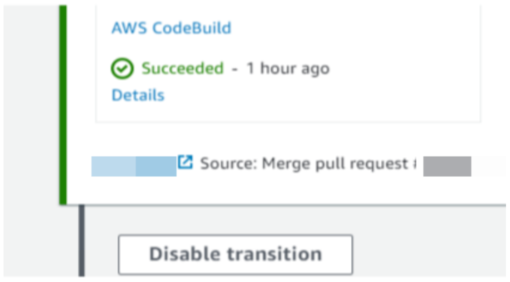
Source Succeeded
Pipeline execution ID: [redacted]

Source
GitHub
Succeeded - 1 hour ago

[redacted] Source: Merge pull request [redacted]

Disable transition

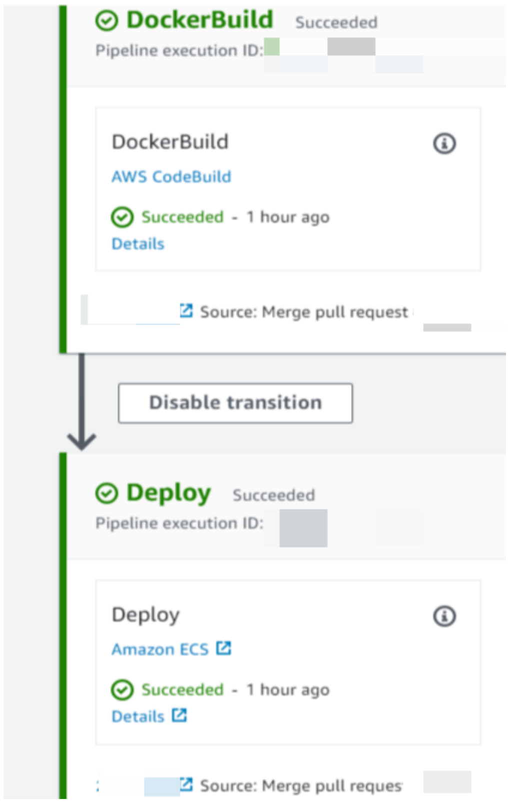
Build Succeeded
Pipeline execution ID: [redacted]



AWS CodeBuild
Succeeded - 1 hour ago
Details

[redacted] Source: Merge pull request [redacted]

Disable transition



DockerBuild Succeeded
Pipeline execution ID: [redacted]

DockerBuild
AWS CodeBuild
Succeeded - 1 hour ago
Details

Source: Merge pull request [redacted]

Disable transition

Deploy Succeeded
Pipeline execution ID: [redacted]

Deploy
Amazon ECS
Succeeded - 1 hour ago
Details

[redacted] Source: Merge pull reques [redacted]

3.1 First Build Stage

In this first build stage, we no longer needed such a complex Dockerfile and simplified it to the following:

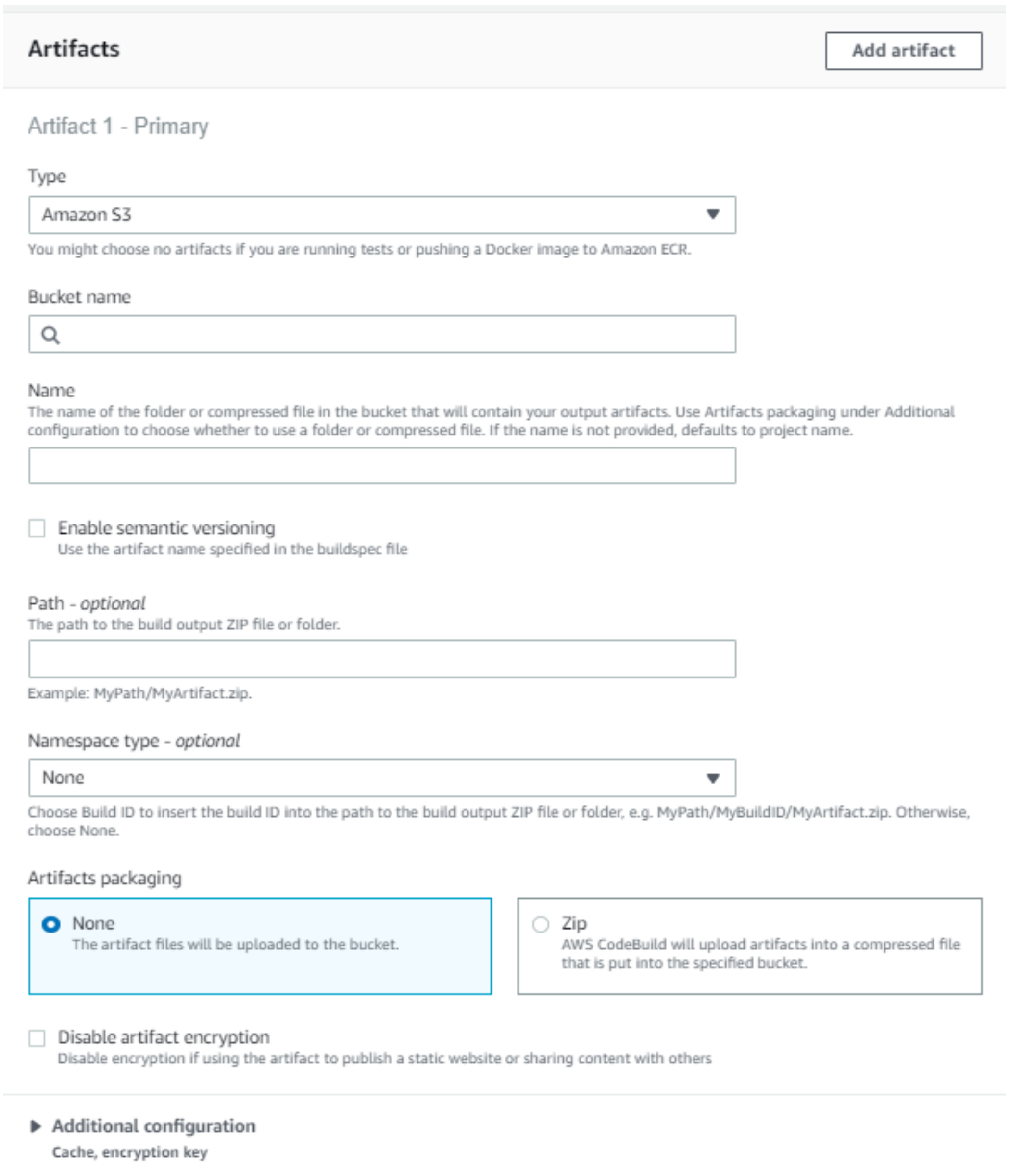
```
FROM openjdk:8-jre-alpine
WORKDIR /app
COPY app-1.0.0-SNAPSHOT.jar /app
ENTRYPOINT ["sh","-c"]
CMD ["java -jar app-1.0.0-SNAPSHOT.jar"]
```

