CloudFirst: A Chatbot & Dashboard For Modern Deployment Pipelines

Final Year Project Report

TU857
BSc in Computer Science (Infrastructure)

Joshua Rogan

C17488402

Supervisor: Ciaran Kelly

School of Computer Science

Technological University, Dublin

06/04/2021

Abstract

The goal of this project is to improve a software development organizations continuous integration and deployment pipeline as the organization grows and adds more developers and product features, while maintaining the quality and speed of their production builds & releases.

CloudFirst will be a solution that helps developers bring their code changes through build pipelines with confidence.

Developers will be able to monitor the pipeline status via the company chat application(e.g. Slack) and web interface. They will also be able to interact with the bot/interface to view the status of builds, debug them, restart pipelines or their various stages.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

_____

Joshua Rogan

Date 06/04/2021

Acknowledgements

Thanks to Ciaran Kelly for being my Supervisor.

Thanks to my Mother and Father Philomena and Peter for their encouragement and support throughout my final year.

Table of Contents

Table of Figures

1. Introduction

1.1 Project Background

As part of my internship I worked as a software engineer in a large organization that had a lot of developers worldwide contributing code daily.

One of the major issues that slowed down shipping features was the merge queue and build pipeline, it was very manual process that required a lot of interaction from the developer submitting code changes, this slowed down the pipeline massively since it isn't possible for an individual developer to be paying attention to a merge queue and build pipeline 24/7.

While researching potential solutions to this during my internship, I found out this was a common problem industry wide regarding software delivery.

The first thing I found was a Shopify blog[1] talking about how they handle 1000+ developers all merging into a single repo, they go into how they've tried to automate as much of the process and the trade-offs they made building their CI/CD system.

In Keeping Master Green At Scale[2] discusses using advanced ML techniques to predict which builds will be successful. It showed me that there's multiple successful approaches for the similar topic of: keeping master "green" and speeding up the CI/CD of their deployment processes.

Smaller companies or development teams also have issues with manual build pipeline processes, Strava[3] pointed out in their blog how they improved their CI/CD processes with a simpler approach of using a slack chat bot that developers interact with by typing commands to.

Agoda[4], a hotel booking site headquartered in Singapore had a similar approach to Strava. They built a Release Manager bot which also utilized Slack to stay on top of their release pipeline, the result means they have automated their previous manual release process meaning developers are freed up to work on more important things.

1.2 Project Objectives

The main objective of this project is to develop a system that helps developers monitor, and triage potential issues that occur while trying to get their code changes through a build pipeline.

The system will help developers push their changes through the deployment pipeline and if anything goes wrong they'll be able to monitor,triage and fix the issues from their teams chat client alongside other team members via a chatbot interface or via a web interface.

To obtain these objectives I'll have to implement the following:

- An interactive chat bot capable of notifying developers of any important updates to their build pipelines, as well aas being able to approve/reject builds or request more in depth information about these pipelines from the chat channel
- An interactive web interface with the same capabilities as above, alongside a birdseye view of all the pipelines in the organization and their status.
- The pipeline infrastructure surrounding this system will also have to be implemented

1.4 Project Scope

My project is focused on improving the way developers get their changes through a build pipeline and the triaging of bugs and monitoring that revolves around it. Not the pipeline itself, so I will not be implementing my own pipeline but will be leveraging existing technologies to do that.

1.3 Structure of the Document

Research

In this chapter, I conduct a literature review into the existing research in the fields of Continuous Delivery, Continuous Integration and their application in industry. I then discuss alternative existing solutions that exist in this problem space. After looking at existing solutions I then research the technologies I'll use for building this project.

Design

In this chapter I delve into my chosen methodology for this project along with the reasoning behind it. I then come up with user requirements which are used to come up with a feature list for the proposed system. After this I design the UI interface for the chatbot, dashboard, and the technical architecture for the system.

Development

In this chapter I will describe how I actually developed the project. e.g. wrote the code and any pitfalls I had along the way, and what changed from the design chapter

Testing and Evaluation

In this chapter I talk about how I tested the project and how I evaluated the system to see whether it was a success or not

Conclusion
In this chapter I talk about the conclusions I have taken from each previous chapter, how the project plan went, potential future work, and offer a final reflection on the entire experience.

2. Research

2.1. Literature Review

Introduction
A literature review was conducted into existing research in the fields of Continuous Integration, Continuous Delivery, and how these practises are adopted in industry. The common themes that emerged from this research are outlined below.

Continuous Integration
Continuous Integration(CI) is the practise of integrating code changes from multiple contributors into a single repository. The main goal of CI is that the software is in a working state all the time[5].

Another benefit of CI is that teams can break down large tasks into smaller deliverable chunks, with the benefit of getting quicker feedback from end users and feedback from the delivery pipeline to make sure it doesn't break the system at large. [10]

Every time somebody commits a change, the application is built and a comprehensive set of automated tests is run against it. If it breaks then you can focus on fixing it immediately.

Compared to projects that do not use CI, projects that do use CI:[6]

- release twice as often,
- accept pull requests faster
- have developers who are less worried about breaking the build

Continuous Delivery
Continuous Delivery(CD) is the ability to get code changes into production safely, quickly, and in a sustainable way.[3]

Without adopting CD, developers have to manually develop, test and deploy their code which reduces the time they could be spending working on more improving the product itself for the end user.

An example CD pipeline would look like this at a high level in figure 1:

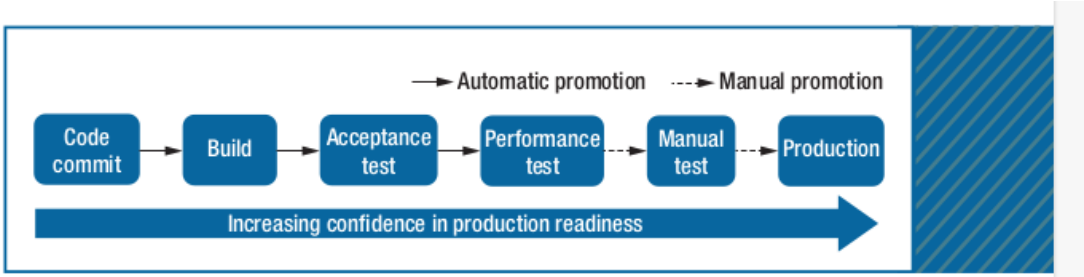

Figure 1 Example Continous Delivery Pipeline

Compared to companies that do not practise CD, companies that do are able to achieve: [7]

- Lower risk releases
- Faster time to market
- Higher software quality
- Lower costs over the product's lifetime
- Better products fuelled by quicker end user feedback
- Happier Developers[4]

Developer Experience

Developer Experience(DX) is how software developers feel about using software tools & methodologies while using them. It is the equivalent of user experience for end users.

The reason why developer experience is important to this product and any product that targets developers is that if you want people to use your product, they have to like using your product.

A Better CI & CD experience makes teams more unified, increases developer motivation and reduces their stress levels.[6,8]

Communication, Collaboration

The main component of my project is to build a centralized chatbot, that enables better communication and collaboration around build pipelines for software development teams.

Communication plays an enormous role in modern distributed development teams. Everyone associated with development should have access to a common instant messaging platform to help keep things running smoothly.[5]

In paper[9] it's noted that there are lots of studies that focus on organizations and the tooling they use to build their deployment pipeline, but less focus on technologies facilitating communication and knowledge sharing in teams.

They[9] suggest that future research should explore the possibilities of promoting communication and collaboration through tools in this area.

From the research I've conducted into communication and collaboration, it's clear that there's a gap in the ecosystem for a product that fulfills the communication and collaboration aspects of building and delivering software.

Transparency, Dashboards

In Continuous Delivery[5], they recommended that each team should have a dashboard that visualizes the status of builds that should be shared with and be visible to everyone on the team.

Team awareness and transparency was found by [10] to be the second most critical factor to adopting continuous practises behind automated testing.

Improved team awareness and transparency across the entire software development team enables its members to find potential conflicts before delivering software to customers and also improves collaboration among all teams in an organization. [10]

They also suggest that it is needed to develop innovative approaches and tools, which not only enable team members to receive build and test results correctly and timely, but also they should be aligned and integrated with deployment pipeline.[10]

Initially this product planned to emphasize the chatbot aspect of build pipelines, but now I understand the importance of dashboards to CI/CD and I plan to implement one to complement the chatbot that offers immediate feedback on the relevant build information.

Scalability & Ease of Implementation

The ability to scale and implement a reusable pipeline is essential if a software organization wishes to adopt a CI/CD mindset.[11]

it's also important that teams aren't slowed down by implementing the infrastructure hardware or software aspects of pipelines[12].

To ensure this product is both scalable and easy to implement I plan to utilize a lot of existing cloud technologies like serverless and any infrastructure that has to be created will be created using an Infrastructure as Code approach.

Conclusion

From reviewing the relevant literature, it is clear that the aspects of CI/CD that contribute to its success in an organisation are:

- Enabling Communication & Collaboration to everyone involved in the development process. This is something current research suggests is lacking [9]

- Visibility into the build process via a Dashboard available to everyone on the team.
- Scalable and easy to implement.
- Hassle free developer experience that allows teams to focus on delivery product features.

I hope to enable communication & collaboration through a chatbot interface, increase visibility with a Dashboard about the build pipeline, and make CI/CD easy to scale and implement by leveraging cloud technologies.

I will use these points to guide my technology choices in the rest of the chapter.

2.2. Alternative Existing Solutions to Your Problem

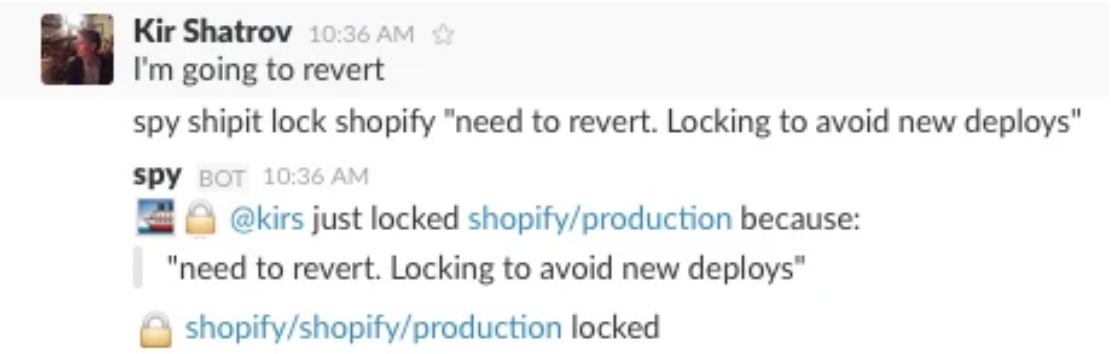

Figure 2: Shipit commands 1

In This section I look at three existing solutions that try to solve the CI/CD problem: Shipit, Octopus Deploy and Jenkins.

To make it easier to compare these existing solutions, I plan on scoring them using a heuristic evaluation that is devised from the conclusion to my literature review.

Heuristic Evaluation Framework

The areas I plan on scoring during evaluation are:

- Facilitation of Communication & Collaboration
- Visibility into the system, build pipeline & dashboards
- Scalability and ease of implementation
- Developer Experience(DX): UI, ease of use, aesthetic design.

I'll be using a 10 point scoring scale for grading:

- 1-3: Does not adhere to the heuristic at all
- 4-6: Provides a decent implementation of the heuristic
- 7-10: Respects the heuristic almost perfectly with only minor or no problems

Shipit

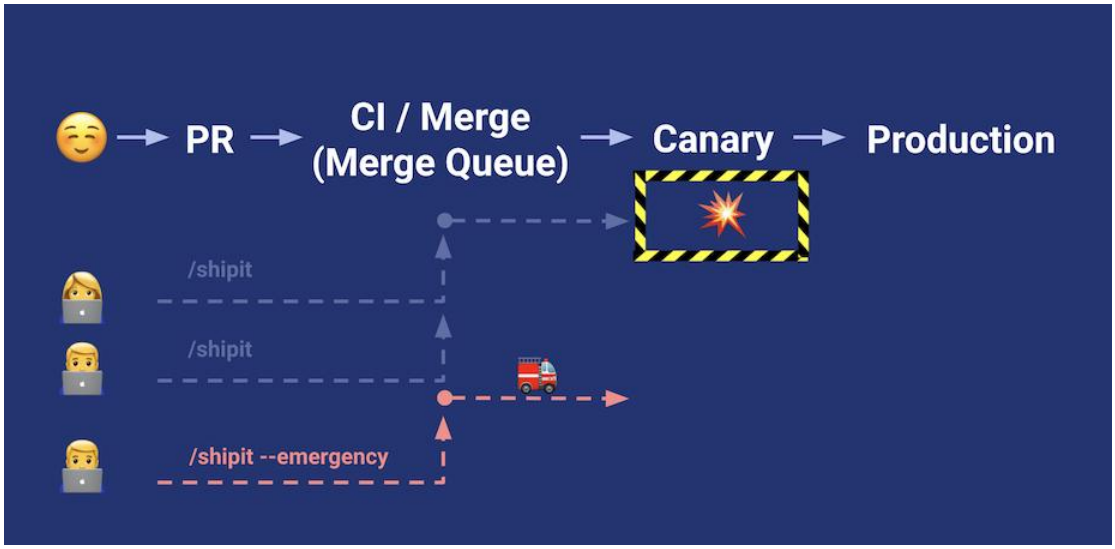

Figure 3: Shipit Diagram

Shipit[13] is a deployment tool that was built in house at Shopify and open sourced for the world to use. It's designed to make it easier for large development teams to frequently deploy their changes through the pipeline up to production.[14]

Features of Shipit:

- Audit trail to keep track of metadata of all deploys and rollbacks performed
- Deployment Dashboard

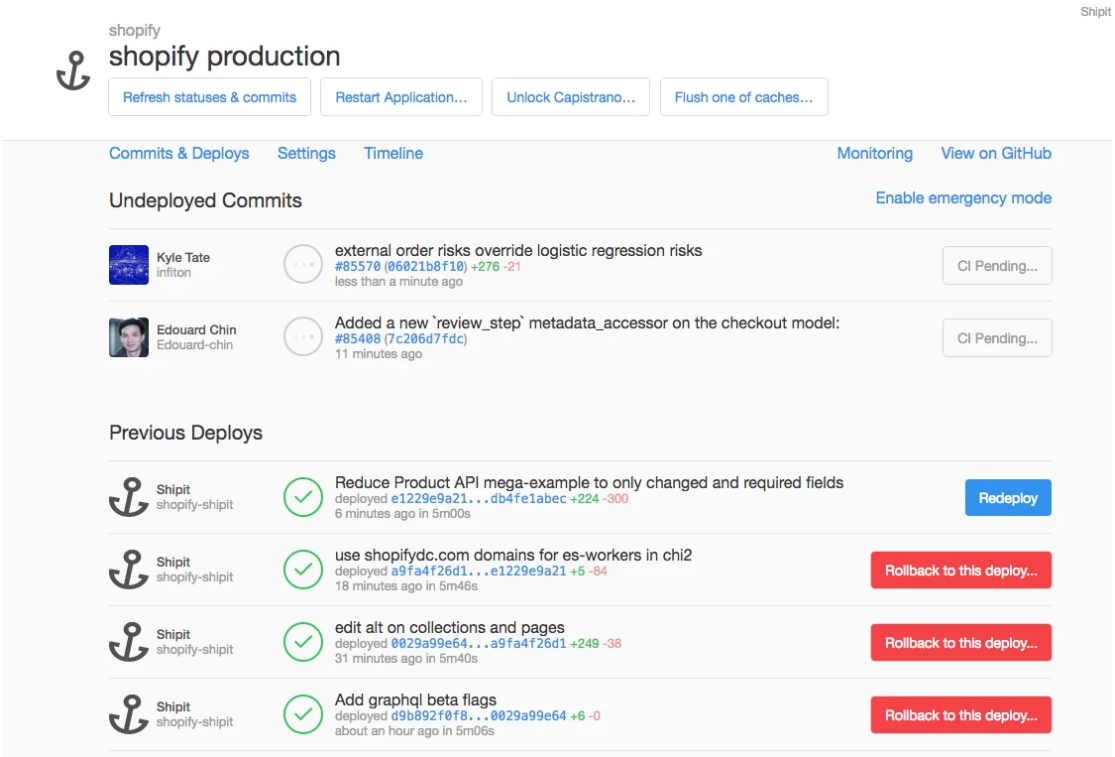

Figure 4: Shipit dashboard

- Chatbot integration for centralized communication and collaboration:

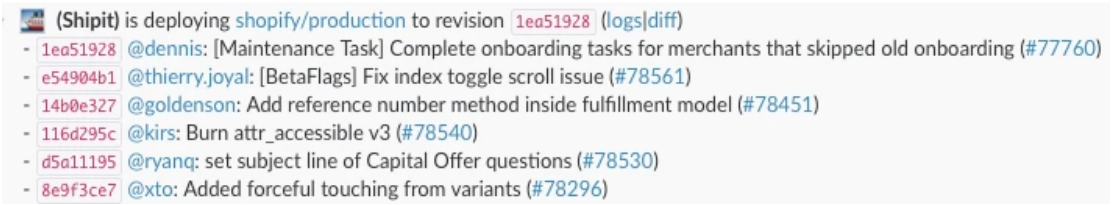

Figure 5: Shipit slack bot

- Also allows developers to issue commands to the chatbot from a centralized location:

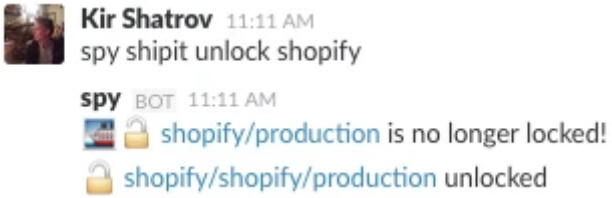

Figure 6: Shipit commands 2

The biggest downside of Shipit is that it's self hosted so last scalability and makes it harder to implement.

