Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

IBM DevOps Pipeline Project

Project Development Report

Software Engineering Project
23rd February 2021

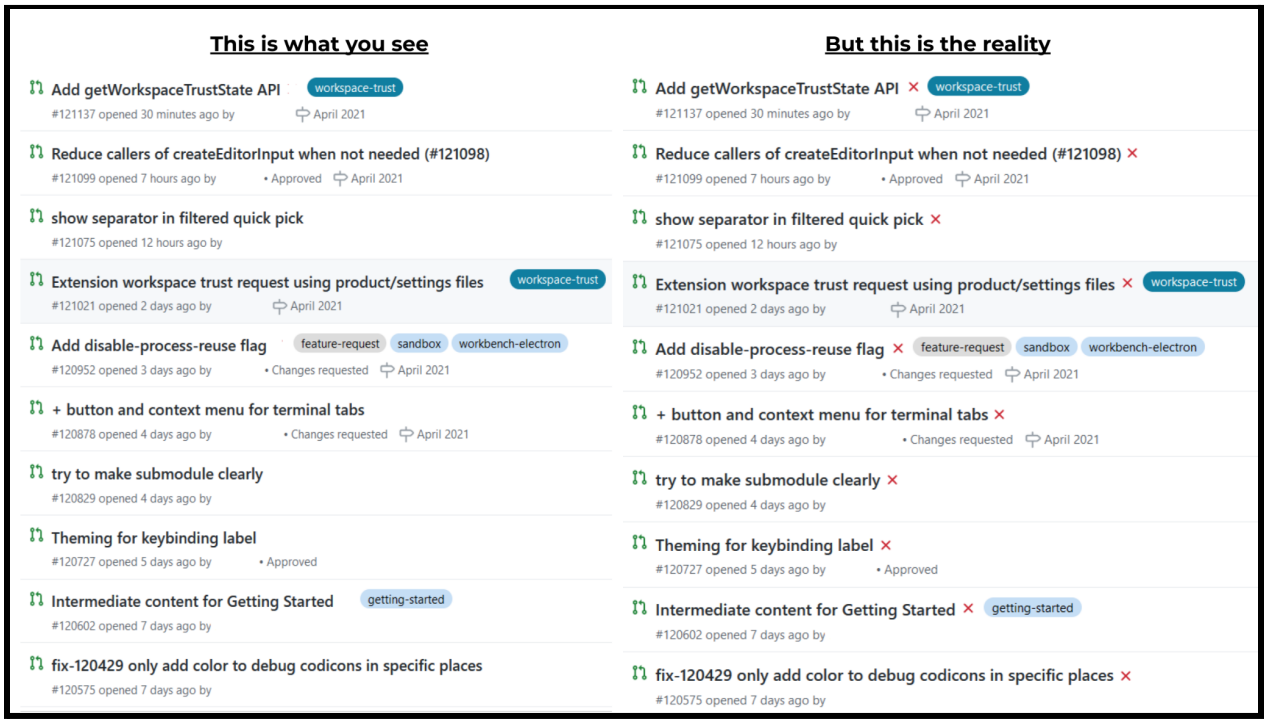
Cormac Madden
Emer Murphy
Tom Roberts
Prathamesh Sai Sankar
Neil Shevlin
Yi Xiang Tan

Contents

1. Introduction

1.1. Background & Problem Statement

When developing applications at scale, it can be hard to streamline development as many people are working on the same code base. Our project was to set up a **complete continuous integration/continuous deployment pipeline** and develop a simple application to demonstrate the use of this pipeline. With the help of our project, one can **streamline development of applications at scale** and find bugs in pull requests before merging them into a main branch. Therefore, one can avoid simple bugs getting into a repository while still maintaining the high rate of development.



Our project will containerize the application and **run testing, building, and linting** for every pull request and deploy it to **Red Hat OpenShift**. Therefore, with the help of this CI/CD pipeline, **one can save time, avoid errors and simultaneously increase the volume and velocity of development.**

1.2. Technical Approach

We divided up the work by dividing the project into two parts.

1. **The CI/CD pipeline.**
2. **The application.**

By doing this, we were able to divide the work necessary to complete this project and we always knew what we had to do. We decided that the 3rd years would be in charge of the CI/CD pipeline and the 2nd years would be in charge of the application. This technical approach of splitting up the work was great, as we were able to develop the project much faster. The 3rd years on our team had minimal experience with CI/CD beforehand, therefore they had to research about it in their own time. Furthermore, only one 2nd year on our team had experience with web development. This meant that we were quite inexperienced coming into this project. However, we were able to adapt and learn as we went along.