Next, we modified the buildspec.yml file for the first Code Build stage to create a Jar package with the following settings.

```
version: 0.2
phases:
  install:
    runtime-versions:
      java: corretto8
  pre_build:
    commands:
      - echo "build the jar file"
      - mvn clean compile test -T1C
  build:
    commands:
      - mvn -T 1C package
artifacts:
  files:
    - target/*
  discard-paths: yes
cache:
  paths:
    - '/root/local/**/*'
```

Now that we have made the necessary modifications, we had to figure out where to put the output Jar file (the output artifact) of the first CodeBuild stage.

Conveniently, AWS CodeBuild had the option of using S3 to store our output artifacts.
This was made simply as we could just use the AWS Management Console UI to configure as shown in the diagram below:



Artifact 1 - Primary

Type

Amazon S3

You might choose no artifacts if you are running tests or pushing a Docker image to Amazon ECR.

Bucket name

Name

The name of the folder or compressed file in the bucket that will contain your output artifacts. Use Artifacts packaging under Additional configuration to choose whether to use a folder or compressed file. If the name is not provided, defaults to project name.

☐ Enable semantic versioning
Use the artifact name specified in the buildspec file

Path - *optional*

The path to the build output ZIP file or folder.

Example: MyPath/MyArtifact.zip.

Namespace type - *optional*

None

Choose Build ID to insert the build ID into the path to the build output ZIP file or folder, e.g. MyPath/MyBuildID/MyArtifact.zip. Otherwise, choose None.

Artifacts packaging

[✓ None] The artifact files will be uploaded to the bucket.

Zip — AWS CodeBuild will upload artifacts into a compressed file that is put into the specified bucket.

☐ Disable artifact encryption
Disable encryption if using the artifact to publish a static website or sharing content with others

The first CodeBuild stage is now fully configured.

3.2 Second Build Stage

As for the second CodeBuild stage that creates the final Docker image from the previous Jar file, we had to add a second buildspec.yml file.

This second "buildspec-second.yml" would handle the docker image creation and push to ECR with the same code as what we had in the first CodePipeline version.
For this setup, we had to setup the ECR repository first and retrieve the following information from the AWS Management Console:

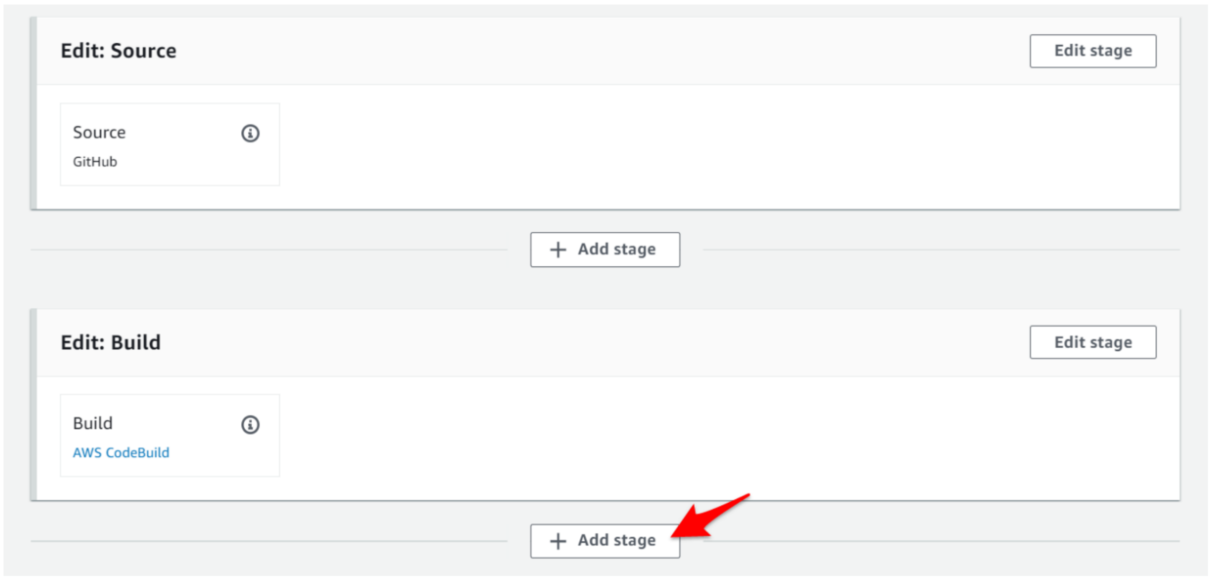
1. the ECR Repository Name
2. the ECR Repository URI
3. ECR Image Tag

```
version: 0.2
phases:
  pre_build:
    commands:
      - echo Logging in to Amazon ECR...
      - $(aws ecr get-login --no-include-email --region
$AWS_DEFAULT_REGION)
  build:
    commands:
      - echo Build started on `date`
      - echo Building the Docker image...
      - docker build -t $IMAGE_REPO_NAME:$IMAGE_TAG .
      - docker tag $IMAGE_REPO_NAME:$IMAGE_TAG
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/$IMAGE_REPO_NAME:
$IMAGE_TAG
  post_build:
    commands:
      - echo Build completed on `date`
      - echo Pushing the Docker image...
      - docker push
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/$IMAGE_REPO_NAME:
$IMAGE_TAG
      - echo Writing image definitions file...
      - printf '[{"name":"<ECR Repository Name>","imageUri":"<ECR
Repository URI>:<ECR Image Tag>"}]' > imagedefinitions.json
artifacts:
  files: imagedefinitions.json
```

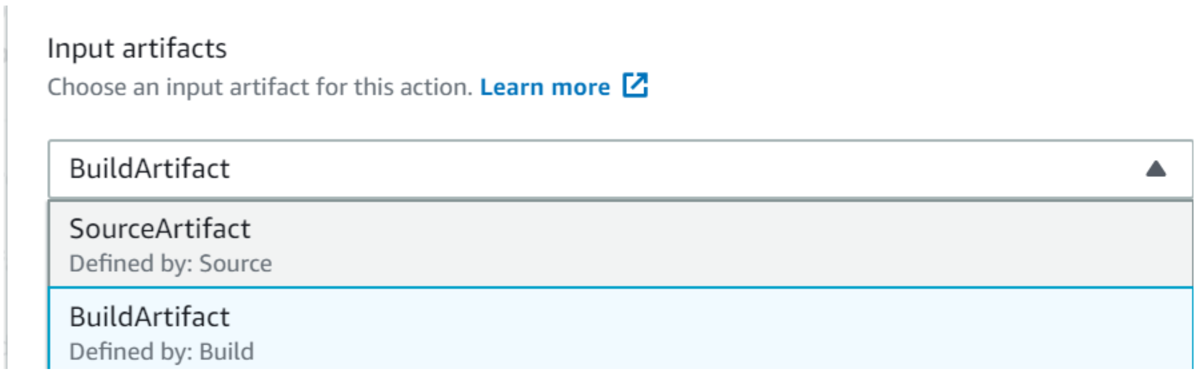
The setup for the second Build stage is now complete.

3.3 Wiring both Build Stages in CodePipeline

On the CodePipeline end, it was simple to just add an extra stage between Build and Deploy as shown:



In the second Build stage, simply define the Input Artifact as the First Build Stage's output artifact, as shown in the screenshot below (Defined by shows the output artifacts of each stage, making it easy to identify which should be the input artifact):


Input artifacts
Choose an input artifact for this action. Learn more

BuildArtifact

SourceArtifact
Defined by: Source

BuildArtifact
Defined by: Build

4 Findings

With this second CodePipeline setup, we were able to cut down the deployment time from Source to Deploy down to about 5 minutes which was great.

CodePipeline was easy to implement especially since all the configuration could be done directly on the AWS Management Console which made it easy to visualize our entire deployment process.

Moving forward, we would love to explore how to automate the CodePipeline setup for new project implementations with AWS Cloud Formation.

While we are happy with this first step of improving our deployment processes, we are excited to explore and learn more ways to increase our deployment speed and be able to deliver rapid code changes to our clients.