Heuristic Evaluation:

Shipit	**Reasoning**	**Score**
Communication & Collaboration	Shipit focuses on centralizing Communication & Collaboration with their chatbot that can interact with and monitor their pipeline. From Figures 4,5, and 6 you can see that it can handle interactive commands as well as provide updates on the status of deployments.	9
Visibility & Dashboards	Shipit provides a centralized dashboard that offers great visibility to all engineers in the company using it, as well as historical information on past deployments and the ability to interact with current and past deployments.	9
Scalability &	Shipit is self hosted and requires a database to function properly,	4

Implementation	it also requires setup for each separate application you use it for.	
Developer Experience	Shipit was designed with the Developer in mind from the very beginning[1]. They focused on improving existing developer workflows by integrating with Slack and Github and letting developers use the tools they want and improved from there with an easy to use chat bot and a polished dashboard.	8
Overall Score		30/40

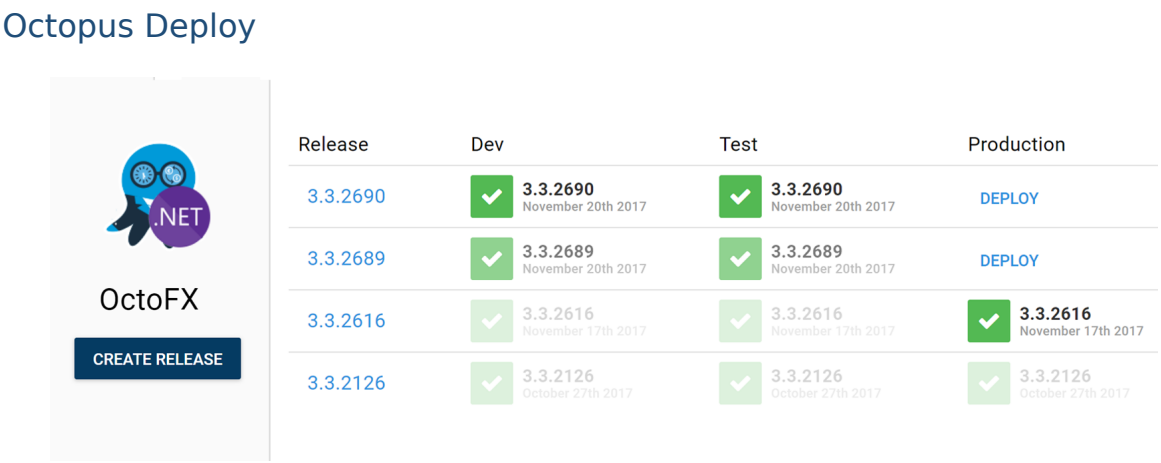

Figure 7: Octopus Deploy Dashboard

Octopus Deploy[16] is an automated deployment and release management tool, delivered in a single place for your team.

Features of Octopus Deploy:

- Single location for all your teams pipeline configuration and monitoring
- Nice UI design
- Easy setup

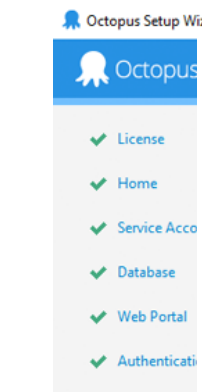

Figure 8: Octo

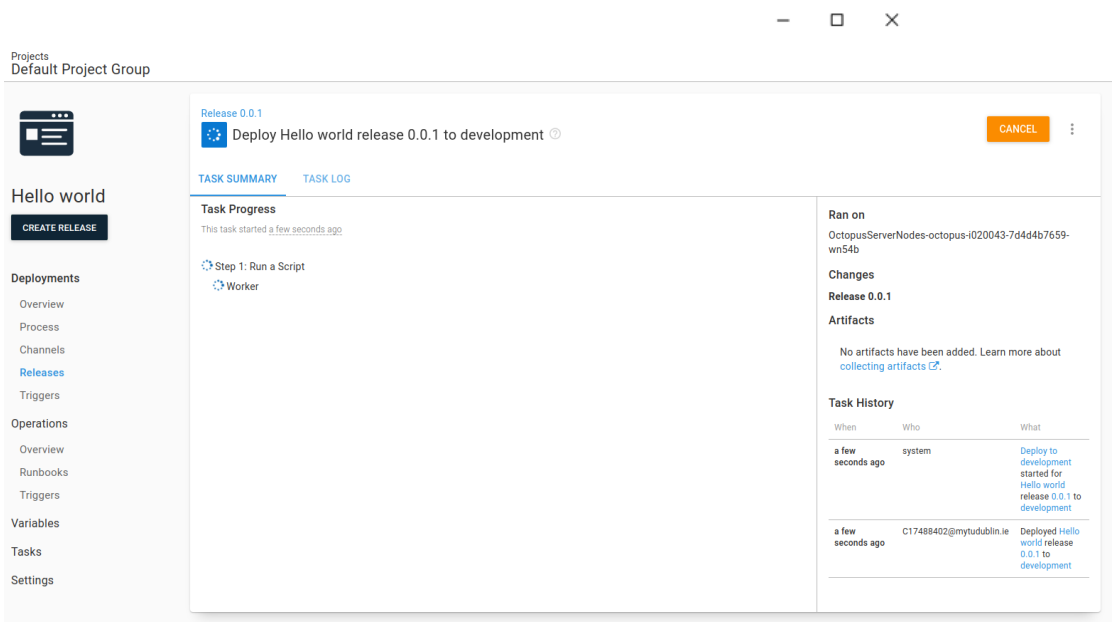

Figure 9: Octopus Deploy Project

Octopus does a lot of things well by being very opinionated on what makes good deployment and release software.[17] I like the UI of the release dashboard a lot and find it very intuitive.

The biggest downside of Octpus is the lack of Communication & Collaboration integration, which is left up to the user.

Heuristic Evaluation:

Octopus Deploy	**Reasoning**	**Score**
Communication & Collaboration	There's no focus directly on Communication & Collaboration, they provide tooling and guides for setting up your own chatbot though.	2
Visibility & Dashboards	They provide decent dashboards and system visibility, although it is not as centralized as Shipit and requires more clicking around through the UI to find out some information.	7
Scalability & Implementation	Nice walkthrough wizard to set up your application, you have the ability to use their cloud platform or self hosting which is a big benefit for scalability and implementation.	8
Developer Experience	The UI is nice and uniform, there's the option to integrate 100s of other developer tooling into the pipeline if you want to.	5
Overall Score		22/40

Jenkins

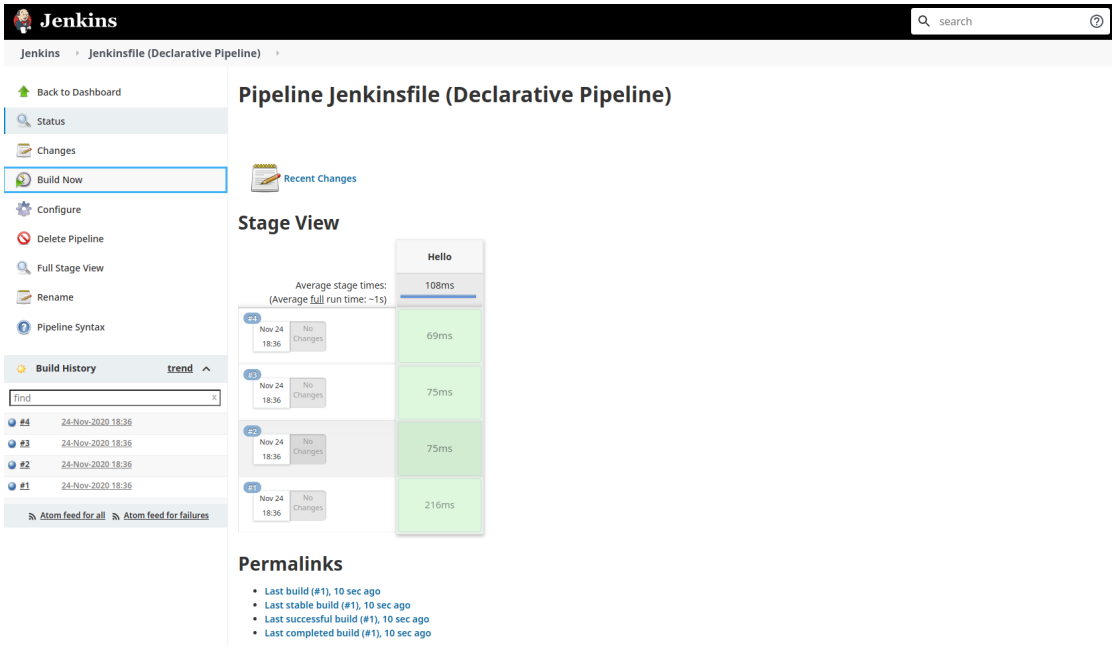

Figure 10: Jenkins Pipeline

Jenkins is an open source automation tool for continuous integration and delivery. It's the most popular CI tool available today with a market share of 71%.[18]

The advantages of Jenkins are:

- It's open source, so it can be self hosted
- A lot of custom plugins available to build to your workflow

The disadvantages of Jenkins are:

- No centralized communication & collaboration, can be achieved somewhat through plugins
- It has to be hosted somewhere, poor scalability and single point of failure
- It's hard to set up, usually a specific team or person is in charge of Jenkins.
- Hard to interpret dashboards

Heuristic Evaluation:

Shipit	**Reasoning**	**Score**
Communication & Collaboration	There's no focus directly on Communcation & Collaboration, but there are a lot of open source plugins that can help enable communication & collaboration.	2
Visibility & Dashboards	basic dashboards per pipeline let you know if a build is passing or failing, the option is there to improve these with plugins.	5
Scalability & Implementation	Self hosted, plenty of guides for integrating with existing systems available online but would require knowledge of the jenkins platform.	4
Developer Experience	Since it's such a popular tool, most developers are somewhat familiar with it so it offers the standard developer experience. Open to customization with plugins.	4
Overall Score		15/40

Conclusion

From my research into existing technologies, it's clear that there is still a lot of work to be done around the communication and collaboration aspects of CI/CD for most tooling. As well as the need for a better dashboard solution to highlight the important information about the build pipelines status.

The best solution by far was the shipit deployment tool that has been open sourced by shopify, which accomplishes a lot of the goals that a complete CI/CD pipeline should have with great Communication & Collaboration, Visibility into the system with dashboards and a nice Developer Experience.

2.3. Technologies you've researched

Introduction

At it's most basic level, our system will require:

1. Managed Version Control to store the code,
2. Continuous Integration Tool to regularly merge in changes,
3. Continuous Delivery Tool to automate deployment to production,
4. Infrastructure as Code to help automate the provisioning of the entire system,
5. A Chat bot that works well with a modern chat application to keep communication centralized,

6. A monitoring, logging and dashboard solution to feed information to the chat bot and offer visiblity into the build pipeline

From our literature review, we have also decided that the following features will be important to the applications success:

- Enable Communication & Collaboration
- Visibility into build process via centralized Dashboard
- Easy to scale and implement
- Hassle free Developer Experience (DX)

Managed Version Control System

Version Control Systems(VCS) are a way to keep multiple versions of your files in a tracked system.

This is essential in any modern software system so that if anything goes wrong you can revert back to a working version.

Aws CodeCommit

AWS CodeCommit is a fully-managed source control service that hosts secure Git-based repositories.[21]

CodeCommit is basically the AWS version of Github with less features(e.g.No issue tracking, forking).[22]

The advantages of CodeCommit is that it integrates well with existing AWS services, the disadvantages are: the UI, less features, and has slower performance[22]

Github

Github does everything CodeCommit does, and more(e.g. has issue tracking, forking). It's the most popular Managed VCS on the market[23].

It's easy to set up, easy to integrate with any other software, and has a nice UI that developers are familiar with. For these reasons Github will be the Managed VCS I've chosen to use in my project.

Continuous Integration

Continuous Integration is a practice where members of a team integrate their work frequently, usually multiple times a day.

Each integration is verified by an automated build to detect errors as quickly as possible.[19]

AWS CodeBuild

AWS CodeBuild is a fully managed continuous integration service. The main advantages of CodeBuild are that it's a:

- Fully managed build service, this is important for scaling and implementation
- Pay as you go, which will keep costs down over other SaaS CI products.

- Secure and extensible with lots of AWS services.

Travis CI

Travis CI is a hosted continuous integration service for open source and private projects. Travis CI has a lot of great pluses like good Github integration, but for scalability, security and easier integration with other AWS Products I plan on using AWS CodeBuild for my project.

Continuous Delivery

Continuous Delivery is the ability to get changes of all types—including new features, configuration changes, bug fixes and experiments—into production, or into the hands of users, safely and quickly in a sustainable way.[7]

AWS CodePipeline

AWS CodePipeline is a fully managed continuous delivery service. It integrates seamlessly with other AWS products.

Spinnaker

Spinnaker is a multi cloud continuous delivery platform that can be used with multiple cloud environments. The downsides of Spinnaker is the set up, config and management overhead of it. That's why I'll be choosing AWS CodePipeline instead for my project

Infrastructure as Code

Infrastructure as Code(IaC) is automating the provisioning of infrastructure through config files, so that it's versioned similar to regular software. This makes sure it's scalable, and versioned if anything goes wrong.

Modern IaC requires you to use a high level language(YAML/JSON) to describe what you want, outlined below are my two choices:

Terraform

Terraform is the most popular IaC tool, it's cloud agnostic and well documented.

CloudFormation

CloudFormation is an IaC tool built by AWS specifically for their platform so integrates well with all AWS services.

Since it's purpose built for AWS and integrates so well I plan to use it as my IaC tool of choice.

Chat Application & Chat Bot

For Chat Application I've the choice between Microsoft Teams, or Slack. Slack has way more integrations with external applications like chatbots. [23]

AWS Chatbot/Lambda

AWS Chatbot is an interactive agent that makes it easy to monitor and interact with your AWS resources in your Slack channels.[24]

Lambda is a Function as a Service that can execute any function you give it without worrying about the server in the background.

They both enable the system to have the ability to send notifications in real time, and they integrate well with the AWS ecosystem for quick setup.

Hubot

GitHub, wrote the first version of Hubot to automate their company chat room[25]

There's a lot more setup and integration involved in setting up Hubot for this project, so for this reason I've chosen to use AWS Chatbot/Lambda combo

Monitoring, Logging

Monitoring and logging enable me to deliver information with a chatbot or visually via a dashboard.

AWS Cloudwatch

Amazon CloudWatch is a monitoring and observability service. It offes an easy way to collect, monitor and gain visibility into your AWS operations and build pipelines.

Splunk

Splunk [26] is a software platform to search, analyse and visualize the machine-generated data gathered from the websites, applications, sensors, devices etc. which make up your IT infrastructure and business.

Splunk is used in conjuction with Cloudwatch and would injest data from it. It seems too heavyweight for a basic dashboard and monitoring solution, so I'll just be going with Cloudwatch by itself.

Dashboard

The dashboard itself has to be built that is fed in the logs from cloudwatch.

I plan on utilizing Javascript & React to build out the dashboard solution for this project because of the vast array of libraries available for developing dashboards in the javascript and react ecosystem.

Conclusion

My technology choices have been guided by my earlier literature review, they are currently:

- Version Control: Github
- Continuous Integration: AWS CodeBuild
- Continuous Delivery: AWS CodePipeline
- Infrastructure as Code tool: Aws CloudFormation

- Chat Application & Chatbot: Slack & AWS Chatbot/Lambda
- Monitoring, Logging: Aws CloudWatch
- Dashboard: Javascript & React

These tool fulfil the criteria I want of a platform that prioritizes developer experience, communication & collaboration, and visibility through a central dashboard.

3. Design

3.1 Introduction

In this chapter I will discuss choosing a methodology for this project, the feature list which will be based on the user requirements, and designs for the UI for the chatbot & dashboard, and the technical architecture design for the chatbot and dashboard.

3.2. Software Methodology

A software development methodology is a series of steps that you choose to go through when delivering software from gathering requirements all the way through to deployment and maintenance. The two methodologies I'll be discussing are the Waterfall model, and the Agile Model.

Waterfall model

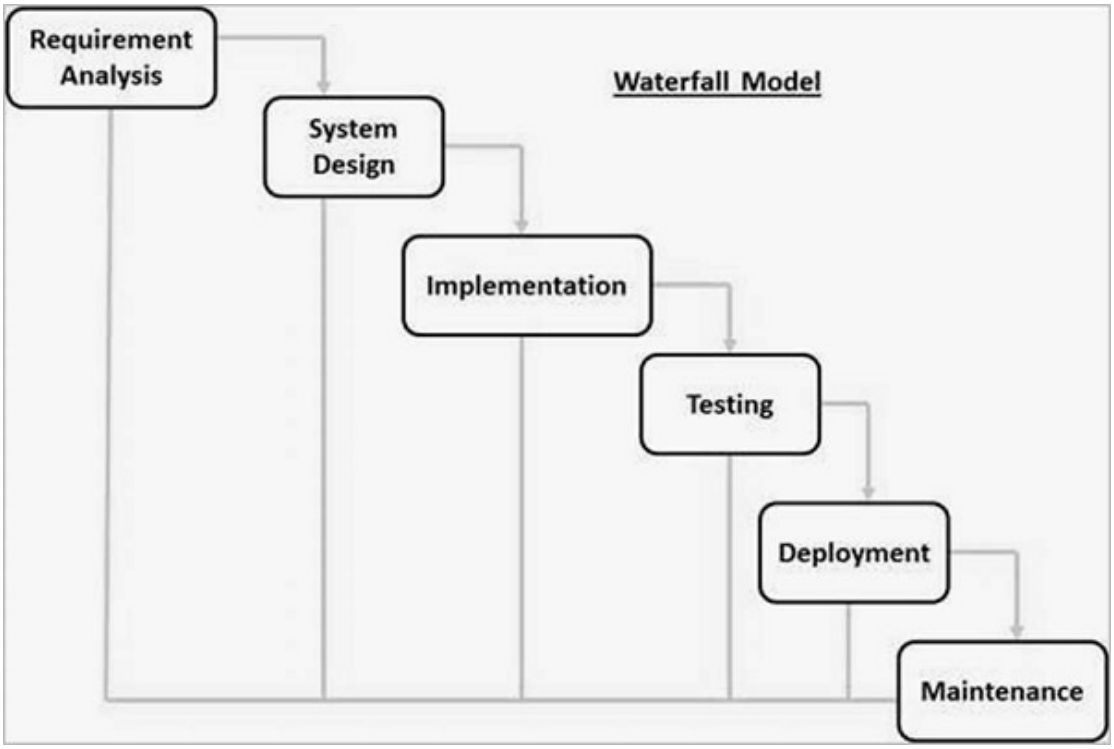

Figure 11: Waterfall Model

The Waterfall methodology was one of the first approaches widely adopted for software development. It consists of the following stages in order:

1. Requirements gathering and analysis
2. System Design
3. Implementation
4. Integration and Testing
5. Deployment of system
6. Maintenance

All these phases are done in a linear fashion and "flow" downhill like a waterfall, once 1 is done you start 2 and don't go back up etc. The waterfall model is rigid in design and doesn't work well with a modern development approach that seeks to have small incremental releases and constant feedback.