2. Requirements

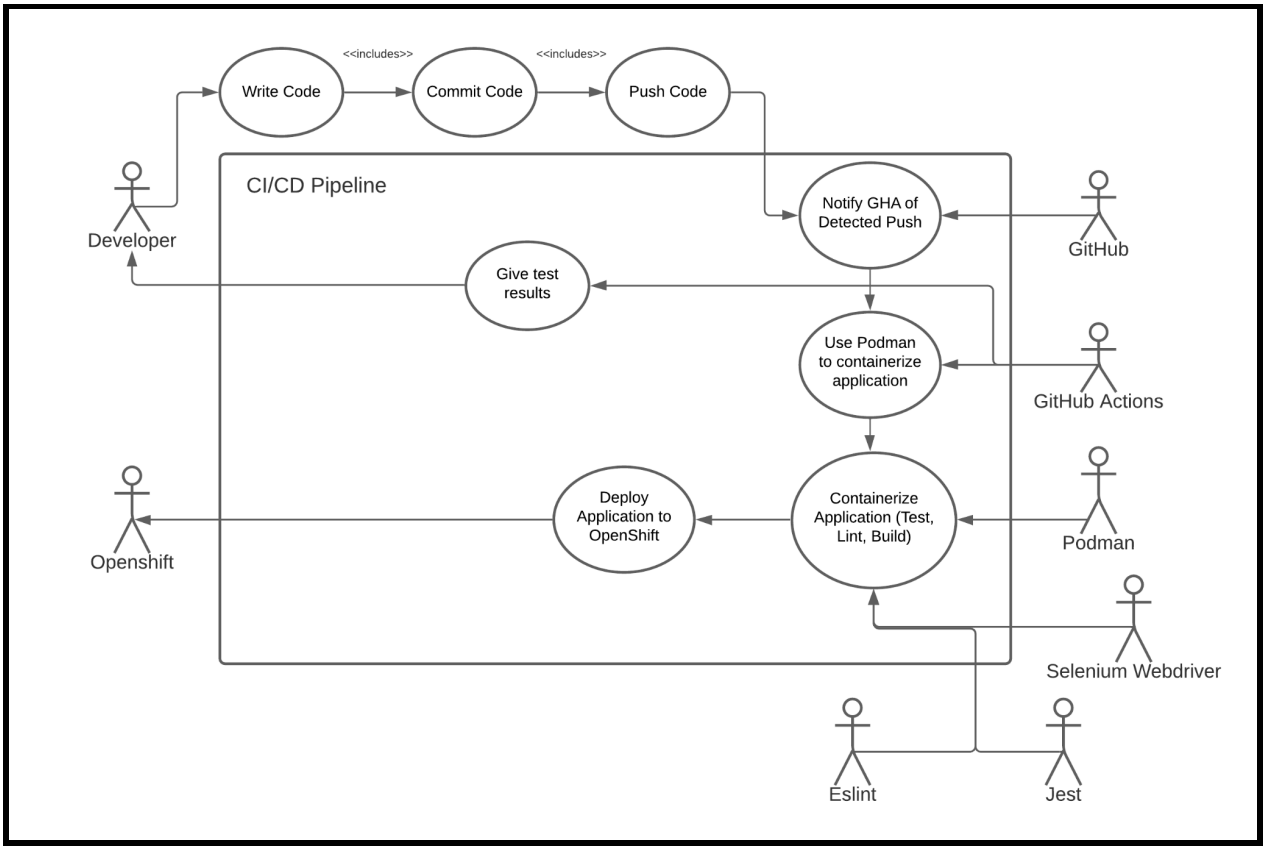
2.1. Functional requirements

- The pipeline **should take code submitted by developers on GitHub**. We should then **build it** and **perform static analysis** such as unit testing.
- The DevOps pipeline must package the application using **Podman**, and then deploy the **containers onto a Red Hat OpenShift Container Platform (OCP).**
- The containers should then run individually on **Linux systems using Podman.**

2.2. Non functional requirements

- **Agile**
    - Client meetup to demonstrate progress on a biweekly basis.
- **Reliability**
    - Performs checks and code analysis for builds.
- **Serviceability**
    - Clearly documented code.
    - Documentation of key architectural decisions.

2.3. User Interaction Scenarios



Name: Push Code **Participating Actor:** Developer **Entry Condition:** The developer must have previously written their code and committed it properly. **Exit Condition:** The developer has successfully pushed their code. **Normal Scenario:** 1. The developer writes their code to the best of their ability. 2. The developer commits their code that they are satisfied with. 3. The developer pushes their code onto the code repository. **Error Scenario:** ● The developer tries to push without a commit beforehand.	**Name:** Notify GitHub Actions of Detected Push **Participating Actor:** GitHub **Entry Condition:** The developer must have pushed code to the code repository for GitHub to notify GitHub Actions. **Exit Condition:** GitHub Actions has been successfully notified, to kick off the CI build. **Normal Scenario:** 1. The developer pushes their code onto the code repository. 2. GitHub recognizes this push and notifies GitHub Actions. 3. GitHub Actions receives a notification from GitHub and starts the CI build. **Error Scenario:** ● GitHub notifying GitHub Actions when there was no push to the repository.

Name: Use Podman to containerize application
Participating Actor: GitHub Actions

Entry Condition: There has been a push to the repository.
Exit Condition: Podman will be invoked to containerize the application.

Normal Scenario:

1. There has been a push made by a developer.
2. Podman will then be invoked to containerize the application with the new changes.

Error Scenario:

- Podman tries to containerize the application when a new push has not occurred.

Name: Containerize the application (Test, Lint, Build)
Participating Actor: Podman, Selenium Webdriver, Jest, Eslint

Entry Condition: GitHub Actions has detected a new push and has now notified Podman to containerize the application..
Exit Condition: The application will be containerized, and testing, linting, and building will be completed.

Normal Scenario:

1. GitHub Actions notifies Podman to containerize the application.
2. Podman containerizes the application and testing, linting and building are completed.

Error Scenario:

- Podman tries to containerize the application without a request from GitHub Actions.

Name: Give test results
Participating Actor: GitHub Actions

Entry Condition: The application has been containerized with testing, linting, and building completed.
Exit Condition: The developer will receive the results of the testing, linting and building to tell them if they need to make any further changes.

Normal Scenario:

1. Podman containerizes the application. The developer then receives the test results from the testing, linting and building.

Error Scenario:

- The test results are not updated and show the results of an older push.

Name: Deploy Application to OpenShift
Participating Actor: Podman

Entry Condition: The application has been containerized.
Exit Condition: The application will be deployed on OpenShift.

Normal Scenario:

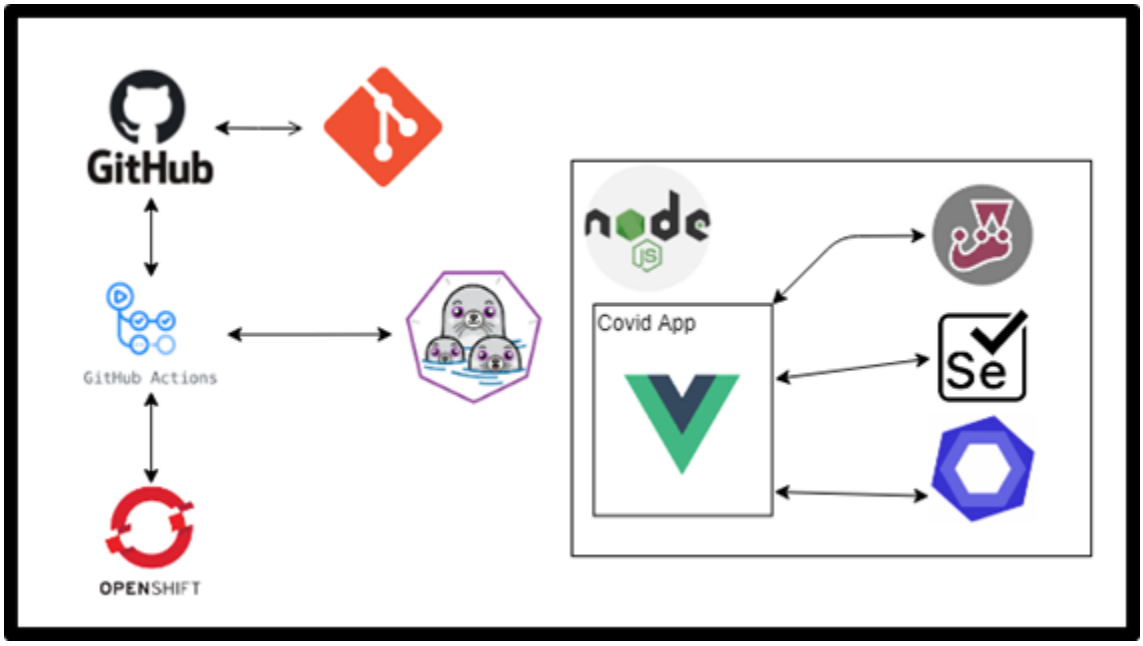
1. The application has been containerized already.
2. OpenShift manages the containers that have been deployed.

Error Scenario:

- Deploying an application that has not been containerized.

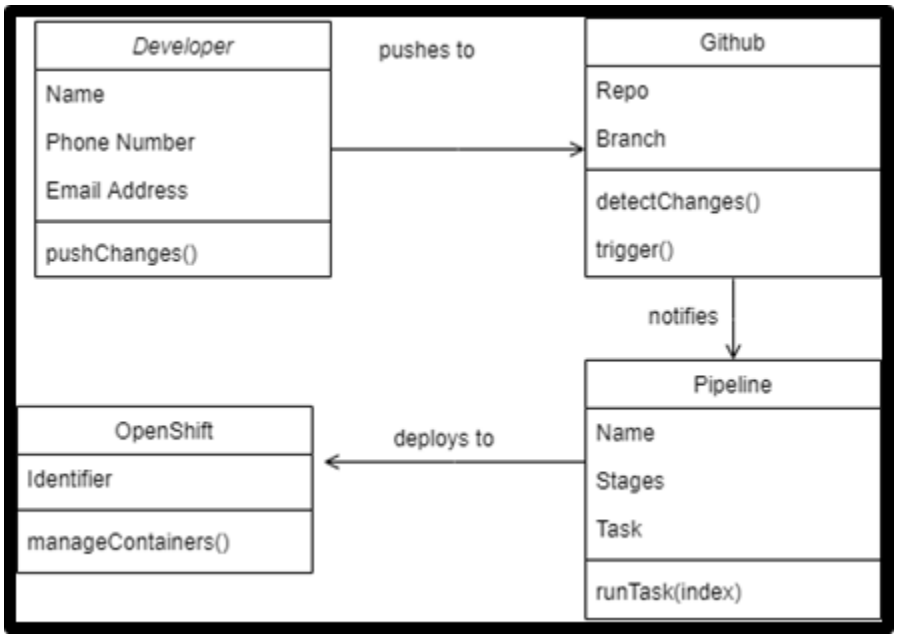
3. Design

3.1. Architecture Diagram

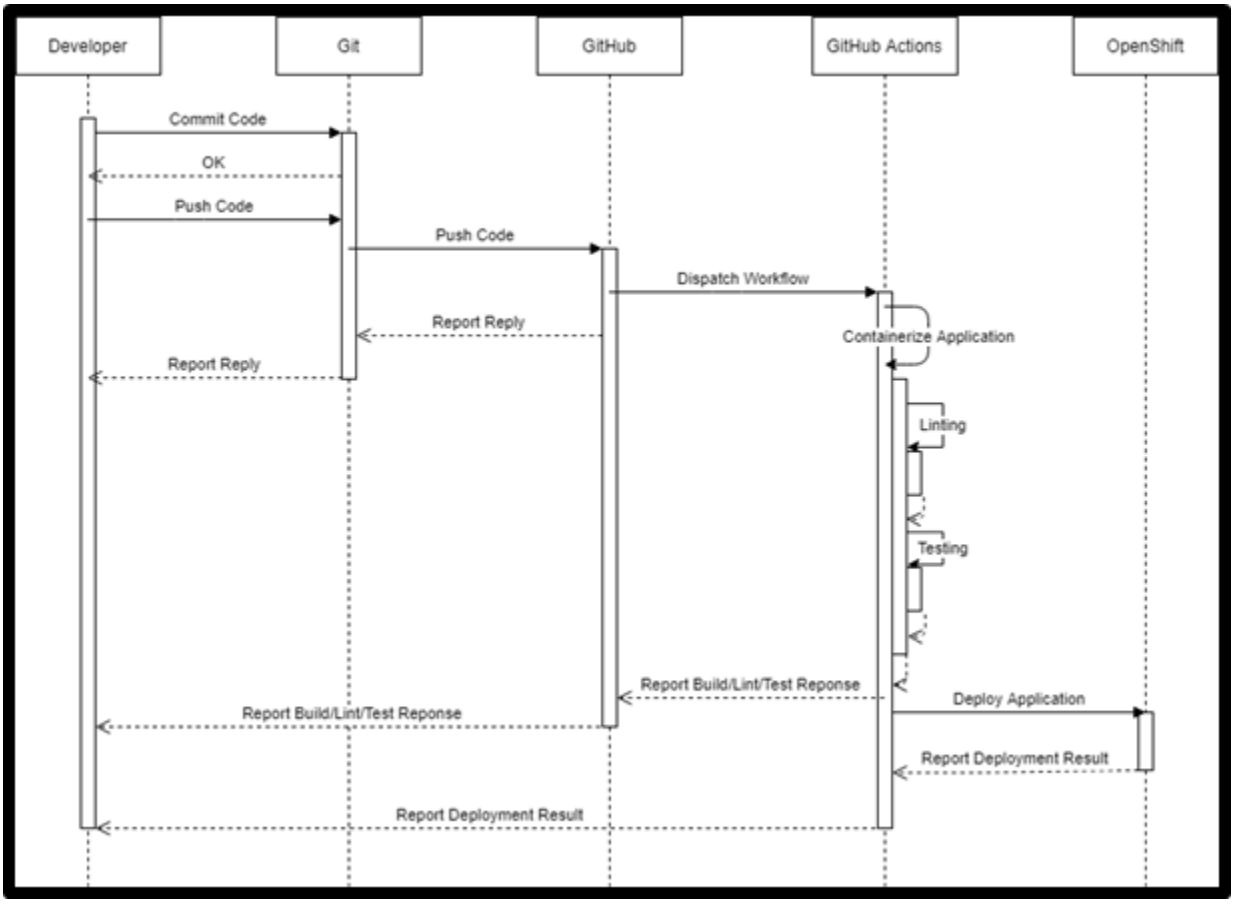


3.2. UML Class and Sequence Diagrams

Class diagram



Sequence diagram



4. Design

4.1. Tools, Libraries, Platforms

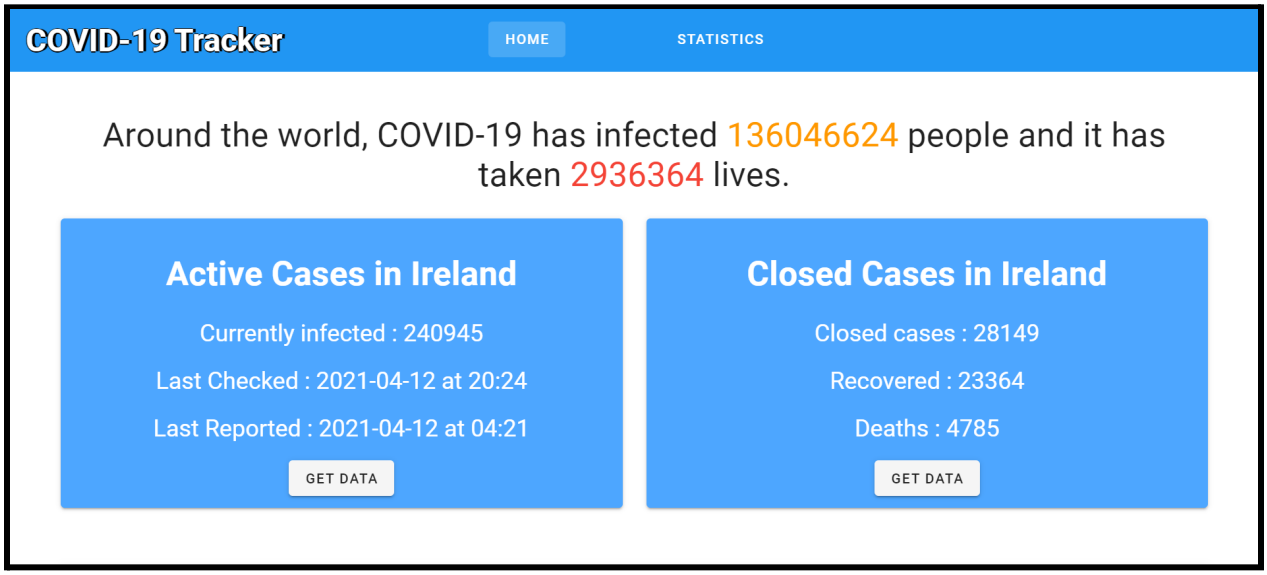
Tools	Functionality
Git	A DevOps tool to enable collaboration of work and track modifications to the files
Google Drive	For collaboration of documentation and storage solution of said documentation.
Selenium Webdriver	To test the code quality of the code we develop.
Jest	To do code coverage testing within the CI/CD pipeline.