Agile Model

Agile SDLC model is a combination of iterative and incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software product. Agile Methods break the product into small incremental builds [28]

Agile focuses on keeping the process lean and creating minimum viable products (MVPs) that go through a number of iterations before anything is final. Feedback is gathered and implemented continually and in all, it is a much more dynamic process where everyone is working together towards one goal. [29]

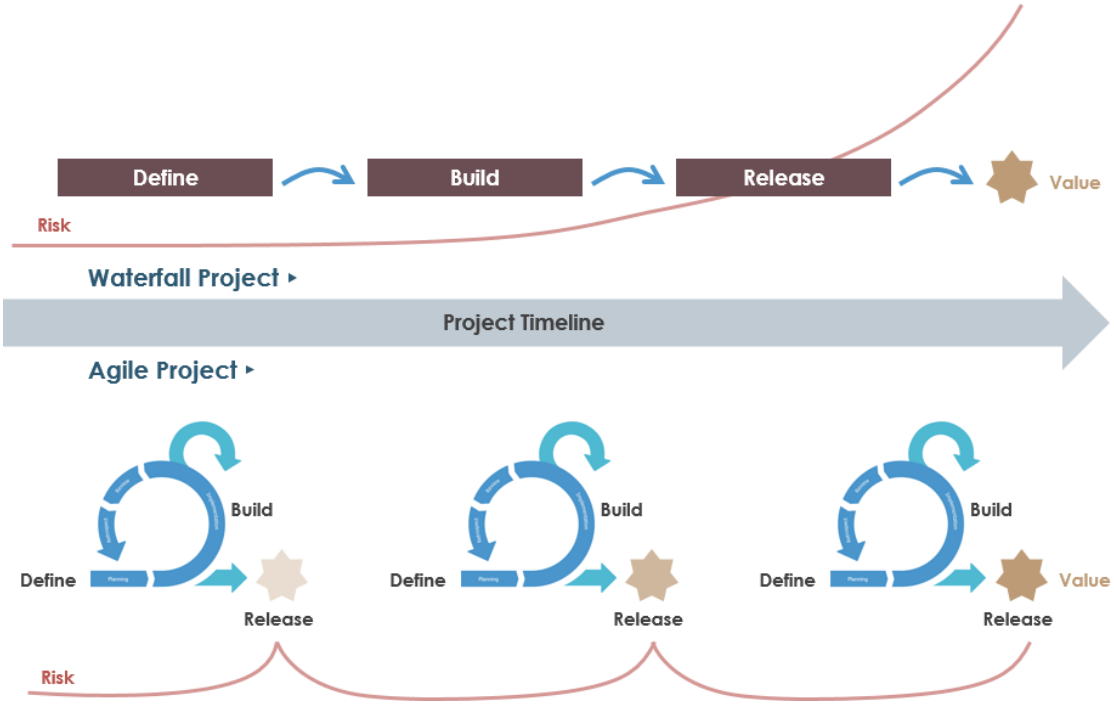

Figure 12: Waterfall vs Agile

Since my project is all about fast iterative development and releasing working software quickly, Agile is an ideal choice.

There are many different brands of agile software methodologies, like extreme programming or scrum. For my project I've chosen to use the Feature Driven Development(FDD) variant of Agile.

FDD was designed to follow a five-step development process, built largely around discrete “feature” projects. That project lifecycle looks like this:

1. Develop an overall model
2. Build a features list
3. Plan by feature
4. Design by feature
5. Build by feature

3.3 Requirements

User Requirements

The user requirements are based on research conducted in chapter 2 and focus on:

- Enable Communication & Collaboration
- Visibility into build process via centralized Dashboard
- Easy to scale and implement
- Hassle free Developer Experience (DX)

The user requirements for the CloudFirst chatbot and dashboard are as follows:

- As a developer, I need access to the status of the system and build pipeline at all times through a dashboard or chatbot command
- As a developer, I need to be able to deploy code from a centralized chatroom or dashboard without having to jump through a lot of hoops
- As a developer, I need to be keep up to date with the status of other deployments happening to the repository
- As a developer, If something goes wrong with a deployment I want to be alerted in the chat application deployment channel
- As a developer, If some part of the build pipeline needs my approval I should be able to do it at the click of a button from a chat bot
- As a developer, I shouldn't have to worry about the inner workings of the deployment process, it should be intuitive when interacting with the chat bot and dashboard.

Now that we have the user requirements we can develop a Feature List

Feature List

A feature list is a list of client valued pieces of functionality

Feature List	Priority
Start build from commit to repo	HIGH
Display basic information about current build via chatbot	HIGH
Display basic information about current build via dashboard	HIGH

Display basic information on request about status of specific build via chatbot	HIGH
Approve build with chatbot command	HIGH
Approve build with dashboard interaction	HIGH
Cancel build with chatbot command	HIGH
Cancel build with dashboard interaction	HIGH
Approve build with chatbot command	HIGH
Approve build with dashboard interaction	HIGH
Stop build with chatbot command	HIGH
Stop build with dashboard interaction	HIGH
Restart build with chatbot command	HIGH
Restart build with dashboard interaction	HIGH
Redeploy past build with chatbot command	HIGH
Redeploy past build with dashboard interaction	HIGH
Display detailed view of specific builds via dashboard	MEDIUM
Receive detailed specific build information via chatbot command	MEDIUM

3.4. User Interface Design

Introduction

This section will go through how I developed the user interfaces for the chatbot and for the dashboard.

The chatbot user interface will be chat messages sent via a Slack chat bot. This limits the scope of the design for these messages to chat features that Slack provides.

Chatbot message design

Developers need to be able to interpret the chatbot messages quickly, here's a list of types of messages the chatbot will send to the channel.

- Build deploying(starting):

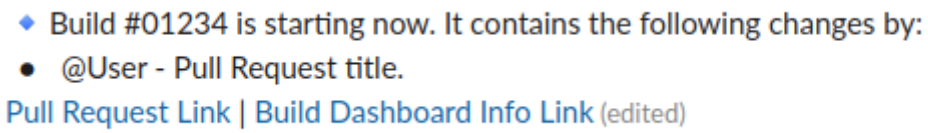

Figure 13: Deploy Message Design

- Build stopped

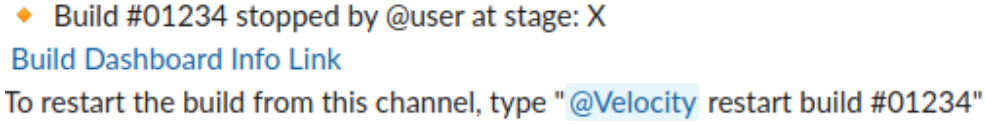

Figure 14: Stop Message Design

- Build failed

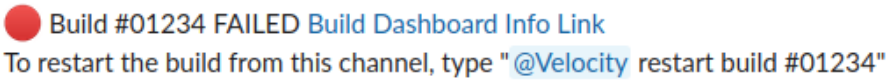

Figure 15: Fail Message Design

- Build passed stage X, proceeding to Y

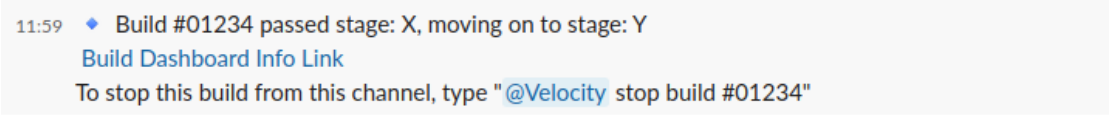

Figure 16: Pass Message Design

- Build deployed(Success)

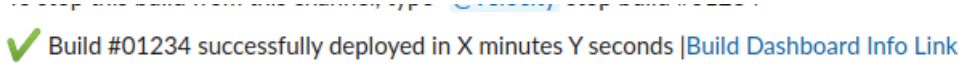

Figure 17: Success Message Deisgn

- Build requires attention

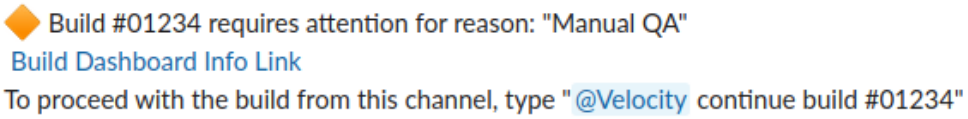

Figure 18: Attention Message Design

- Pinned Message with link to dashboard at top of channel and bot instructions:

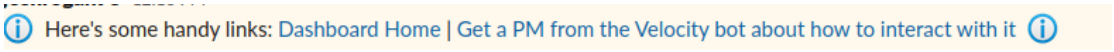

Figure 19: Pinned Message Design

Dashboard UI Prototyping

I initially did a low fidelity prototype on paper, taking inspiration from some of the dashboards I reviewed in chapter 2.

Low Fidelity Prototype

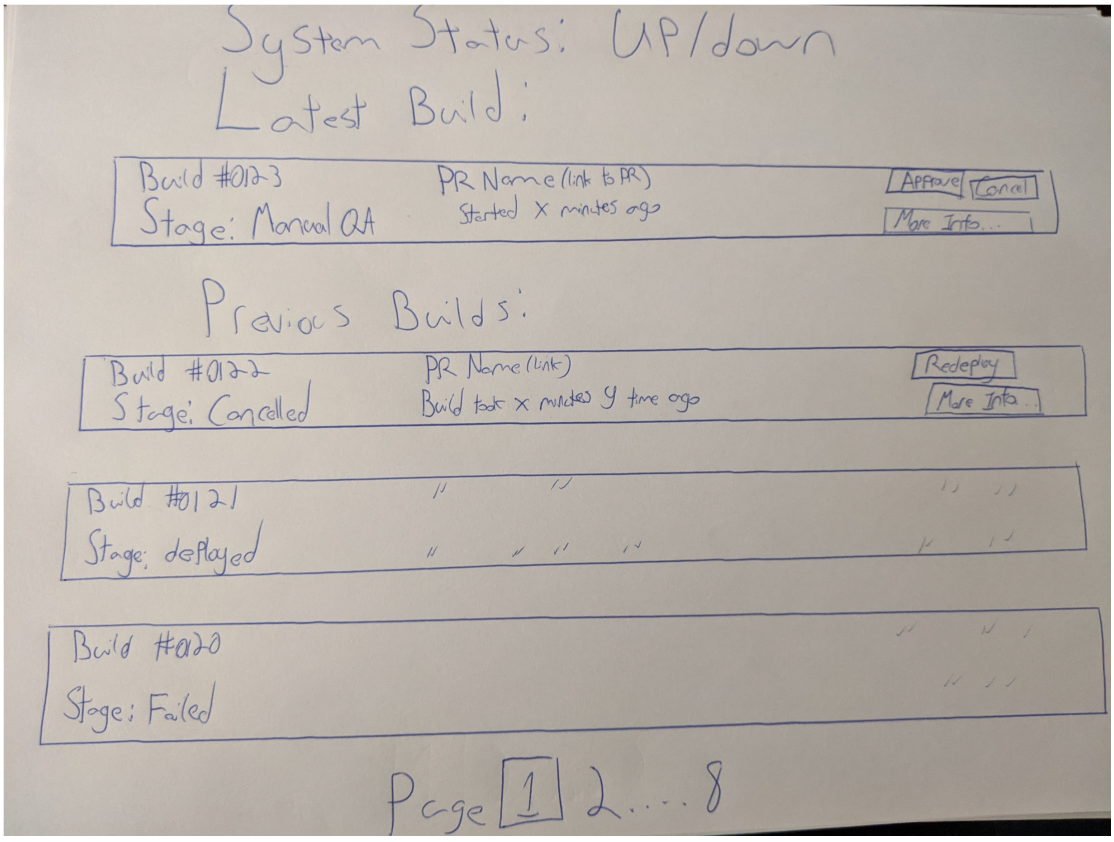

Figure 20: Low Fidelity Dashboard

Once I had this sketched out I was able to design a high fidelity prototype, I've chosen to use basic HTML, CSS & Bootstrap to mock up the high fidelity prototype:

High Fidelity Prototype

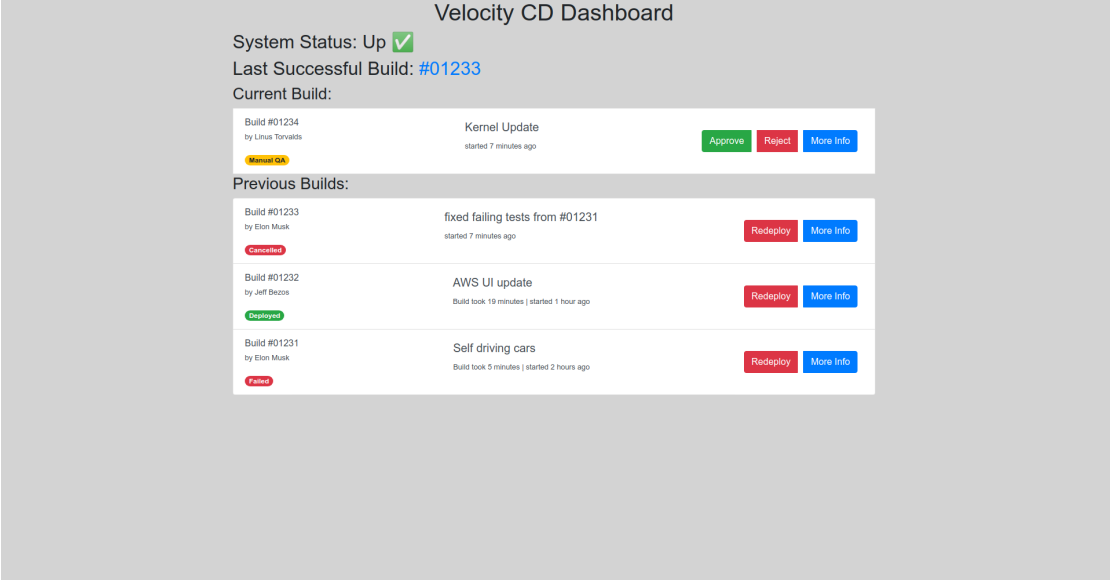

Figure 21: High Fidelity Prototype

3.5. System Overview

Before designing the technical architecture it's important to have an overview of how the system behaves. To help visualize this I've created a use case diagram that shows how the developer can interact with the system as a whole. The system in this case is the combined chatbot and dashboard.

Use Case Diagram

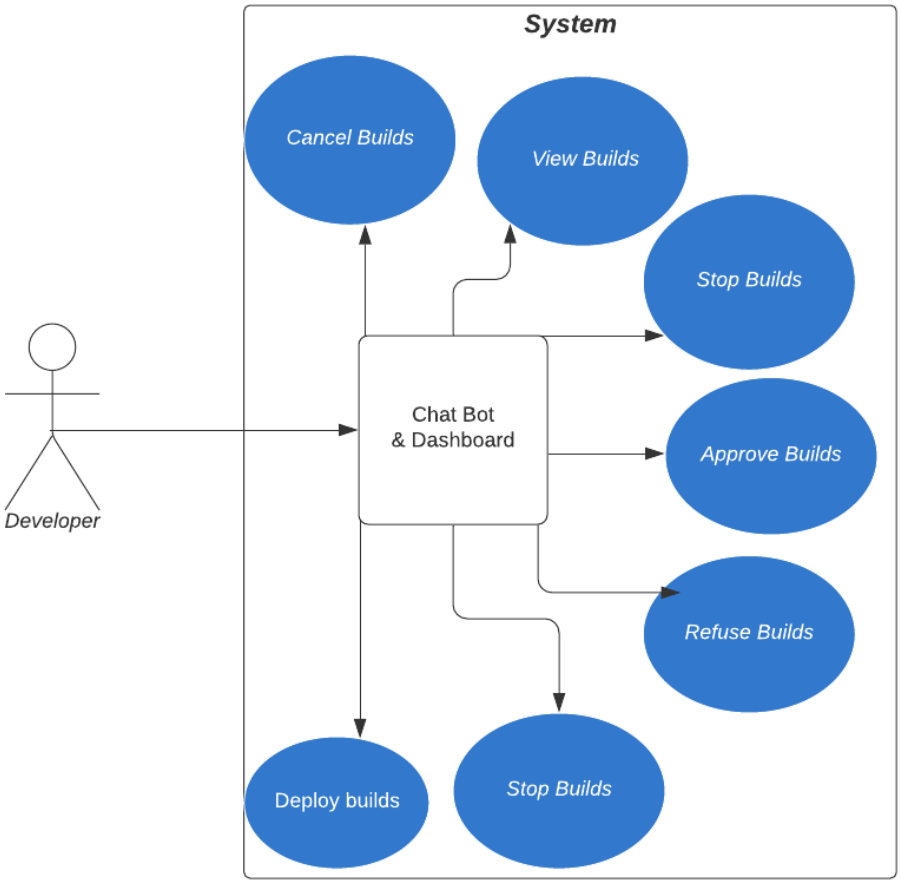

Figure 22: Use Case Diagram

3.6. Technical Architecture

The Technical Architecture was created with LucidChart

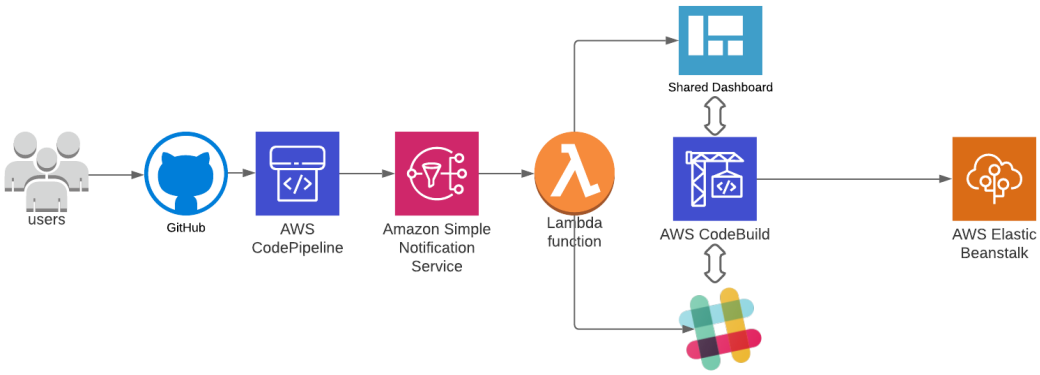

Figure 23: Pipeline Architecture

Components:

- Github: Managed Version control system, where the source code lives.
- AWS CodePipeline: Continuous Delivery service to help automate pipelines
- AWS Simple Notification Service(SNS): Pub/Sub Messaging service
- Lambda: serverless function that is triggered by SNS. It will integrate with Slack and the dashboard which will control the CodeBuild component
- AWS CodeBuild: Continuous Integration service
- AWS Elastic Beanstalk: Simple managed server for web app deployment
- Dashboard & Slack: These are how the user will interact with the pipeline once it has started. They both share the same functionality laid out in the use case diagram in section 3.5

Flow of Pipeline:

- When the developer's code is committed to Github and merged, it starts the CodePipeline process.
- CodePipeline then publishes a notification that starts a Lambda function which notifies the slack chatbot with a web hook and sends it whatever information you want.
- The developer will be able to interact with either the chatbot or the dashboard, to monitor or interact with builds that are run via CodeBuild.
    - CodeBuild is embedded into the CodePipeline here
- Once the application is ready to be fully deployed it's deployed to Elastic Beanstalk.

The architecture is subject to change during the developement stage as I learn more about what each AWS service has to offer.

4. Development

4.1 Introduction

My project is comprised of two applications: The dashboard and the chatbot. On top of these applications the underlying cloud infrastructure has to be developed as well. So this chapter will be broken down into three main sections: Infrastructure, Dashboard, and Chatbot. In these sections I'll break down the challenges encountered during implementation, their architecture, and give a detailed insight into how they were developed.

Dashboard code: https://github.com/Joshrogan/dashboard

Infrastructure and chatbot code:

https://github.com/Joshrogan/Infrastructure-And-Chatbot

4.2 Infrastructure

CloudFormation

After developing my prototype using the AWS console GUI I planned to switch to writing my Infrastructure as Code(IaC) using AWS CloudFormation with the YAML language. However I ran into many issues trying to write CloudFormation by myself having never used it before this final year project. Some of the regular issues I ran into were Insufficient IAM permissions [31], Dependency errors [32], and problems rolling back and deleting stacks[33] [34].

From searching the internet I found out that Amazon released a product with a new way to write and deploy IaC in 2019 that is considered more developer friendly. The new product was called the AWS Cloud Development Kit(CDK)[35]

AWS Cloud Development Kit

The CDK is a is a software development framework for defining cloud infrastructure in code and provisioning it through AWS CloudFormation.[35] The CDK lets you write your IaC using a high level programming language like Java, Python, or Typescript. The CDK then synthesizes it down to CloudFormation Templates which are deploys to AWS.

I initially started using the CDK with Python as the high level language, I kept having issues with python packaging and decided to switch to Typescript, a typed superset of Javascript since it's the language I'm the most familiar with. The added benefit of using Typescript for the infrastructure is that I would be using it for the dashboard development as well.

Pipeline Architecture

An AWS CodePipeline is made up of stages, each stage contains actions that are performed on artifacts(e.g. source code, built application). The action types are: source, build, test, deploy, approval, and invoke. [36]

The structure of a pipeline requires it to have at least:[36]

- 2 stages
- First stage is source action

For each action there's several options for valid action providers, to narrow the scope of this project I've chosen to build my pipeline to incorporate a source action stage, a build/test action stage, an approval action stage, and a deploy action stage.

Source Action

To setup a Github repository as the source action stage, AWS would need access to the project Github. To do this I created a Github Personal Access Token that I stored in AWS Secrets Manager. This allows me to use it in my CDK code safely without the risk of leaking it anywhere.

Figure 25: CDK source code to get github-token

Deploy Action

In the prototype I used AWS ElasticBeanstalk which is an all in one application hosting service managed by AWS. This is harder to integrate with CDK than I anticipated so I switched it out for Amazon S3. S3 stands for Simple Storage Service.

It's AWS' global object storage and it is great for hosting simple single page applications(SPA) and static websites if configured correctly. The application I would be building would be a react app that is eventually "built" into a bunch of basic html, css, and javascript files.

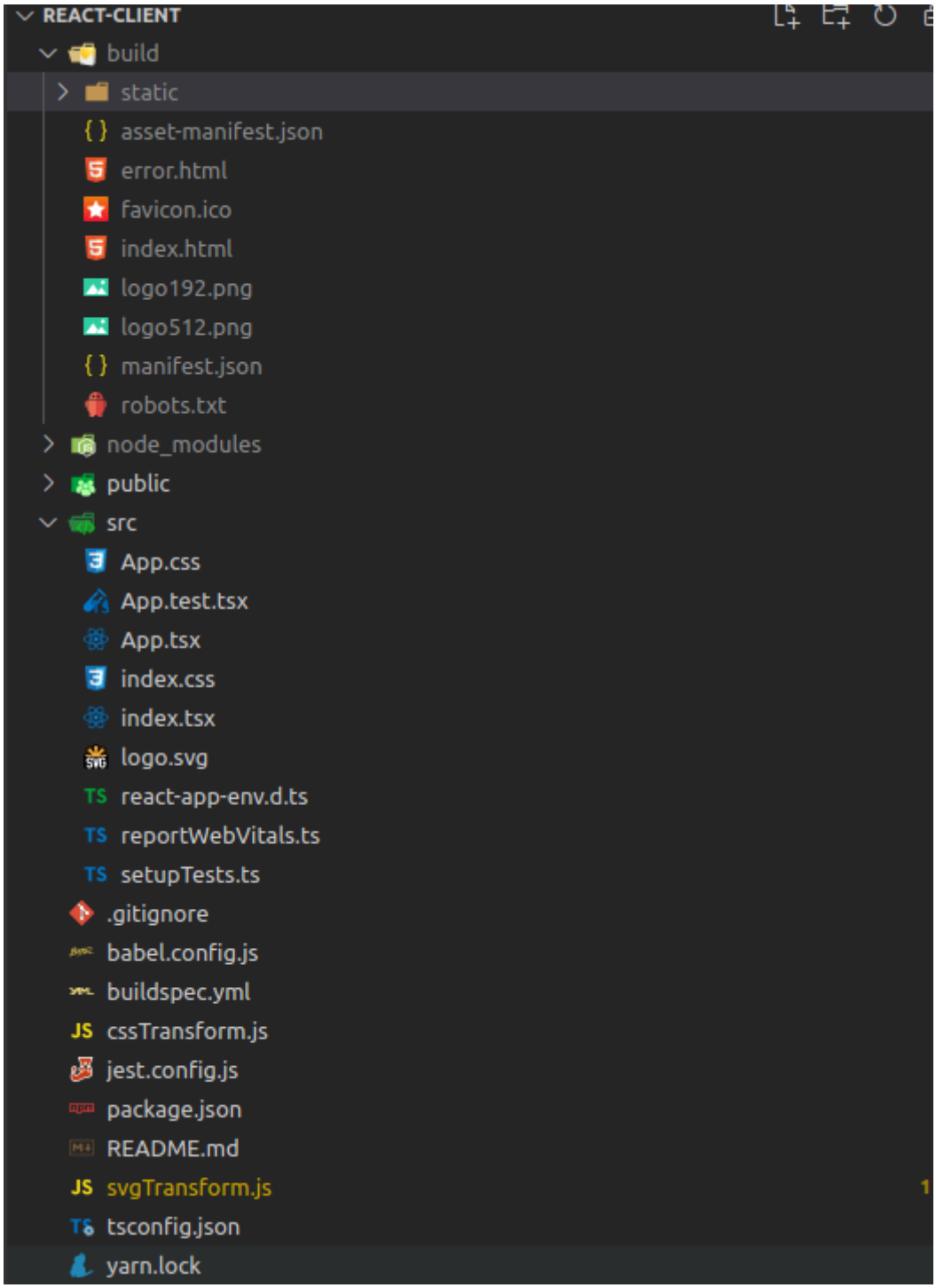

Figure 26: Built React App

The src/ and public/ folders are compiled down to the build/ folder which is just a bunch of javascript,html,css,image files and json.

CDK Application Structure

To deploy the infrastructure necessary to host a pipeline and a SPA I decided to split my CDK Application into 3 seperate logical stacks based on a blog detailing a similar process[38]. Stacks in CDK are a basic unit of deployment.

Figure 27: CDK Stacks

The first stack created had to deal with domain certificates, I initailly deployed this stack in the irish AWS region eu-west-1(dublin above). But this caused errors. The problem was since Route53 is a global service, that all certs must be registered in us-east-1 for global services to work.

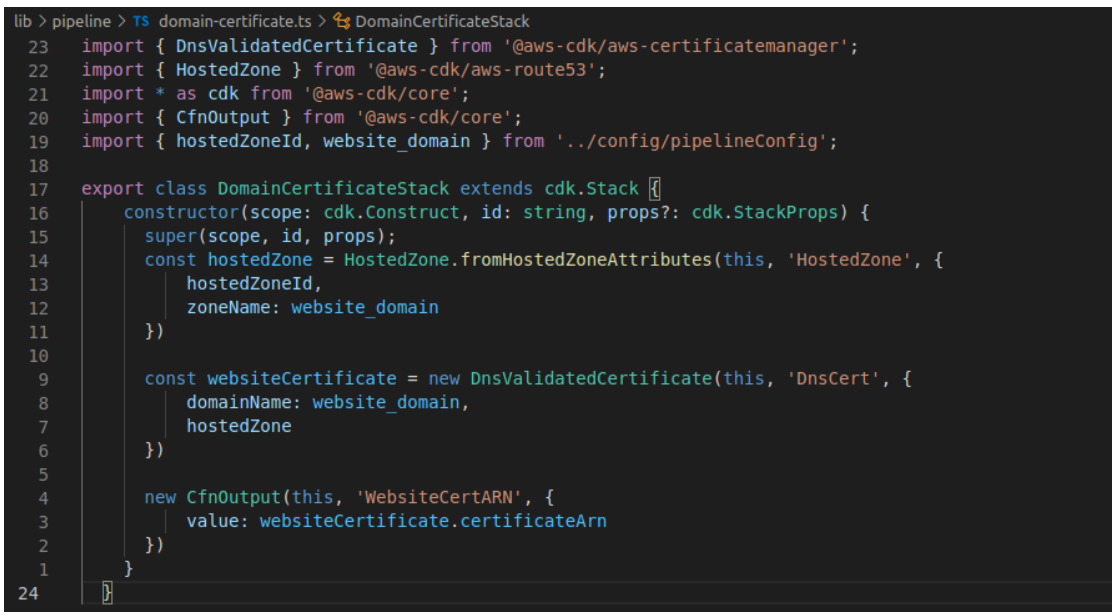

Figure 28: Domain Cert

The power of CDK is really displayed in the above Domain Cert image. After registering a domain with AWS you'll be assigned a hosted zone ID, with just this and the domain you're able to leverage the CDK to generate the certificates and resources necessary to deploy a fully functioning website.

The next stack required to deploy an S3 Bucket as a website was the CloudFrontStack. In this stack the S3 Bucket was created, The Web Distribution for CloudFront[37] that points to the S3 Bucket as it's origin, and redirects all non https requests to http.

The final stack deployed was the CodePipelineStack. This contains the logic for the basic Pipeline infrastructure alongside the custom construct I wrote for the chatbot to send messages to slack channels called PipelineEvent.

```
export class PipelineEvent extends Construct {

    constructor(scope: Construct, id: string, props: PipelineEventProps) {
      super(scope, id,);

      const eventPipeline = EventField.fromPath('$.detail.pipeline');
      const eventState = EventField.fromPath('$.detail.state');
      const eventFull = EventField.fromPath('$.detail');

      props.pipeline.onStateChange('OnPipelineStateChange', {
          target: new SnsTopic(props.topic, {
            message: RuleTargetInput.fromText(`Pipeline ${eventPipeline} changed state to ${eventState}, eventFull ${eventFull}`),
          }),
        });

        const target: CfnNotificationRule.TargetProperty = {targetType: 'SNS', targetAddress: props.topic.topicArn}

        new CfnNotificationRule(this, 'statusChangeNotificationRule', {
            detailType: 'FULL',
            eventTypeIds: [
               "codepipeline-pipeline-action-execution-succeeded",
               "codepipeline-pipeline-action-execution-failed",
               'codepipeline-pipeline-stage-execution-started',
               'codepipeline-pipeline-pipeline-execution-started'
              ],
              name:'notificationRuleName',
              resource: props.pipeline.pipelineArn,
              targets: [target]
            })


            let fn = new Function(this, 'SODemoFunction', {
                runtime: Runtime.NODEJS_10_X,
                handler: 'slack-alert.handler',
                code: Code.fromAsset(path.join(__dirname, 'functions'))
             });

              props.topic.addSubscription(new LambdaSubscription(fn))

    }
}
```

Figure 29: Custom Pipeline Event Construct

In PipelineEvent I pass in a pipeline which represents a AWS CodePipeline resource and a topic which represents a Simple Notification Service(SNS) topic which is a simple pub/sub messaging service provided by AWS.

The construct publishes a custom message when the pipeline resource changes state that corresponds to one of the provided `eventTypeIds`. It does this by adding the lambda function as a subscriber to the SNS topic.

```
var https = require('https');
var util = require('util');

exports.handler = function(event, context) {
    console.log(JSON.stringify(event, null, 2));
    console.log('From SNS:', event.Records[0].Sns.Message);

    var postData = {
        "channel": "#final-year-project",
        "username": "Velocity",
        "text": "*Dev us-west-2 Notifications*",
        "icon_emoji": ":rotating_light:"
    };

    var message = event.Records[0].Sns.Message;

    postData.attachments = [
        {
            "text": message
        }
    ];

    var options = {
        method: 'POST',
        hostname: 'hooks.slack.com',
        port: 443,
        path: '/services/T01G1B6T4P4/B01FLJYP90X/k3L1yDga7npDpBwcWmjXsVhe'
    };

    var req = https.request(options, function(res) {
      res.setEncoding('utf8');
      res.on('data', function (chunk) {
        context.done(null);
      });
    });

    req.on('error', function(e) {
      console.log('problem with request: ' + e.message);
    });

    req.write(util.format("%j", postData));
    req.end();
};
```

Figure 30: Slack Alert Lambda

The lambda function sends off a POST request that uses the slack webhook to post to a specific channel.

CDK Deployment

CDK stacks are deployed through the CLI. Before deploying your first CDK applications you have to boostrap CDK in your AWS region. This is accomplished by running cdk bootstrap command in the CDK application root directory. This provides CDK the resources and permissions it needs to deploy other apps.[39] Then you can deploy your stacks by running cdk deploy $STACK_NAME.

Figure 31: CDK Deploy Snippet #1

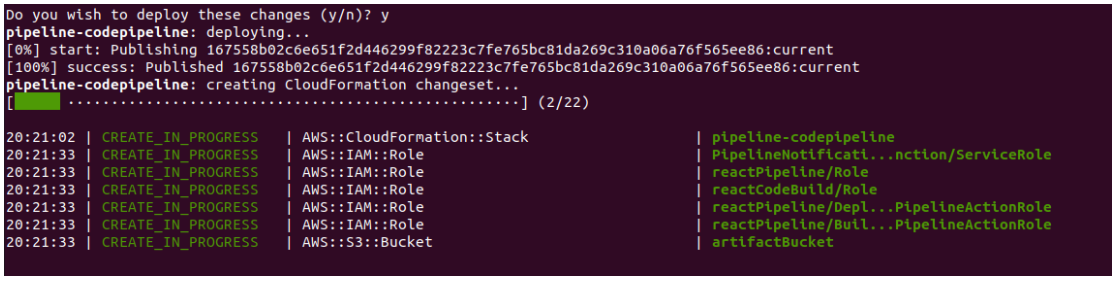

Figure 32: CDK Deploy Snippet #2

To delete resources, you just speicify the stack name in the command `cdk destroy $STACK_NAME`.

By utilizing the CDK to build out the underlying infrastructure for my dashboard and chatbot, I'm able to keep the infrastructure modular, extensible, and reusable which is important for adding or removing pipeline stages and actions in the future easily.

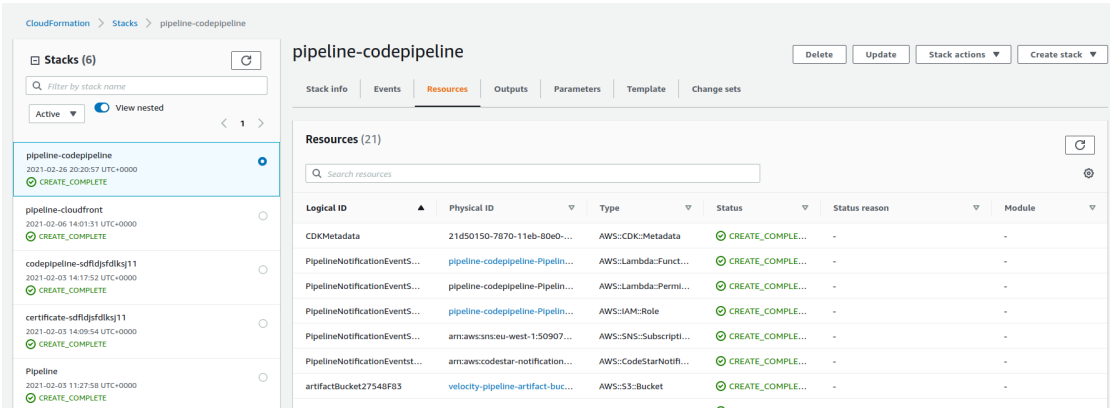

Figure 33: CloudFormation Stacks

4.3 Dashboard

The dashboard of my project originally was going to be written in Javascript. However I switched the language to Typescript since my infrastructure was already being written in Typescript.

Typescript's dynamic typing helps a lot when working with well documented external APIs like the AWS SDK[40] I would be using for this project. It makes it easy to check what arguments need to be passed and their types from inside your IDE/Code Editor directly.