Podman	For containerisation of the application.

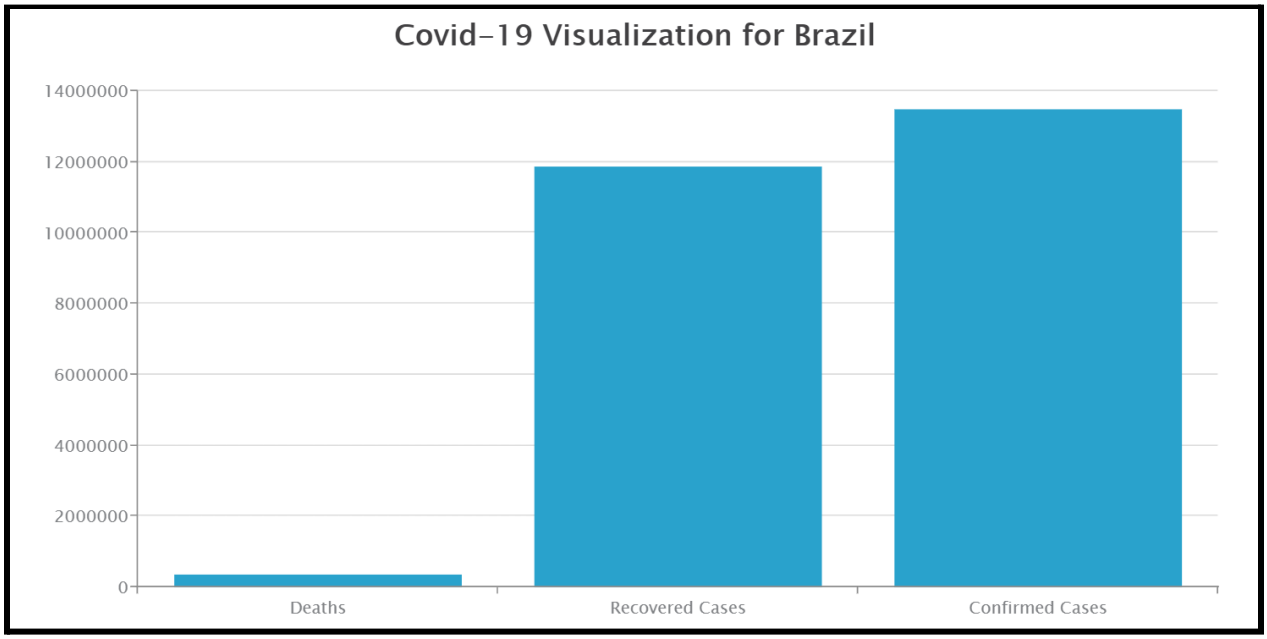
Libraries	Functionality
Vue	A JavaScript library for building web interfaces, which can be used for frontend development.
Node	A JavaScript library for building web interfaces, which can be used for backend development.
ZingChart	A JavaScript library for building interactive and intuitive charts for showing various data elements.
Axios	A promise based HTTP client for browsers and Node.
Vue-Multiselect	A Vue library for a complete selecting solution for Vue without JQuery.
Vuetify	A Vue library for making Vue components simple and aesthetically pleasing.