Material UI[41] is the component library I would choose to help build out my dashboard. They provide standard React components based on Google's

Material Design which helps to build out high-quality and cross platform interfaces across devices. [42]

Dashboard Architecture Overview

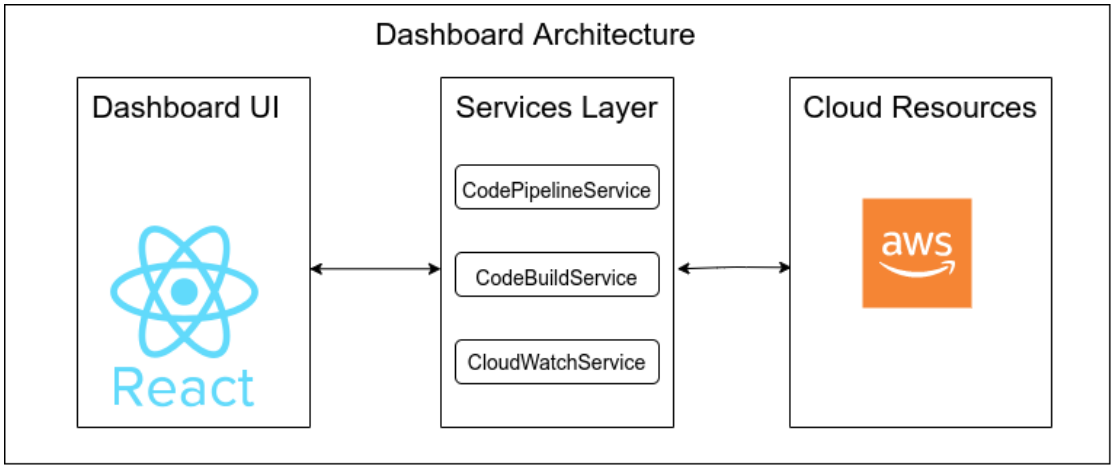

Figure 34: Dashboard Architecture

Dashboard UI

To be easy to use and share what pipeline you're looking at with other team members requires URL routing. React doesn't come with the functionality to route URL's out of the box. To make up for this one of the first libraries I needed to install was React Router[43]. React Router let's users share or bookmark specific pipelines instead of having to navigate through the webpage from the root every time.

The two main routes in my application are:

- "/" the Root URL which display the homepage, which renders a list of all current pipelines
- "/pipeline/:pipelineName" this path renders a specific pipeline's stages by the pipeline name passed in.

```
return (
  <Container maxWidth="lg">
    <Router>
      <Typography variant="h2">
        <RouterLink to="/">Velocity CD Dashboard</RouterLink>
      </Typography>
      <Route exact path="/">
        {pipelines?.map((pipeline) => (
          <Pipeline pipeline={pipeline ?? null} key={pipeline?.pipelineName} />
        ))}
      </Route>
      <Route
        exact
        path="/pipeline/:pipelineName"
        render={(props) => (
          <Stages
            pipeline={pipelines.find((pipeline) => pipeline.pipelineName === props.match.params.pipelineName)}
            pipelineClient={codePipelineClient}
          />
        )}
      ></Route>
    </Router>
  </Container>
);
}
```

Figure 35: Dashboard Routing

Dashboard UI Components

Homepage

The initial view a user sees when navigating to the dashboard will be the "/" root view of the pipelines. Each pipeline renders a <Pipeline> component which displays an overview into it's status, when it was last updated, and the latest commit. There's also a link which will render the pipeline's stages by routing to "/pipeline/:pipelineName".

```
return (
  <div className={classes.root}>
    <Card className={classes.card} raised={true}>
      <CardHeader
        className={classes.cardHeader}
        title={pipeline.pipelineName}
        subheader={'Status: ' + pipelineStatus}
        action={<RouterLink to={`/pipeline/${pipeline.pipelineName}`}>More Info</RouterLink>}
      />
      <CardContent>
        {'Last Updated: '}
        {lastUpdateTime && <ReactTimeAgo date={lastUpdateTime} />}
      </CardContent>
      <CardContent>
        {'Latest Commit: '}
        {latestCommitSummary}{' '}
        {
          <Link href={latestCommitUrl}>
            {latestCommitId?.substring(0, 9)} {<LaunchIcon fontSize={'inherit'} viewBox={'0 0 24 18'} />}
          </Link>
        }
      </CardContent>
    </Card>
  </div>
);
```

Figure 36: Pipeline Component Code

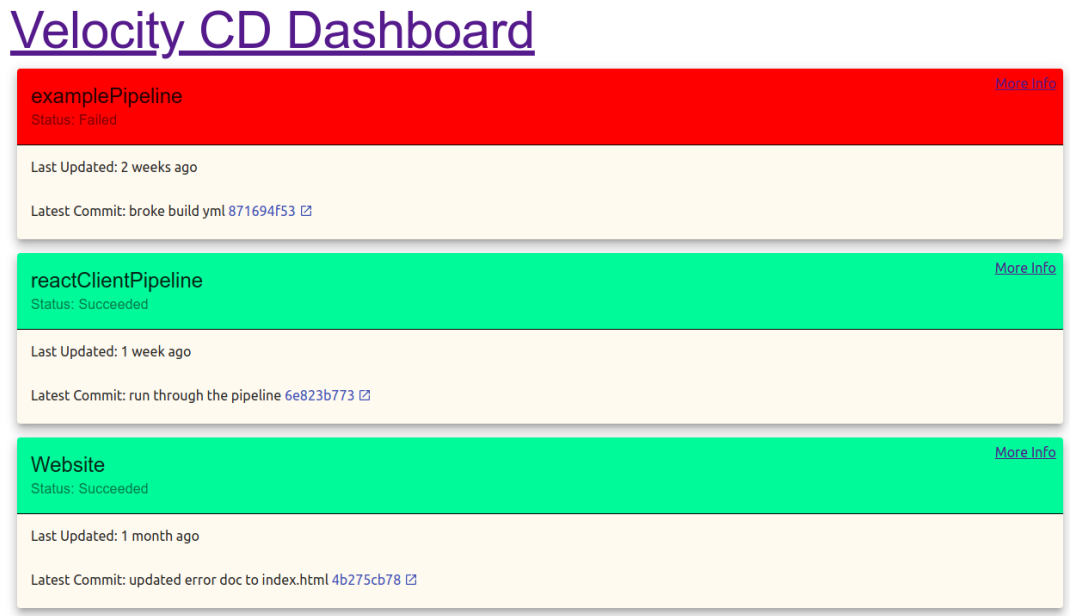

Figure 37: View of rendered Pipelines

Pipeline Stages

When a users clicks the more information button on one of the <Pipeline> components on the homepage they are routed to the <Stages> View of that Pipeline.

This renders a page that contains all the stages of the pipeline, the overall pipeline status and the ability to restart the entire pipeline via a button above the array of stages. Clicking on the individual stage will open a <Stage> component which contains more in depth information about that particular stage.

Stage Component

When designing this component it was important to offer the end user a full view of the entire pipeline they were viewing without cluttering the page with all the information at once. The Material UI Tree View component[44] was chosen because

it's great at displaying a hierarchical view of lists that the user can toggle to the depth they want.

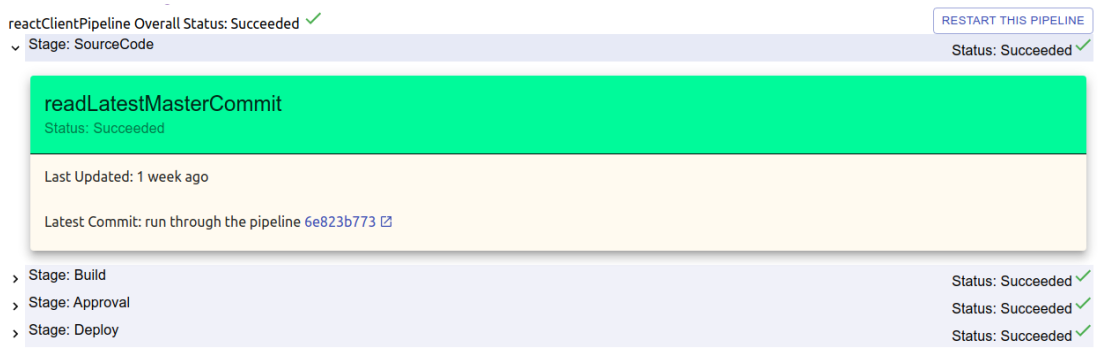

Figure 38: Dashboard Stage toggled down

Users can also retry the stage executuion again if it failed from this stage by clicking the RETRY STAGE button if available.

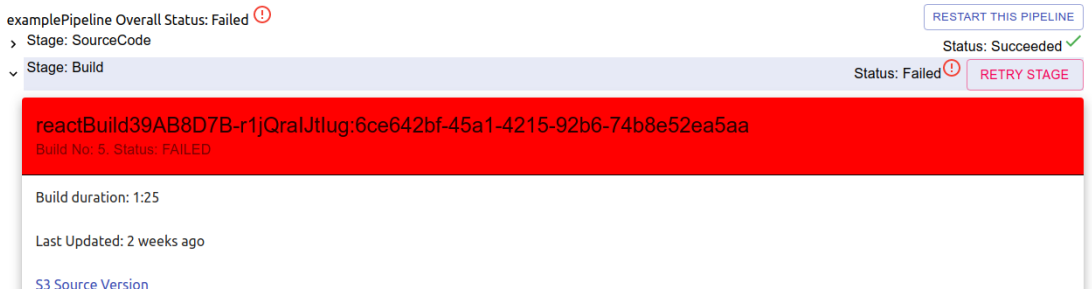

Figure 39: Dashboard Stage retry

Inside each Tree View Component, is a Tree Item which contains a List of Actions. Each Stage can contain multiple actions of different types.

Action Component

The four action types I'm supporting are Source, Build, Approval, and Deploy. To determine which action to render, I have used an <Action> component that renders a different Action depending on which category the action falls under.

The Source Action contains information about when the latest commit was added to the repo, when this was, and a link to the commit with the latest commit message to give the user some context.

The Deploy Action contains information about the last successful deployment, the last successful commit that made it to deployment, and the entity URL which is a link with more information about the action state.

Approval Action Component

The approval stage lets users approve or reject a potential deployment, whilst adding their comments as to why they accepted or rejected. Once it has been approved/rejected then the component changes from a text area with two buttons to approve/reject to displaying info about the latest commit, and comments that approved/rejected it.

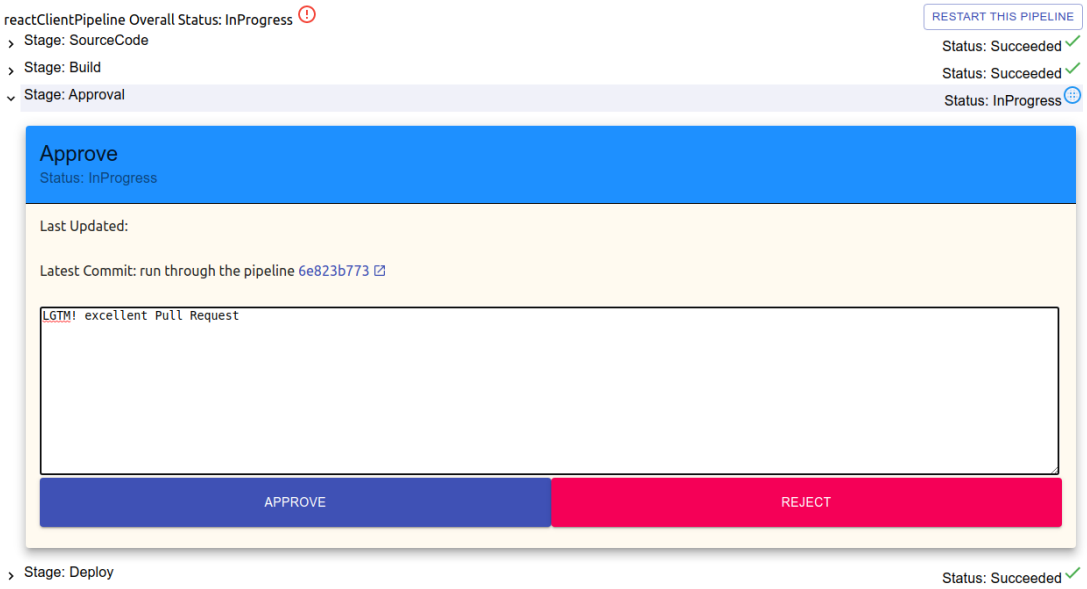

Figure 40: Dashboard approval in progress

To determine whether to show the text area + buttons or the post-approve/disapprove status comes down to whether the 'action.token' from 'getPipelineInfo' method in the CodePipelineService class. This AWS system-generated token is undefined if approval/rejection requests are not valid.

Build Action Component

The build action component is the most complex component in the dashboard, it required me to write a new service class for AWS CodeBuild and AWS CloudWatch.

```tsx
const BuildAction: React.FC<BuildActionProps> = ({ action, pipeline, stage }: BuildActionProps) => {
  const [config] = useState<CodeBuildClientConfig>(CONFIGURATION);
  const [builds, setBuilds] = useState<BuildModel[]>([]);

  const buildProjectId = action.buildProject!;

  useEffect(() => {
    const fetchData = async (): Promise<void> => {
      const codeBuildClient = new CodeBuildService(config);
      const CloudWatchLogsClient = new CloudWatchService(config);
      const buildIds = await codeBuildClient.listBuildsForProject(buildProjectId);
      let builds: BuildModel[] | undefined = await codeBuildClient.batchGetBuilds(buildIds);
      if (builds !== undefined) {
        let buildsWithLogs: BuildModel[] = await Promise.all(
          builds.map(async (build) => {
            return {
              ...build,
              logs: await CloudWatchLogsClient.getLogEvents(build.cloudWatch?.groupName, build.cloudWatch?.streamName)!,
            };
          })
        );

        setBuilds(buildsWithLogs);
      }
    };

    fetchData();

    // setInterval(fetchData, 30000);
  }, [config, buildProjectId]);

  if (builds.length > 0 && builds !== undefined) {
    return (
      <div>
        {builds.map((build) => (
          <BuildListComponent build={build} action={action} />
        ))}
      </div>
    );
  } else {
    return <div>{'Loading...'}</div>;
  }
};
```

Figure 41: Dashboard Build Action Code

The useEffect Function runs every render, or when the config or buildProjectId changes. Inside the useEffect I instantiate the CodeBuild and CloudWatch clients to gather all the information required about a specific pipeline build and pass it to my builds array of type BuildModel.

It then renders the BuildListComponent Component if the builds array is valid. This component then displays a list of every build run for this pipeline.

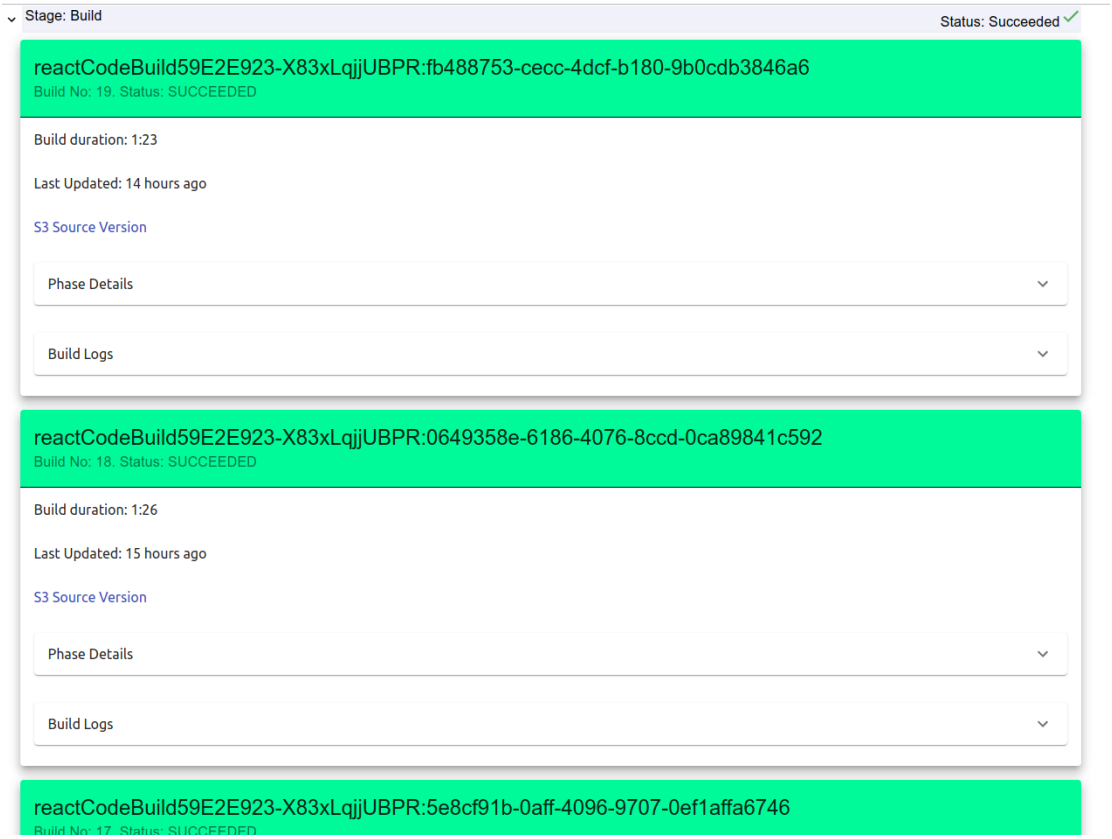

Figure 42: Dashboard build list view

BuildListComponent is a Card component that contains basic information about the build duration and when it was last updated, as well as more detailed components that specify the status and duration of each build phase in the PhaseDetail Component and the BuildLogs Component that gives the CloudWatch Logs for that build.

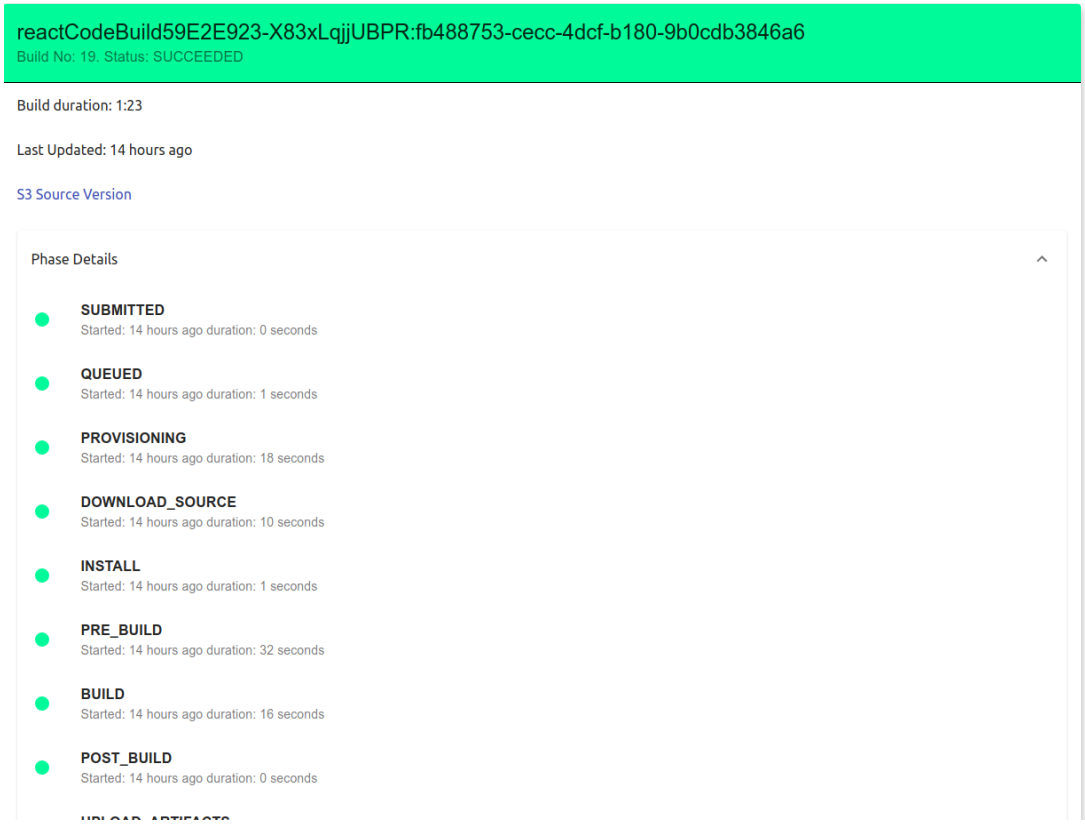

Figure 43: Dashboard build phase details

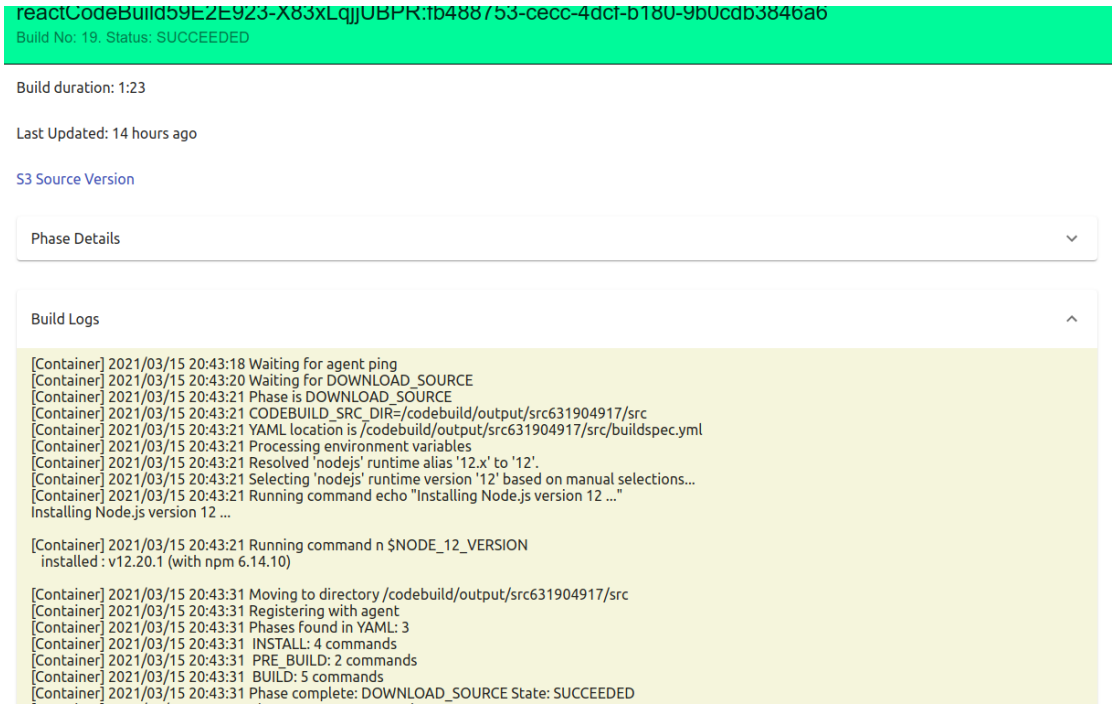

Figure 44: Dashboard build phase build logs

Dashboard Services and Models

To interact with the AWS SDK, I initialized clients from the AWS SDK for each respective service in their own services class. Thanks to the online documentation it was easy to find out the request parameters and response structure of each API call. This made it easier to construct my own models for the data I wanted to display in the dashboard. Every API call followed a similar structure of $_Command, $_CommandInput, $_CommandOutput. Typescript's type-checking capabilites really shine when working with well documented API's like AWS and allow for quick iteration and debugging of your code.

```
export interface GetPipelineInput {
    /**
     * <p>The name of the pipeline for which you want to get information. Pipeline names must
     *             be unique under an AWS user account.</p>
     */
    name: string | undefined;
    /**
     * <p>The version number of the pipeline. If you do not specify a version, defaults to
     *             the current version.</p>
     */
    version?: number;
}
```

Figure 45: Example of well documented AWS Models

CodePipelineServices & Models

The CodePipelineService class takes one argument in it's constructor: CodePipelineClientConfig which is an object containing AWS Credentials and the region you're creating the client in. Inside this class I've created 8 methods: listPipelines, getPipelineInfo, getPipelineExecutionInfo, getPipelineExecution, getPipelineState, startPipelineExecution, retryStageExecution, and putApprovalResult. These methods make requests to AWS, structure the response in models if needed, and return this response to the dashboard UI.

```
export class CodePipelineService {
  private client: CodePipelineClient;

  constructor(configuration: CodePipelineClientConfig) {
    this.client = new CodePipelineClient(configuration);
  }

  public async listPipelines(): Promise<PipelineModel[] | undefined> {
    try {
      const results: ListPipelinesCommandOutput = await this.client.send(new ListPipelinesCommand({}));
      if (results.pipelines !== undefined) {
        const pipelineList: PipelineModel[] = results.pipelines.map((pipeline) => ({
          pipelineName: pipeline.name as string,
          updated: pipeline.updated as Date,
          created: pipeline.created as Date,
          stages: [],
        }));
        return pipelineList;
      }
    } catch (error) {
      const { requestId, cfId, extendedRequestId } = error.$metadata;
      console.error(error);
      console.log({ requestId, cfId, extendedRequestId });
      return undefined;
    }
  }
```

Figure 46: CodePipelineService Snippet

The models I created to structure the data are outlined in the image below:

```
export interface ActionModel {
  actionName: string;
  category: string;
  actionId?: string;
  repo?: string;
  branch?: string;
  status?: string;
  buildProject?: string;
  lastUpdated?: Date;
  entityUrl?: string;
  token?: string;
  summary?: string;
}

export interface StageModel {
  stageName: string;
  status?: string;
  actions: ActionModel[];
}

export interface SourceRevisionModel {
  actionName: string | undefined;
  revisionSummary?: string;
  revisionUrl?: string;
  revisionId?: string;
}

export interface PipelineExecutionSummaryModel {
  lastUpdateTime: Date;
  sourceRevisions?: SourceRevisionModel[];
  startTime: Date;
  status: string;
  stopTrigger?: string;
}

export interface PipelineModel {
  pipelineName: string;
  created?: Date;
  updated?: Date;
  status?: string;
  pipelineExecutionId?: string;
  pipelineExecutionSummary?: PipelineExecutionSummaryModel[];
  stages: StageModel[];
}
```

Figure 47: CodePipelineModels

The PipelineModel is the main model that all the data I request lives within, and inside this model their are various other models that keep track of info needed for other requests or just to display.

The request itself happens at the root of the dashboard in the <App/> Component, I initialize a CodePipelineService client and then make the api call on app render, and then every 20 seconds to update the dashboard with new information.

```
const [config] = useState<CodePipelineClientConfig>(CONFIGURATION);

const codePipelineClient = new CodePipelineService(config);

useEffect(() => {
  const fetchData = async (): Promise<void> => {
    const currentPipelines = await codePipelineClient.listPipelines();

    if (currentPipelines !== undefined) {
      const getCurrentPipelinesInfo = async () => {
        return Promise.all(
          currentPipelines.map((pipeline) => codePipelineClient.getPipelineInfo(pipeline.pipelineName))
        );
      };
      const pipelinesFullDetail = await getCurrentPipelinesInfo();
      if (pipelinesFullDetail !== undefined) {
        let notUndefined: PipelineModel[] = pipelinesFullDetail as PipelineModel[];
        if (notUndefined.every((pipeline) => pipeline !== undefined)) {
          setPipelines(notUndefined);
        }
      }
    }
  };
  fetchData();
  setInterval(fetchData, 20000);
```

Figure 48: App API Call

A lot of the error checking above for testing if it's undefined is one of the helpful features of Typescript, if this dashboard was developed in Javascript I would not have been aware of the various states my data could be in but with Typescript it's able to tell if there is a potential for a variable to be undefined and makes sure that is accounted for. This helps ensure the program doesn't crash unexpectedly and helped to speed up development a lot.

CodeBuildServices & Models

The CodeBuildService class is needed specifically for the build stage of each pipeline, the class is structured similarly to the CodePipelineService class, except it only has 2 methods: batchGetBuilds and listBuildsForProject. The Models in CodeBuildModels.ts also contain the models used for the CloudWatchService class since all CloudWatchService calls and activity take place within the Build stage.

```
export interface LogModel {
  message: string;
  timestamp: string;
}

export interface CloudWatchModel {
  cloudWatchLogsArn: string;
  deepLink: string;
  groupName: string;
  streamName: string;
}

export interface PhaseModel {
  contextMessage?: string;
  contextStatusCode?: string;
  phaseStatus: string;
  phaseType: string;
  durationinSeconds: number;
  startTime: Date;
  endTime: Date;
}

export interface BuildModel {
  buildRun: string;
  buildStatus: string;
  buildNumber: number;
  sourceVersion: string;
  duration: number;
  completed: boolean;
  phases: PhaseModel[];
  cloudWatch?: CloudWatchModel;
  logs?: LogModel[];
  endTime: Date;
}
```

Figure 49: CodeBuildModels

The client is initialized in the <BuildAction/> component and follows a similar structure to the codePipelineClient above.

CloudWatchService & Models

The models for the CloudWatchService are intertwined with the CodeBuildService so they live in the CodeBuildModels file. The models are the LogModel and the CloudWatchModel.

The class itself contains one method, getLogEvents which is used to parse the CodeBuild logs and return them to the dashboard.

```
export class CloudWatchService {
  public client: CloudWatchLogsClient;

  constructor(configuration: CloudWatchLogsClientConfig) {
    this.client = new CloudWatchLogsClient(configuration);
  }

  public async getLogEvents(
    logGroupName: string | undefined,
    logStreamName: string | undefined
  ): Promise<LogModel[] | undefined> {
    try {
      const results: GetLogEventsCommandOutput = await this.client.send(
        new GetLogEventsCommand({ logGroupName, logStreamName })
      );

      if (results.events !== undefined) {
        let parsed: LogModel[] = results.events.map((log) => {
          if (log.timestamp !== undefined && log.message !== undefined) {
            return {
              timestamp: new Date(log.timestamp!).toISOString(),
              message: log.message,
            } as LogModel;
          } else {
            return {
              timestamp: '',
              message: '',
            };
          }
        });
        return parsed;
      }
    } catch (error) {}
  }
}
```

Figure 50: CloudWatchService class

This is then called inside <BuildAction>, where we map the logs to the appropriate build.

```
const BuildAction: React.FC<BuildActionProps> = ({ action, pipeline, stage }: BuildActionProps) => {
  const [config] = useState<CodeBuildClientConfig>(CONFIGURATION);
  const [builds, setBuilds] = useState<BuildModel[]>([]);

  const buildProjectId = action.buildProject!;

  useEffect(() => {
    const fetchData = async (): Promise<void> => {
      const codeBuildClient = new CodeBuildService(config);
      const CloudWatchLogsClient = new CloudWatchService(config);
      const buildIds = await codeBuildClient.listBuildsForProject(buildProjectId);
      let builds: BuildModel[] | undefined = await codeBuildClient.batchGetBuilds(buildIds);
      if (builds !== undefined) {
        let buildsWithLogs: BuildModel[] = await Promise.all(
          builds.map(async (build) => {
            return {
              ...build,
              logs: await CloudWatchLogsClient.getLogEvents(build.cloudWatch?.groupName, build.cloudWatch?.streamName)!,
            };
          })
        );

        setBuilds(buildsWithLogs);
      }
    };

    fetchData();

    setInterval(fetchData, 20000);
  }, [config, buildProjectId]);
```

Figure 51: BuildAction with both CloudWatch and CodeBuild in use

4.4 Chatbot

The chatbot can be broken down into two seperate systems and architectures.

The first system architecture is a Publish/Subscribe(Pub/Sub) messaging architecture.

AWS Descibes Pub/Sub as: A form of asynchronous service-to-service communication used in serverless and microservices architectures.

In a pub/sub model, any message published to a topic is immediately received by all of the subscribers to the topic.

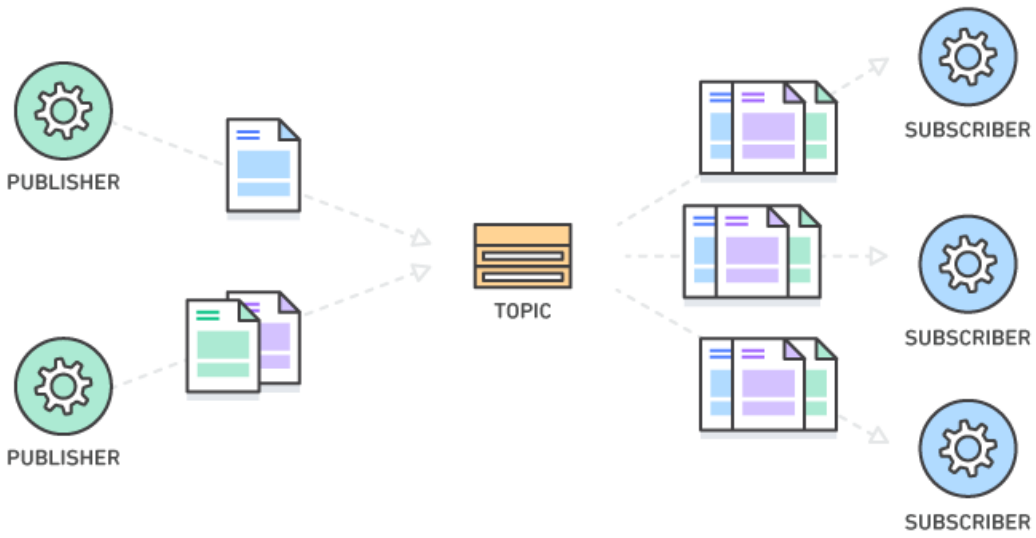

Figure 52: Pub Sub Architecture[45]

The second system architecture is one where the client/end-user(developer in slack channel) sends off a chat message, that triggers the bot in the channel to make a HTTP POST request to a URL you specified on the slack website. This URL will be a Lambda that has been integrated with an AWS API Gateway.

The reason we need two seperate architectures is so users can interact with the chatbot of their own accord, but also they are notified immediately when any important event happens or they're required to intervene in the process.

Pub/Sub Notifications

In my original prototype I developed a simple Pub/Sub system that triggered a python Lambda whenever the SNS topic was notified of a pipeline event.

```
export class PipelineEvent extends Construct {
  constructor(scope: Construct, id: string, props: PipelineEventProps) {
    super(scope, id);

    const eventPipeline = EventField.fromPath("$.detail.pipeline");
    const eventState = EventField.fromPath("$.detail.state");
    const eventFull = EventField.fromPath("$.detail");

    props.pipeline.onStateChange("OnPipelineStateChange", {
      target: new SnsTopic(props.topic, {
        message: RuleTargetInput.fromText(
          `Pipeline ${eventPipeline} changed state to ${eventState}, eventFull ${eventFull}`
        ),
      }),
    });

    const target: CfnNotificationRule.TargetProperty = {
      targetType: "SNS",
      targetAddress: props.topic.topicArn,
    };

    new CfnNotificationRule(this, "statusChangeNotificationRule", {
      detailType: "FULL",
      eventTypeIds: [
        "codepipeline-pipeline-action-execution-succeeded",
        "codepipeline-pipeline-action-execution-failed",
        "codepipeline-pipeline-stage-execution-started",
        "codepipeline-pipeline-pipeline-execution-started",
      ],
      name: "notificationRuleName",
      resource: props.pipeline.pipelineArn,
      targets: [target],
    });

    let fn = new Function(this, "SODemoFunction", {
      runtime: Runtime.NODEJS_10_X,
      handler: "slack-alert.handler",
      code: Code.fromAsset(path.join(__dirname, "functions")),
    });

    props.topic.addSubscription(new LambdaSubscription(fn));
  }
}
```

Figure 53: PipelineEvent Construct

```
    const topic = new Topic(this, "pipelineEventTopic");

    const pipelineEvent = new PipelineEvent(this, "PipelineNotificationEvent", {
      pipeline: pipeline,
      topic: topic,
    });
```

Figure 54: PipelineEvent Attached to Pipeline

Since I'm using CDK for my infrastructure, I made a construct that could be attached to any event called PipelineEvent, this works the same way as the GUI method but it has been CDK-ified to work with my new infrastructure. This construct could then be attached to any pipeline.

External CDK Slack Approval and Notification Module

The great thing about the AWS CDK is how modular it is and how easy it is to share

and reuse tried and tested patterns from online. I discovered a 3rd party node module called @cloudcomponents/cdk-codepipeline-slack[46] that provides a Cdk component that provisions a slack approval workflow and notification messages on codepipeline state changes. It follows the same architecture as my original PipelineEvent but is more polished and well tested.

I decided to use this instead of my own custom Pub/Sub notification and approval system. To integrate it with my CDK pipeline I installed the package, gave it the correct configuration on the slack website and in the code and it worked flawlessly.