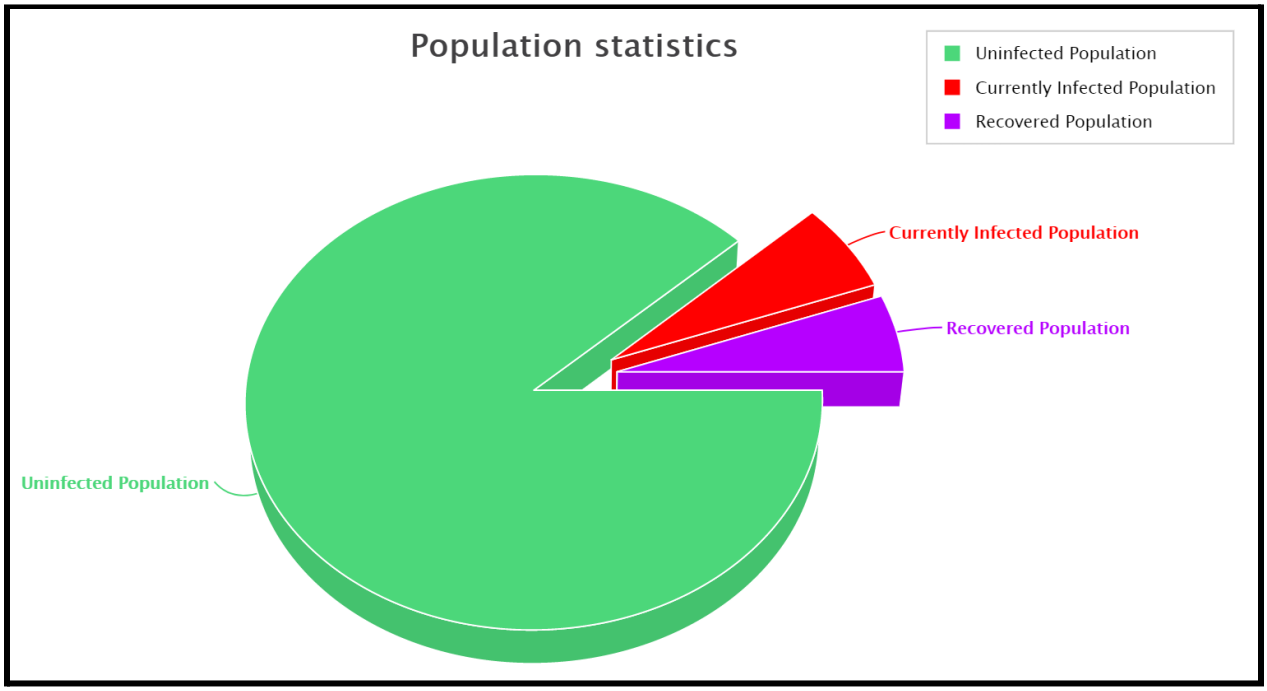
Platforms	Functionality
GitHub	A code hosting platform where we will keep all of our code.
GitHub Projects	For the outline of tasks.
Discord	Structured communication and video conference for scheduled meetings.
Jist Meets	For the recording of video conferences for scrum meeting video submissions.
Google Meet	Video Conference channel with the client.
GitHub Actions	To support the CI/CD use cases of application development.
OpenShift	For building and scaling containerized applications.

4.2. User Interfaces

Our application is a COVID-19 Tracker which shows information about COVID-19 for multiple countries. It gets its information from multiple APIs and shows this information through both text, and graphs such as the bar chart and pie chart shown below. The frontend components were developed with Vue.js and the backend was made with Node.js. The user can update the data shown to them by requesting to “get data”. Therefore, this application offers real time COVID-19 statistics that can be immensely helpful during the current pandemic we are in.







4.3. Algorithms

In the application, we used particular algorithms to collect data from multiple APIs such as the COVID-19 API and the Population API and show them in real time. We used Axios to make HTTP requests to the API's we needed, and then we implemented an algorithm to set values for particular properties on the page from "N/A" and "..." to their valid information from the API in real time. We decided to set important data values to "N/A" and "..." on startup of the application and then we update those values after the API calls were successfully made. We did this because we wanted to ensure that both developers and users of the application can see when the API calls are properly working.

On the statistics page, we had a drop-down menu which let the user choose a country to visualize COVID-19 information from. The implementation of an algorithm to refresh the data from the API based on the drop down menu's options was tricky. This was implemented by automatically setting the data value for the country in question based on the user's drop down menu option, and then calling algorithms to update the information from both the COVID-19 and population API with said country name from the drop-down menu as one of the parameters to the API call.

Finally, we also had to implement not only algorithms for retrieving data from the COVID-19 API but also the population API. The information from this algorithm is then shown in both text and with the help of pie charts. We calculated the uninfected population by getting real time population

data from the country in question and subtracting it from both the recovered and infected population. Furthermore, we made similar API calls for other information and we updated the charts such as the bar chart and pie chart continuously as the user requested from the “Render Latest Information” button.

5. Conclusion

5.1. Design and Implementation

An issue with the design of the project was that it was hard to find much information on implementing Vue.js with Node.js both in the same application; the resources that were available were too advanced for some members of the group. This resulted in spending a lot of time trying to understand high level documentation for the first time. Furthermore, one of the issues we faced in the implementation was the lack of experience with the different technologies required for this project. This resulted in a large period of time spent researching and learning about the new tools and reduced the amount of time that we could spend programming.

5.2. Project Objectives

We worked hard to reach our objectives. Overall we wanted to create a project that we are proud of and that we can show off, and we succeeded at this. We created a pipeline using GitHub Actions that builds an image of the application using Podman. It performs static analysis, unit testing and code coverage on the image using WebDriver. It also packages our application in the form of a container and deploys the application to OpenShift. We created graphs for the application to use a public Covid API. We had clear documentation and a clear plan and have our code available as an open-source project on GitHub.

5.3. The Team

Things that we did well:

Overall the group worked well together. This project required a lot of different technologies, the members with more experience helped those with less. When making decisions, as a group we were able to successfully discuss various options and ideas, coming to a conclusion without conflict. We had group meetings every week, this allowed the team to report and track our progress easily. It allowed us to keep in contact in regards to any issues or if the plan was changing. We were able to bring up any struggles we had completing our individual tasks and get some help. The meetings gave us the opportunity to redirect our focus on the task at hand and ensure that we’re on the right track based on what our client had requested from us. The 3rd year students managed the project well, this contributed to the overall success of the project.

Thing that could have been improved:

Communication was sometimes difficult with interactions entirely online. When there was an issue it was difficult to communicate it over text or video, a lot of key communication is lost when things are not face to face. Sometimes people's wifi was not reliable and people having opposite schedules made organising meetings difficult. We could improve these issues if we could meet in college.

5.4. Algorithms

The algorithms we used in our application were well designed. We were able to design algorithms to retrieve information from APIs efficiently and quickly. For further improvements, we would have liked to condense some functions for getting data for different countries and merge them into one singular algorithm.

For example, in the "DataTable" Vue component, we have different algorithms and functions for getting data for different countries. Possibly merging the functionality of these functions would have been nice, but we were limited by the lack of prior experience in these areas. However, having said that, we were still able to implement efficient functions which accurately pull information from the APIs in question and show the data to the user.

In conclusion, we were able to successfully complete this project to a high standard with the implementation of algorithms and functions for the application. We were unsure whether we would be able to complete the project to a high standard initially because of a lack of experience in CI/CD and web development. However, we were able to complete the project to a high degree and our client Mr. Mihai Criveti from IBM is very pleased with the implementation and design of our project overall.

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

IBM DevOps Pipeline Project

Requirements Document

Software Engineering Project
23rd February 2021

Cormac Madden
Emer Murphy
Tom Roberts
Prathamesh Sai Sankar
Neil Shevlin
Yi Xiang Tan

1. Introduction

1.1. Overview - Purpose of system

The purpose of building a CI/CD pipeline is to facilitate DevOps[1] — the amalgamation of cultural philosophies, practices, and tools that speeds up an organization's ability to deliver applications and services at high velocity. This allows a team to evolve and improve products quicker than organizations using traditional software development and infrastructure management processes.

Working together with IBM, we plan to build a GitOps pipeline to build and deploy an application to OpenShift. This automation will include performing static analysis, unit testing, code coverage, packaging in the form of a container using Podman[2] and deploying onto Red Hat OpenShift[3].

1.2. Scope

In order to build this GitOps CI/CD pipeline project, there are a few things that must be developed,

1. A web application using Node.js to test the deployment of the code.
2. Automated testing and static program analysis.
3. Automated deployment to Red Hat OpenShift.

1.3. Objectives and Success Criteria

Our objectives are:

Familiarise ourselves with the technologies available to build CI/CD Pipeline.

Decide on the suitable infrastructure to build the pipeline.

Develop a rudimentary web application using Javascript & Node.js

Create a pipeline using GitHub Actions that will:

- Build and image of the application using Podman
- Perform static analysis, unit testing and code coverage on the image using Webdriver and Jest.
- Package our application in the form of a container.
- Deploy the application to OpenShift.

Further develop the application to use a public Covid API.

Create more graph and visualisation options for the web application.

Create clear documentation of the plan and the process.

Deliver the code bundle to the client before April 23rd.

Our Success Criteria are:

Completing the objectives outlined.

Satisfying our client with the code we deliver.

Ultimately having a pipeline that will build and deploy an application on to OpenShift.

1.4. Definitions, abbreviations

- **CI/CD:** Continuous Integration/Continuous Delivery.
- **CI/CD pipeline:** This helps us to automate the steps in the software design process. The pipeline builds code, runs tests (CI), and safely deploys a new version of the application(CD). Automated pipelines remove manual errors, provide standardized feedback loops to developers, and enable fast product integration.
- **Standardisation:** This is a common format of a document or file that is used by one or more software developers while working on the same computer program. Software standards allow for the interoperability between different programs created by different developers.

2. Current system

Pipelines for building node.js applications, performing tests and deployment are widely available.

We will be using this project below, as a reference for some of our work.
https://GitHub.com/christophernixon/DevOps-Pipeline-sweng?fbclid=IwAR0yLz6i-m9xEMPvn06DXHnyWXYT-p4yzV9VmGzWeRELuFqoo2L7NsHSL5g
This project was created in 2020 by another group of students doing a similar project to this.

Their pipeline had the following structure: **Git - Travis [React - Jest - Docker - IBM]**

We will be using GitHub Actions instead of Travis because one of our group members has some experience with it and setting it up is relatively straightforward.

We plan on using Javascript and Webdriver instead of React and Jest for developing and testing our application. We decided on this based on the experience of our team members.

We will be using Podman instead of Docker. Podman is quite similar to Docker. They use the same commands and Podman and docker images are compatible.

We will also be deploying the pipeline to OpenShift instead of IBM cloud. This is one of the requirements from the client

Updated Pipeline Structure: **Git - GitHub Actions [Node.js - Webdriver - Podman - OpenShift]**

Available CI solutions:

GitHub Actions	Built-in support by GitHub with documentation covering most of our use cases.
Travis CI	Ideal for open source projects that require testing in multiple environments.
Jenkins	Suited for larger projects that require a high degree of customization.

Other options include Gitlab, Circle CI and Codeship.

Version-control platforms:

GitHub	Distributed version control system.
SVN	A centralized version control system.
Apache Subversion	Distributed version control system.

Containerisation:

Podman	We are mandated to use this. Can run rootless.
Docker	Uses OS-level virtualization to deliver software in packages called containers.

3. The Proposed System

3.1. Overview

Our design is for a complete CI/CD pipeline and a simple application to demonstrate the use of this pipeline. The application will visualise covid statistics on a dashboard. Version-control will be handled with Git to enable collaboration of work and track modifications to the files, and we'll use GitHub Actions to perform Continuous Integration. To test code quality we'll be using Webdriver. There won't be a database, it will be replaced with an external API for demonstration. The simple application to test the effectiveness of pipelines will be a web application made with JavaScript and Node js.

3.2. Functional Requirements

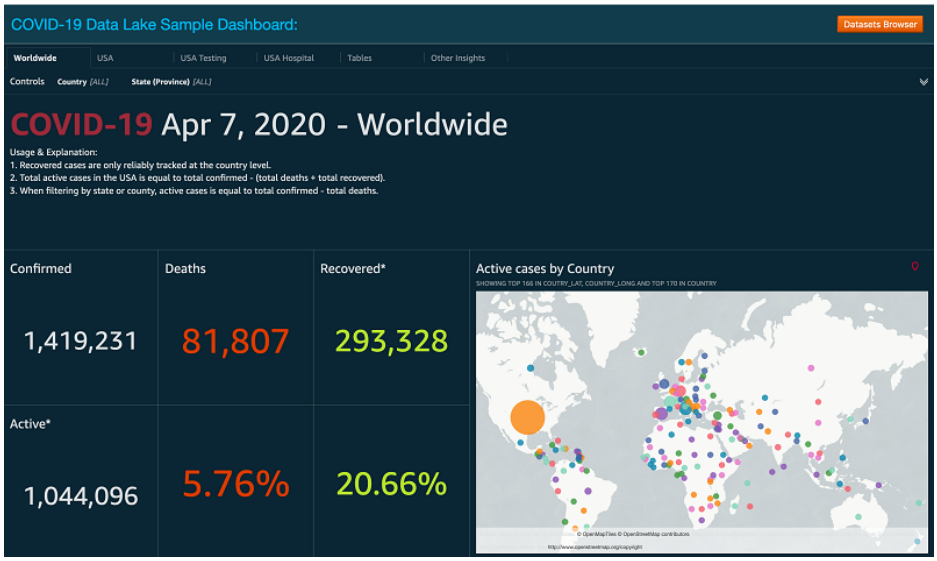
- The pipeline should take code submitted by developers on GitHub, build it. Then perform static analysis such as unit testing.
- The DevOps pipeline must package the application using Podman, and then deploy the containers onto a Red Hat OpenShift Container Platform (OCP).
- The containers should then run individually on Linux systems using `Podman`.