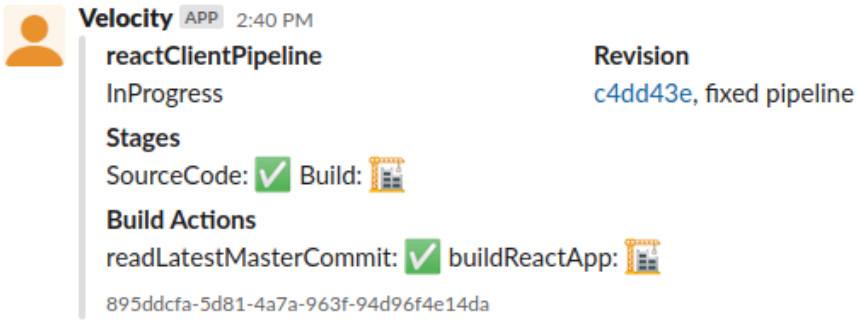

Figure 55: Cloud Component cdk pipeline notification

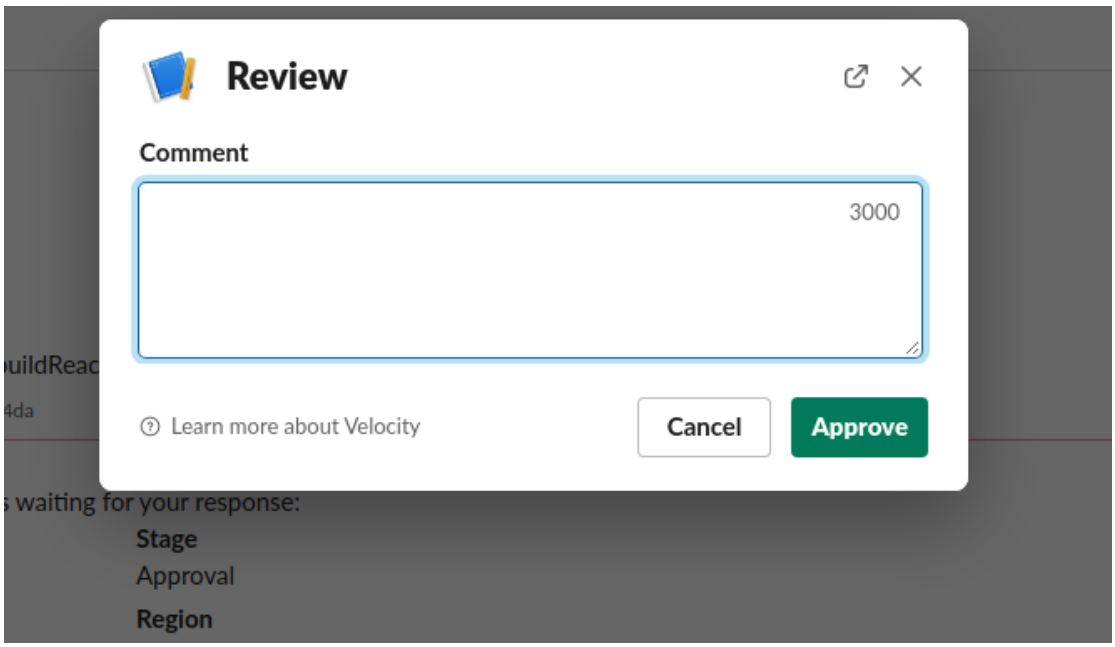

Figure 56: Cloud Component cdk pipeline approval

Slack Slash Commands

The second architecture type is one where the end-user can enter slash commands (/command [parameter]) and receive information back from the bot which runs a lambda and returns the results.

Slash Command Architecture

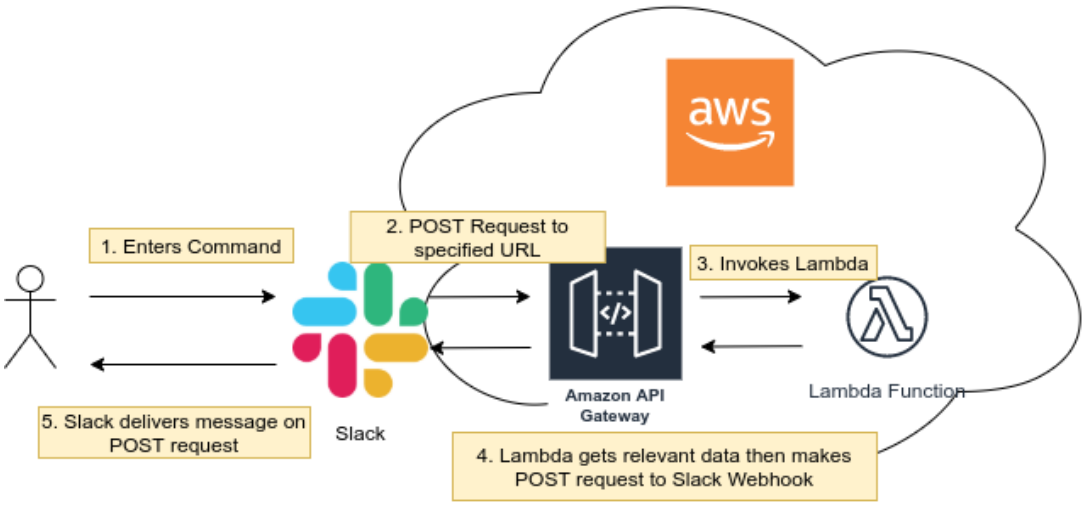

Figure 57: Slash Command Architecture

Adding Commands to Slack

After creating a bot, you can add slash commands throught the website by navigating to the slash commands section and entering the required information.

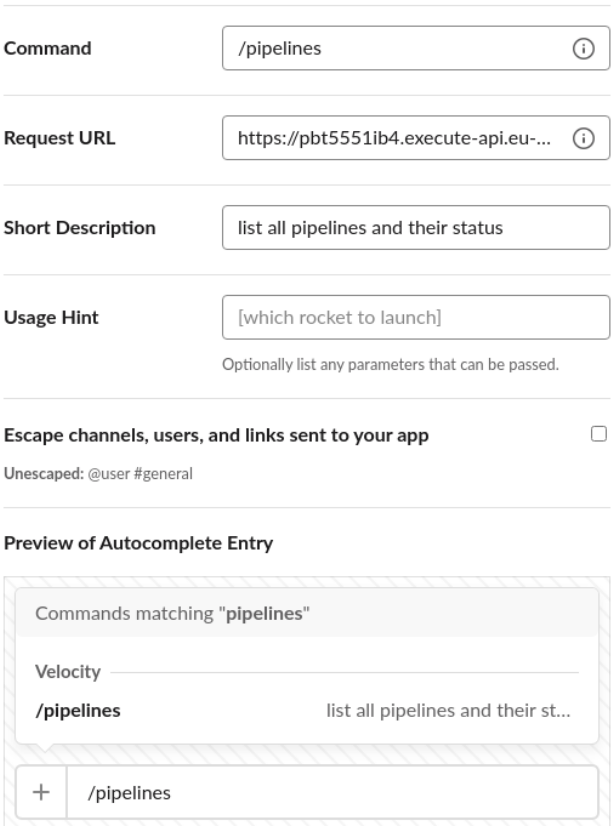

Figure 58: Adding new slash commands

To have feature parity with the dashboard, I created the following slash commands:

- **/pipelines**: lists all pipelines and their status
- **/pipeline-info [pipelineName]**: info about the specified [pipelineName] pipeline state.
- **/stage-info [pipelineName] [stageName]**: info about the specified [stageName] stage in the [pipelineName] pipeline
- **/restart-pipeline [pipelineName]**: restarts the specified [pipelineName] pipeline.
- **/restart-stage [pipelineName] [stageName]**: restarts the specified [stageName] stage in the [pipelineName] pipeline if possible.

CDK Slash Command Creation

I created a 'slack_bot_service_stack' CDK Stack that deployed all the lambda's and API gateways with one command, 'cdk deploy slack-bot-service'.

This single stack contained the 5 seperate lambda/API gateway constructs.

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

    new pipeline_service.PipelineService(this, "PipelineService");
    new pipeline_info_service.PipelineInfoService(this, "PipelineInfoService");
    new pipeline_restart_service.PipelineRestartService(
      this,
      "PipelineRestartService"
    );
    new stage_info_service.StageInfoService(this, "StageInfoService");
    new stage_restart_service.StageRestartService(this, "StageRestartService");
  }
}
```

Figure 59: Slack Bot Service Stack

```
// 4. deploys the api gateways + lambdas for the chatbot to function
new SlackBotServiceStack(app, "slack-bot-service", { env: dublin });
```

Figure 60: Deploying SlackBotServiceStack in CDK

To create the lambda for each slash command I had to write a function that was capable of parsing the request, make a request to the relevant services, and make another request with the information to slack and ensure the permissions were correct.

I had trouble writing these functions originally because I didn't read the slash command documentation[48] all the way through initially. To perform a sucessful

slash command you have to let the application know that you've successfully received the command right away via a callback or you'll get errors around timeouts.

After Figuring this out it was just a case of ensuring the async code worked correctly and the lambda gave the correct console.logs in cloudwatch along the way before POSTing the finished message back to slack.

```
const pipelineExecutionId =
  data.stageStates[0].latestExecution.pipelineExecutionId;

console.log("# pipelineExecutionId", pipelineExecutionId);

let retryStageParams = {
  pipelineExecutionId: pipelineExecutionId,
  pipelineName: pipelineName,
  stageName: stageName,
  retryMode: "FAILED_ACTIONS",
};

console.log("# retryStageParams", retryStageParams);

codePipeline
  .retryStageExecution(retryStageParams)
  .promise()
  .then((resp) => {
    var postData = {
      channel: "#final-year-project",
      username: "Velocity",
      response_type: "in_channel",
      text: `Successfully restarted ${pipelineName}'s ${stageName} stage`,
    };

    var options = {
      method: "POST",
      hostname: "hooks.slack.com",
      port: 443,
      path:
        "/services/T01G1B6T4P4/B01FLJYP90X/k3L1yDga7npDpBwcWmjXsVhe",
    };

    var req = https.request(options, function (res) {
      res.setEncoding("utf8");
      res.on("data", function (chunk) {
        context.done(null);
      });
    });

    req.on("error", function (e) {
      console.log("problem with request: " + e.message);
    });

    req.write(util.format("%j", postData));
    req.end();
```

Figure 61: Stage Restart Lambda 2/2

Once a Lambda was written, it was then added to it's own service construct e.g. stage_restart_service.ts which creates the lambda, tells CDK where to find it, adds the neccessary permissions and then initializes an API gateway and adds it to it.

```
import * as core from "@aws-cdk/core";
import * as apigateway from "@aws-cdk/aws-apigateway";
import * as lambda from "@aws-cdk/aws-lambda";
import * as iam from "@aws-cdk/aws-iam";

export class StageRestartService extends core.Construct {
  constructor(scope: core.Construct, id: string) {
    super(scope, id);

    const handler = new lambda.Function(this, "StageRestartHandler", {
      runtime: lambda.Runtime.NODEJS_10_X,
      code: lambda.Code.fromAsset("slack-bot-lambdas"),
      handler: "stage-restart.handler",
    });

    const statement = new iam.PolicyStatement();
    statement.addResources(
      "arn:aws:logs:*:*:*",
      "arn:aws:codepipeline:eu-west-1:*"
    );
    statement.addActions(
      "codepipeline:ListPipelines",
      "codepipeline:StartPipelineExecution",
      "codepipeline:GetPipelineState",
      "codepipeline:RetryStageExecution"
    );

    handler.addToRolePolicy(statement);

    const api = new apigateway.RestApi(this, "tage-restart-api", {
      restApiName: "Stage Restart Service",
      description: "This service restarts a specific Stage.",
    });

    const stageRestartIntegration = new apigateway.LambdaIntegration(handler);

    api.root.addMethod("POST", stageRestartIntegration);
  }
}
```

Figure 63: Stage Restart Service Construct

So in total there were 5 Lambdas, 5 Api Gateway/Lambda constructs, and 1 SlackBotServiceStack Stack which initialized the 5 constructs.

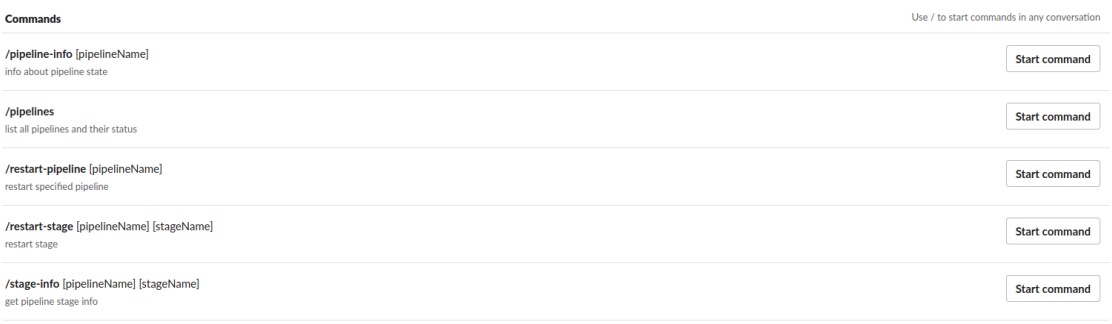

Figure 64: Slash Commands Overview

an example output of /pipeline-info [pipelineName];

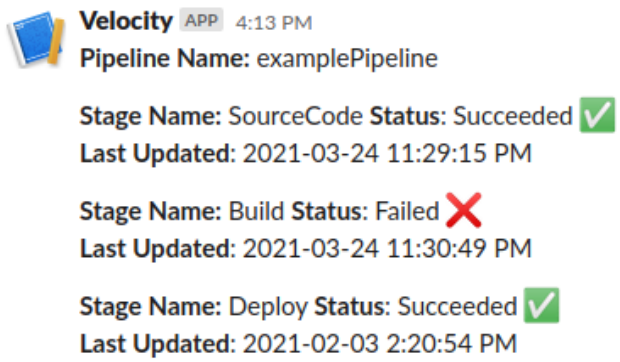

Figure 65: Example Slash Command output

4.5 Missing Features

Due to the change in architecture and technology choices from the design chapter, the redeploy feature was undeliverable because CodePipeline doesn't support redeploying past builds, only CodeDeploy does which is not in my architecture.[47]

Similarly, the stop build feature didn’t seem worthwhile in the pipeline architecture I built so I left it out, users can just restart the pipeline and get to the build stage nearly instantly too.

5. Testing and Evaluation

5.1 Introduction

This chapter will detail the types of testing used and not used while developing this project. As well as testing, system evaluation was carried out to find out what potential users of the both systems thought about the them and if they were missing

any features. I also evaluated the system against the heuristic evaluation matrix I devised in section 2.2.

5.2 Testing

Black Box Testing

Black Box Testing is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. [49]

The following tables are the results of the black box testing carried out within this project.

Test No.	**Input**	**Expected**	**Actual**	**Pass/ Fail**
1.	Push commit to repository enabled for slack notification	Display information about current pipeline status updates with chatbot as they happen	Information is displayed about current pipeline status updates as they happen in slack	Pass
2.	Enter the slash command /pipeline-info examplePipline	Display information on request about specific pipeline via chatbot	Information is displayed about examplePipeline in the slack channel	Pass
3.	Enter the slash command /stage-info examplePipeline Build	display information on request about specific stage of specific pipeline via chatbot	Information is displayed about examplePipeline's Build stage in the slack channel	Pass
4.	Build with approval stage prompts user for approval/rejection, approve	approve build via chatbot button and modal	Build is approved via button and modal and progresses on	Pass

5.	Build with approval stage prompts user for approval/rejection, reject	reject build via chatbot button and modal	Build is rejected via button and modal and does not progresses on	Pass
6.	Enter the slash command /pipeline-restart examplePipeline	restart pipeline via chatbot	ExamplePipeline is restarted via chatbot	Pass
7.	Naviagate to dashboard homepage route: /	Display overview of all pipeline statuses on dashboard homepage	Overview of all pipeline statuses is displayed	Pass
8.	Push commit to pipeline on homepage	Display information about pipeline status updates with dashboard without refreshing	Status information changes without refreshing the page	Pass
9.	Navigate to the examplePipeline URL: /examplePipeline	Display detailed pipeline information via dashboard UI on naviagtion to correct pipline's URL route.	Detailed information is displayed about each stage of examplePipeline	Pass
10.	Build with approval stage prompts user for approval/rejection, approve	approve builds via dashboard	Build is approved and passes on to next stage	Pass
11.	Build with approval stage prompts user for approval/rejection, reject	reject builds via dashboard	Build is rejcted and passes on to next stage	Pass
12.	Click restart this pipeline button on examplePipeline	restart pipelines via dashboard	Pipeline is restarted and goes through all stages again	Pass

13.	Click Retry Stage on examplePipeline’s Build Stage	restart specific pipeline stages via dashboard	Build Stage is Restarted	Pass

Unit & Integration Testing

I planned to have a lot of unit and integration testing for the chatbot & dashboard. When it came to implementation I had issues with coming up with a good solution to actually doing this since my projects rely so heavily on external dependencies.

I tried mocking and stubbing various services and functionality but I found it too time consuming.I did not think it delivered any value to the project as it was being developed by one person, and the attempts I made to mock weren't giving me confidence that the tests actually ensured the functionality of the programs. Below are images of some different failed mocking solutions I came up with.

Figure 66: Mocking attempt in dashboard #1

```
import { CloudWatchService } from './CloudWatchService';
import { mock, when, anyString, instance } from 'ts-mockito';

const MockedCloudWatchService = mock(CloudWatchService);

describe('CloudWatchService', () => {
  it('testing', async () => {
    let mockInstance = instance(MockedCloudWatchService);

    when(MockedCloudWatchService.getLogEvents(anyString(), anyString())).thenResolve([
      {
        timestamp: '1549312452',
        message: 'hello',
      },
    ]);

    const actual = await mockInstance.getLogEvents('logGroupName', 'logStreamName');

    expect(actual).toEqual([
      {
        timestamp: '1549312452',
        message: 'hello',
      },
    ]);
  });
});
```

Figure 67: Mocking attempt in dashboard #2

5.3 Evaluation

Heuristic Evaluation

Back in section 2.2 I devised a Heuristic Evaluation Framework for comparing existing solutions and gave them points in each of these areas:

• Facilitation of Communication & Collaboration

• Visibility into the system, build pipeline & dashboards

• Scalability and ease of implementation

• Developer Experience(DX): UI, ease of use, aesthetic design.

These 4 areas were deemed the most important from the literature review I conducted into what makes CI/CD implementations successfully in software organizations.

The scoring system remains the same:

- 1-3: Does not adhere to the heuristic at all

- 4-6: Provides a decent implementation of the heuristic
- 7-10: Respects the heuristic almost perfectly with only minor or no problems

CloudFirst Heuristic Evaluation

CloudFirst	**Reasoning**	**Score**
Communication & Collaboration	CloudFirst Really prioritized centralizing communication and collaboration with the chatbot and dashboard. The chatbot offers real time updates into pipeline notifications for all channel members in slack, and also allows you to prompt the chatbot for information or to restart pipelines/stages as well. The dashboard with it's clean URL routing implementation allows members to bookmark and share pipelines easily.	8
Visibility & Dashboards	CloudFirst provides a great basic dashboard that allows users to have a quick view into any pipeline and it's stages they wish as well as a good birdseye overview. Compared to existing solutions it lacks fully fleshed out features like historical pipeline info,and fancy graphs. I feel like this is an area that could be improved in the future as more features could be added at the request of it's users.	5
Scalability & Implementation	if you use AWS CodePipeline and the implemented stages & actions it is easily implementable, just pass in your configs and parameters in the right places and away you go. Unfortunately, CodePipeline alone has 30 valid action providers broken into 6 categories so it's hard to build a one solution fits all product. However the groundwork is laid for it to support any action or stage you require. It scales well since it utilizes Lambda for the chatbot, and Cloudfront CDN/S3 for the dashboard hosting.	6
Developer Experience	It's integrated with existing developer workflows very well, if you're used to using slack then the chatbot is very handy for keeping an eye on all the notifications from your pipeline	7
Overall Score		26/40

User Evaluation

User Evaluation was carried out over video calls via Google Meet. Since my product is designed to be used for existing software engineers I focused my efforts on getting former colleagues to use and give feedback instead of non-technical users or other students.

The overall feedback from the interviews was very positive. The chatbot was seen as a really great way to increase communication and collaboration across an organization. The dashboard was liked for how simple and clean the UI was, and how easy it was to navigate to specific pipelines/stages

The constructive feedback I received was:

- The use of slash commands leads to anonymity when the commands are executed, is there a way to show who executed the commands via the bots response?
- There isn't a way to limit the chatbot responses to particular people by direct messaging them or highlighting them only so everyone isn't spammed with messages
- There isn't there a way to enable text notifications for urgent failures/notifications
- The Dashboard has no Historical graphs, visualizations etc. which are nice for seeing daily/weekly/monthly trends

System Demonstration

https://youtu.be/sjVdAx_9VWo

Youtube link of demonstration of fixing a failing test in the pipeline and watching it go through the various stages in the chatbot and dashboard. Chatbot commands to get information about pipelines,restart pipelines and stages at the end.

6. Conclusion

6.1 Chapter Conclusions

In this section I'll present some of my key takeaways from each chapter.

Literature Review

- Communication & Collaboration is vital in modern software development teams, Too many organizations neglect this aspect and think all their CI/CD problems can be solved with tools that do not enable this.
- Dashboards have to be easily sharable across the whole team, update in real time and help teams diagnose and fix issues quickly.
- If an organization plans on trying to improve their CI/CD pipelines it's important that they look at what their current workflows look like and to try to

work with it and gradually improve it over time rather than sweeping changes or bringing in new systems that their developers aren't familar with.

Design

- Redeploying past builds from the dashboard and chatbot were not possible with the pipeline I designed in this chapter or with the one I eventually implemented in the Development chapter. the action of redeploying in AWS is specific to AWS CodeDeploy inside AWS CodePipeline.

- The Techincal architecture was not as intertwined as drawn, the chatbot and dashboard both function as two seperate systems that can interact with pipelines seperately.

Development

- The infrastructure development was a big time sink early on, I wasted a lot of time deciding on writing CloudFormation templates by hand until I discovered the CDK which lets you abstract that and all the permissions issues away to a higher level language

- Python and it's packaging system also was frustrating to use early on since I'm a novice python user, the switch to Typescript was a great success and really helped get into the swing of things with the entire project since it all became Typescript/Javascript so less context switching between languages.

- AWS' CDK and SDK are both a joy to work with in Typescript, it's very easy to find out what parameters you have to pass, what you'll receive and the error messages are usually easy to read and fix the problem you are having.

- Slack's error messaging/API on the other hand left me a little dissapointed, they were very vague with what was going wrong when trying to send/recieve requests and took me longer than expected to fix issues I was having setting up the Lambda/Slack webhook connection.

Testing & Evaluation

- Writing meaningful unit & integration for a project like this with lots of external dependencies is hard, I definitely overestimated how hard it would be write these and did not find lots of good information online specific to my usecase of mocking CodePipeline and relevant services.

- My heuristic evaluation framework was something I devised based on what I researched in the literature review chapter on what makes a good CI/CD

system. I think if I could redo it I would get more user feedback prior to development to see what others thought of the categories.

- User Evaluation was hindered by COVID, I only managed to interview 2 engineers I had worked with previously over a video call to let them user my system and give me feedback. If I could redo this I would involve them, and others earlier in the development process to improve things as I went along.

6.2 Future Work

I think there's loads of areas to expand into for this project, if I could redo it over I think it would really benefit from working closely with a company and trying to improve their existing CI/CD pipelines with a dashboard or chatbot solution.

For the dashboard I think there's a lot of room for improvement by supporting multiple regions, multiple envrionments, historical dashboard information and graph solutions.

For the chatbot I think adding support for DMing specific members about pipeline issues instead of broadcasting to a channel would be a good addition, as well as displaying information about how initiated each command and storing this information somewhere to help find out what caused different issues.

Developing machine learning solutiuons that could learn from past build failures and try diagnose why they failed and how to fix them in the future.

6.3 Project Plan

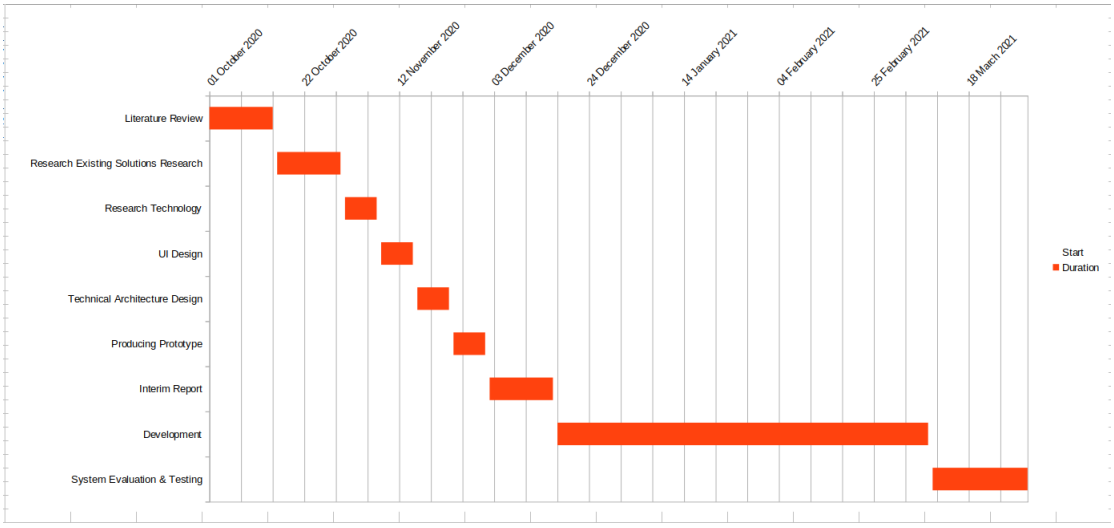

Figure 68: Gantt Chart from Proposal

The development started the week of the 14 of January because of Exams/Christmas Holidays but since we had extensions out to the 31st and then 6th of April for submission this didn't affect the project timeline that much.

If I had to do it all over again, I would incorporate more user evaluation early on since I think it really helps when developing things to have short feedback loops for the work you're doing to make sure it's as good as possible. By leaving it to the last few weeks it's hard to implement other features users ask for and leads to a worse product.

6.4 Final Reflection

This project was born of my internships working as both a Quality Engineer(Testing Software) & Frontend Engineer. I Have seen firsthand the issues and frustrations of large development teams when it comes to modern chat software and build pipelines that are clogged with multiple people waiting to push their changes through it in slow manual processes.

It was very enjoyable to research how other organizations tackle their CI/CD solutions and design and develop my own solution passed on the criteria I came up with for what makes a successful CI/CD product.

My least favourite part of the project was not being able to interact more with my classmates and peers in computer science in person while developing our projects. From previous years you could see the camaraderie that was built amongst the final year students as they coded their projects side by side.

My favourite part was definitely using all the different AWS products and technologies. A lot of the technologies I used didn't exist 5 or 10 years ago, some like the AWS Cloud Development Kit were not released until July 2019 and I think the paradigm of creating being able to create object-oriented infrastructure in modern languages will completely change how infrastructure is developed, deployed, and maintained.

Bibliography

1. Li, J., Neely, K. and Geiger, J., 2020. Software Release Culture At Shopify. [online] Shopify. Available at: <https://engineering.shopify.com/blogs/engineering/software-release-culture-shopify> [Accessed 13 October 2020].

2. Ananthanarayanan, S., Ardekani, M., Haenikel, D., Varadarajan, B., Soriano, S., Patel, D. and Adl-Tabatabai, A., 2019. Keeping Master Green at Scale. Proceedings of the Fourteenth EuroSys Conference 2019 CD-ROM on ZZZ - EuroSys '19,.

3. Thomakos, P., 2020. Butler Merge Queue — How Strava Merges Code. [online] Medium. Available at: <https://medium.com/strava-engineering/butler-merge-queue-how-strava-merges-code-7095a3310930> [Accessed 1 December 2020].

4. Dickson, J. and Wongkampoo, S., 2020. Tech @ Agoda #7: Bots (Merge Queue). [video] Available at: <https://www.youtube.com/watch?v=cj1fH_Lk7Fs> [Accessed 1 December 2020].

5. Humble J, Farley D. Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education; 2010 Jul 27.

6. Hilton M, Tunnell T, Huang K, Marinov D, Dig D. Usage, costs, and benefits of continuous integration in open-source projects. In2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE) 2016 Sep 3 (pp. 426-437). IEEE.

7. Continuousdelivery.com. 2020. What Is Continuous Delivery? - Continuous Delivery. [online] Available at: <https://continuousdelivery.com/> [Accessed 1 December 2020].

8. Kärpänoja P, Virtanen A, Lehtonen T, Mikkonen T. Exploring peopleware in continuous delivery. InProceedings of the scientific workshop proceedings of xp2016 2016 May 24 (pp. 1-5).

9. Shahin M, Zahedi M, Babar MA, Zhu L. Adopting continuous delivery and deployment: Impacts on team structures, collaboration and responsibilities. InProceedings of the 21st international conference on evaluation and assessment in software engineering 2017 Jun 15 (pp. 384-393).

10. Shahin M, Babar MA, Zhu L. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access. 2017 Mar 22;5:3909-43.

11. Chen L. Continuous delivery: Overcoming adoption challenges. Journal of Systems and Software. 2017 Jun 1;128:72-86.

12. Claps GG, Svensson RB, Aurum A. On the journey to continuous deployment: Technical and social challenges along the way. Information and Software technology. 2015 Jan 1;57:21-31.

13. 2020. Shipit - Https://Github.Com/Shopify/Shipit-Engine. Shopify.

14. Boussier, J., 2020. Introducing Shipit. [online] Shopify. Available at: <https://shopify.engineering/introducing-shipit> [Accessed 1 December 2020].

15. Li, J., 2020. Successfully Merging The Work Of 1000+ Developers. [online] Shopify. Available at: <https://shopify.engineering/successfully-merging-work-1000-developers> [Accessed 1 December 2020].

16. Octopus Deploy. 2020. Deployment & Runbook Automation - Octopus Deploy. [online] Available at: <https://octopus.com/> [Accessed 1 December 2020].

17. Octopus Deploy. 2020. Features - Octopus Deploy. [online] Available at: <https://octopus.com/features> [Accessed 1 December 2020].

18. BrowserStack Blog. 2020. Top Continuous Integration (CI) Tools Comparison | Browserstack. [online] Available at: <https://www.browserstack.com/blog/best-ci-cd-tools-comparison/> [Accessed 1 December 2020].

19. Fowler, M., 2020. Continuous Integration. [online] martinfowler.com. Available at: <https://martinfowler.com/articles/continuousIntegration.html> [Accessed 1 December 2020].

20. monday.com Blog. 2020. Slack Vs Microsoft Teams | Monday.Com Blog. [online] Available at: <https://monday.com/blog/remote-work/slack-vs-microsoft-teams-which-is-the-best-chat-app/> [Accessed 1 December 2020].

21. Amazon Web Services, Inc. 2020. AWS Codecommit | Managed Source Control Service. [online] Available at: <https://aws.amazon.com/codecommit/> [Accessed 1 December 2020].

22. StackShare. 2020. AWS Codecommit Vs Github | What Are The Differences?. [online] Available at: <https://stackshare.io/stackups/aws-codecommit-vs-github> [Accessed 1 December 2020].

23. Slintel.com. 2020. Github Source Code Management Tool | Top Customers And Competitor Details 2020. [online] Available at: <https://www.slintel.com/tech/source-code-management/github-market-share> [Accessed 1 December 2020].

24. Amazon Web Services, Inc. 2020. AWS Chatbot - Amazon Web Services. [online] Available at: <https://aws.amazon.com/chatbot/> [Accessed 1 December 2020].

25. HUBOT. 2020. HUBOT. [online] Available at: <https://hubot.github.com/> [Accessed 1 December 2020].

26. Edureka. 2020. What Is Splunk? A Beginners Guide To Understanding Splunk | Edureka. [online] Available at: <https://www.edureka.co/blog/what-is-splunk/> [Accessed 1 December 2020].

27. Chen L. Continuous delivery: Huge benefits, but challenges too. IEEE Software. 2015 Jan 12;32(2):50-4.

28. SDLC - Agile Model - Tutorialspoint [Internet]. Tutorialspoint.com. 2020 [cited 9 December 2020]. Available from: https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm

29. 2. What is Agile Software Development? [Internet]. Visual-paradigm.com. 2020 [cited 9 December 2020]. Available from: https://www.visual-paradigm.com/scrum/what-is-agile-software-development/

30. 3. Scrum - Framework - Tutorialspoint [Internet]. Tutorialspoint.com. 2020 [cited 9 December 2020]. Available from: https://www.tutorialspoint.com/scrum/scrum_framework.htm

31. Troubleshooting AWS CloudFormation - IAM Permissions - AWS CloudFormation [Internet]. Docs.aws.amazon.com. 2021 [cited 31 March 2021]. Available from: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/troubleshooting.html#troubleshooting-errors-insufficient-iam-permissions

32. Troubleshooting AWS CloudFormation - Dependency Errors - AWS CloudFormation [Internet]. Docs.aws.amazon.com. 2021 [cited 31 March 2021]. Available from: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/troubleshooting.html#troubleshooting-errors-dependency-error

33. Troubleshooting AWS CloudFormation - Delete Stack Fails - AWS CloudFormation [Internet]. Docs.aws.amazon.com. 2021 [cited 31 March 2021]. Available from: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/troubleshooting.html#troubleshooting-errors-delete-stack-fails

34. Troubleshooting AWS CloudFormation - Nested Stacks - AWS CloudFormation [Internet]. Docs.aws.amazon.com. 2021 [cited 31 March 2021]. Available from: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/troubleshooting.html#troubleshooting-errors-nested-stacks-are-stuck

35. What is the AWS CDK? - AWS Cloud Development Kit (AWS CDK) [Internet]. Docs.aws.amazon.com. 2021 [cited 31 March 2021]. Available from: https://docs.aws.amazon.com/cdk/latest/guide/home.html

36. CodePipeline pipeline structure reference - AWS CodePipeline [Internet]. Docs.aws.amazon.com. 2021 [cited 31 March 2021]. Available from: https://docs.aws.amazon.com/codepipeline/latest/userguide/reference-pipeline-structure.html

37. What is Amazon CloudFront? - Amazon CloudFront [Internet]. Docs.aws.amazon.com. 2021 [cited 31 March 2021]. Available from: https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html

38. Deploying React App With AWS CDK, Cloudfront & Codepipeline | Apoorv Blog [Internet]. Apoorv.blog. 2021 [cited 31 March 2021]. Available from: https://apoorv.blog/deploy-reactjs-cloudfront-codepipeline-cdk/

39. Bootstrapping - AWS Cloud Development Kit (AWS CDK) [Internet]. Docs.aws.amazon.com. 2021 [cited 31 March 2021]. Available from: https://docs.aws.amazon.com/cdk/latest/guide/bootstrapping.html

40. AWS SDK for JavaScript [Internet]. Amazon Web Services, Inc. 2021 [cited 1 April 2021]. Available from: https://aws.amazon.com/sdk-for-javascript/

41. Material-UI: A popular React UI framework [Internet]. Material-ui.com. 2021 [cited 1 April 2021]. Available from: https://material-ui.com/

42. Material Design [Internet]. Material Design. 2021 [cited 1 April 2021]. Available from: https://material.io/design/introduction

43. React Router: Declarative Routing for React [Internet]. ReactRouterWebsite. 2021 [cited 1 April 2021]. Available from: https://reactrouter.com/

44. Tree View React component - Material-UI [Internet]. Material-ui.com. 2021 [cited 1 April 2021]. Available from: https://material-ui.com/components/tree-view/

45. What is Pub/Sub Messaging? [Internet]. Amazon Web Services, Inc. 2021 [cited 1 April 2021]. Available from: https://aws.amazon.com/pub-sub-messaging/

46. @cloudcomponents/cdk-codepipeline-slack [Internet]. npm. 2021 [cited 1 April 2021]. Available from: https://www.npmjs.com/package/@cloudcomponents/cdk-codepipeline-slack

47. Redeploy and roll back a deployment with CodeDeploy - AWS CodeDeploy [Internet]. Docs.aws.amazon.com. 2021 [cited 1 April 2021]. Available from: https://docs.aws.amazon.com/codedeploy/latest/userguide/deployments-rollback-and-redeploy.html

48. Enabling interactivity with Slash Commands [Internet]. Slack. 2021 [cited 1 April 2021]. Available from: https://api.slack.com/interactivity/slash-commands

49. What is BLACK Box Testing? Techniques, Example & Types [Internet]. Guru99.com. 2021 [cited 1 April 2021]. Available from: https://www.guru99.com/black-box-testing.html