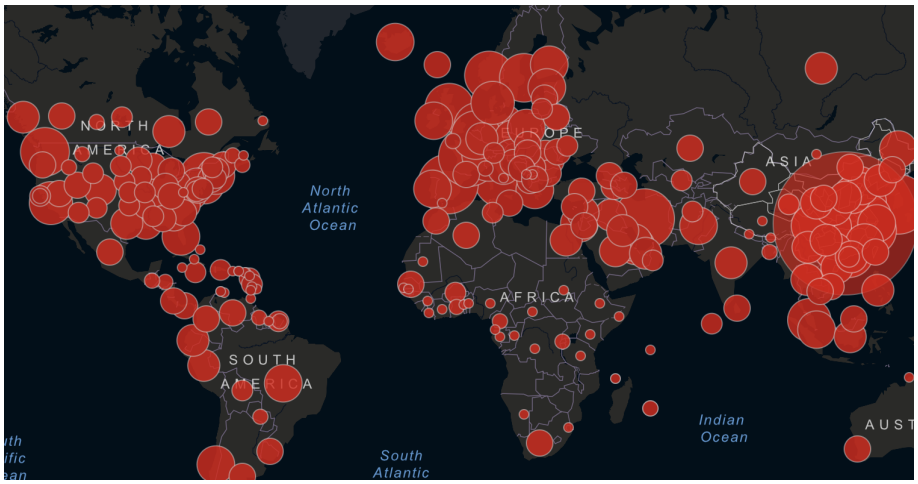
3.3. Non-functional requirements

- **Agile**
    - Client meetup to demonstrate progress on a biweekly basis
- **Reliability**
    - Performs checks and code analysis for builds
- **Serviceability**
    - Clearly documented code
    - Documentation of key architectural decisions

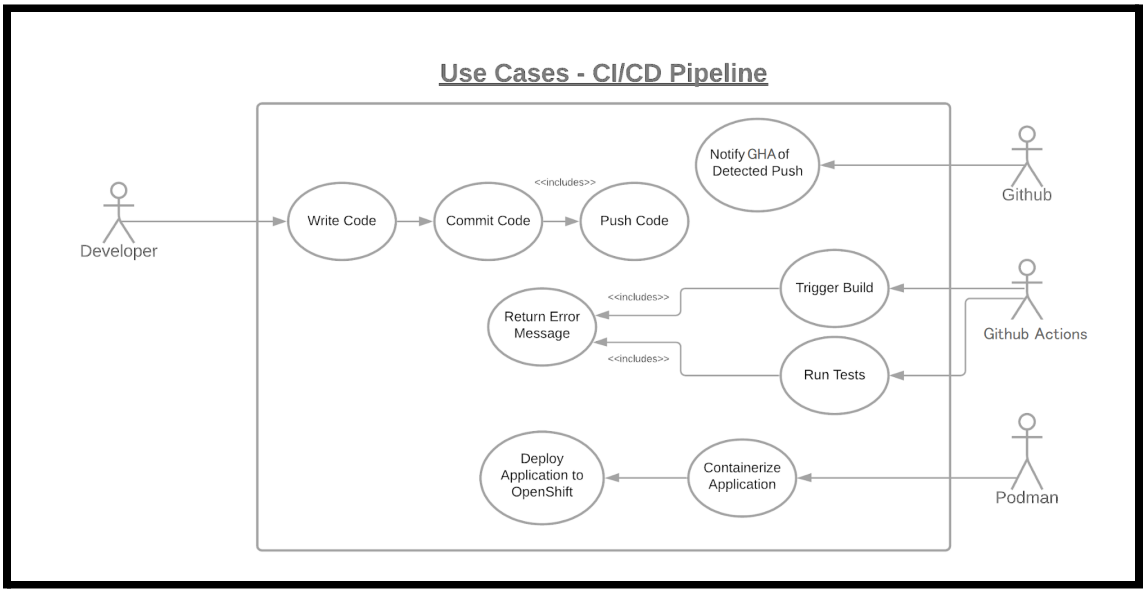
3.4. System prototype (models)

User interface mockups





Use cases



Use Case Descriptions

Name: Push Code **Participating Actor:** Developer **Entry Condition:** The developer must have previously written their code and committed it properly. **Exit Condition:** The developer has successfully pushed their code. **Normal Scenario:** 1. The developer writes their code to the best of their ability. 2. The developer commits their code that they are satisfied with. 3. The developer pushes their code onto the code repository. **Error Scenario:** ● The developer tries to push without a commit beforehand.	**Name:** Notify GitHub Actions of Detected Push **Participating Actor:** GitHub **Entry Condition:** The developer must have pushed code to the code repository for GitHub to notify GitHub Actions. **Exit Condition:** GitHub Actions has been successfully notified, to kick off the CI build. **Normal Scenario:** 1. The developer pushes their code onto the code repository. 2. GitHub recognizes this push and notifies GitHub Actions. 3. GitHub Actions receives a notification from GitHub and starts the CI build. **Error Scenario:** ● GitHub notifying GitHub Actions when there was no push to the code repository.

Name: Trigger Build
Participating Actor: GitHub Actions

Entry Condition: GitHub has successfully notified GitHub Actions of a code push.
Exit Condition: The code will be successfully built.

Normal Scenario:

1. GitHub Actions receives a notification from GitHub.
2. GitHub Actions starts the CI build.

Error Scenario:

- There was an error when building the code.

Name: Run Tests
Participating Actor: GitHub Actions

Entry Condition: GitHub has successfully notified GitHub Actions of a code push.
Exit Condition: The code will be successfully tested.

Normal Scenario:

1. GitHub Actions receives a notification from GitHub.
2. GitHub Actions starts the QA testing with selenium.

Error Scenario:

- There was an error when testing the code.

Name: Containerize Application
Participating Actor: Podman

Entry Condition: The code has been successfully built and tested.
Exit Condition: The application will be containerized.

Normal Scenario:

1. The code has been successfully built and tested.
2. Podman containerizes the application.

Error Scenario:

- Trying to containerize an application that has not been built and tested.

Name: Deploy Application to OpenShift
Participating Actor: Podman

Entry Condition: The application has been containerized.
Exit Condition: The application will be deployed on OpenShift.

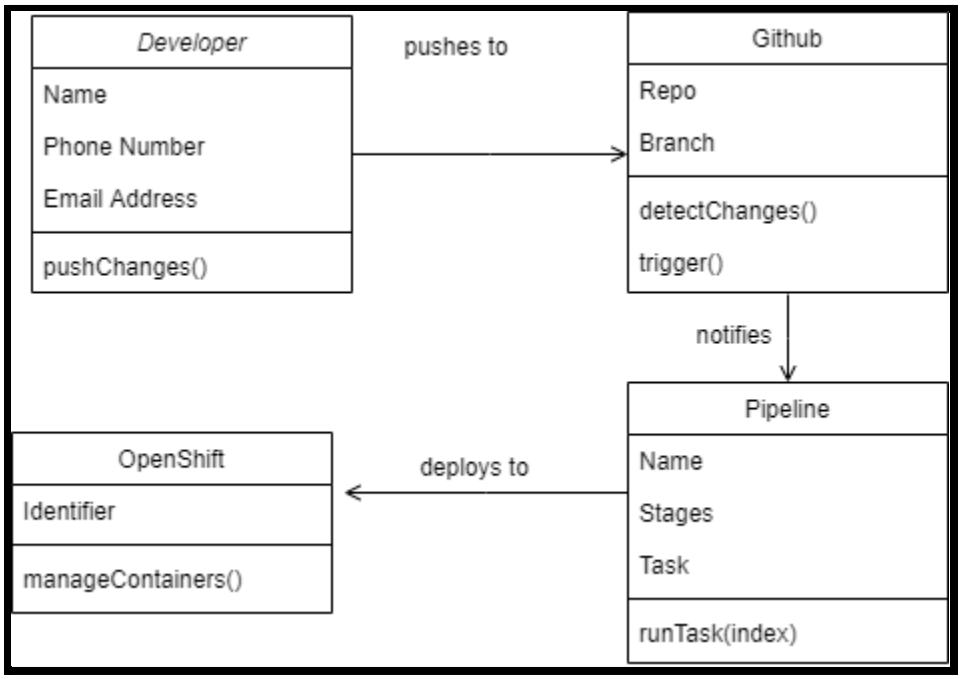
Normal Scenario:

1. The application has been containerized already.
2. OpenShift manages the containers that have been deployed.

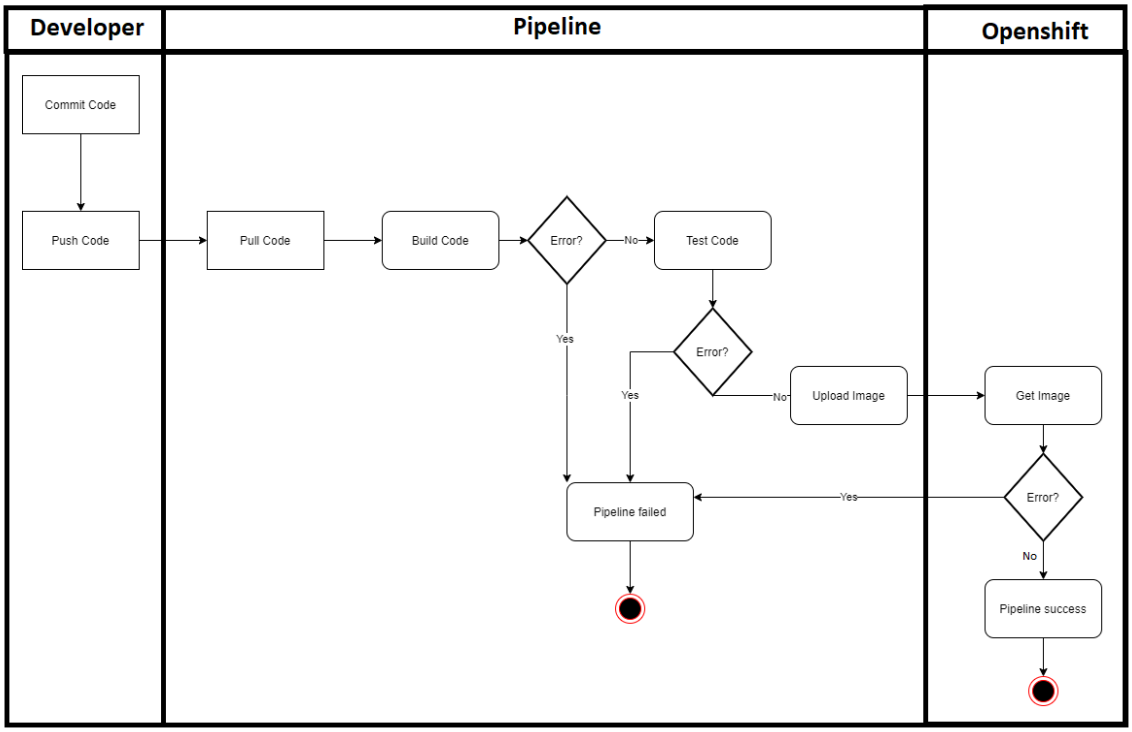
Error Scenario:

- Deploying an application that has not been containerized.

3.4.1. Object model



3.4.2.Dynamic model



References

What is DevOps? - Amazon AWS

Podman

Red Hat OpenShift

https://learn.OpenShift.com

Deploying to OpenShift using GitHub Actions

OpenShift Pipelines with Jenkins, Tekton and more...

Podman and Buildah for Docker users – Red Hat Developer Blog

https://developers.redhat.com/developer-sandbox

Software Design Specification

1. Introduction

1.1. Overview - Purpose of system

The purpose of building a CI/CD pipeline is to support **DevOps** [1] — the amalgamation of cultural philosophies, practices, and tools that speeds up an organization's ability to deliver applications and services at high velocity. This allows a team to develop and improve products quicker than organizations using traditional software development and infrastructure management processes.

Working together with **IBM**, we plan to build a GitOps pipeline to build and deploy an application to OpenShift. This automation will include performing static analysis, unit testing, code coverage, packaging in the form of a container using **Podman** [2] and deploying onto **Red Hat OpenShift** [3].

1.2. Scope

In Scope	• Automated building • Automated testing & static program analysis • Automated containerisation • Automated deployment to Red Hat OpenShift • Creating a web application to test the pipeline of the code.
Out of Scope	• Setting up a messaging system

1.3. Definitions, abbreviations

- **CI/CD:** Continuous Integration/Continuous Delivery.
- **CI/CD pipeline:** This helps us to automate the steps in the software design process. The pipeline builds code, runs tests (CI), and safely deploys a new version of the application(CD). Automated pipelines remove manual errors, provide standardized feedback loops to developers, and enable fast product integration.
- **Standardisation:** This is a common format of a document or file that is used by one or more software developers while working on the same computer program. Software standards allow for the interoperability between different programs created by different developers.
- **Containerisation:** The process of packaging an application along with its required libraries, frameworks, and configuration files together so that it can be run in various computing environments efficiently.
- **Deployment:** This refers to the process of running an application on a server or device.
- **Image:** an immutable file that contains the source code, libraries, dependencies, tools, and other files needed for an application to run.

Software Design Specification

2. System Design

2.1. Design Overview

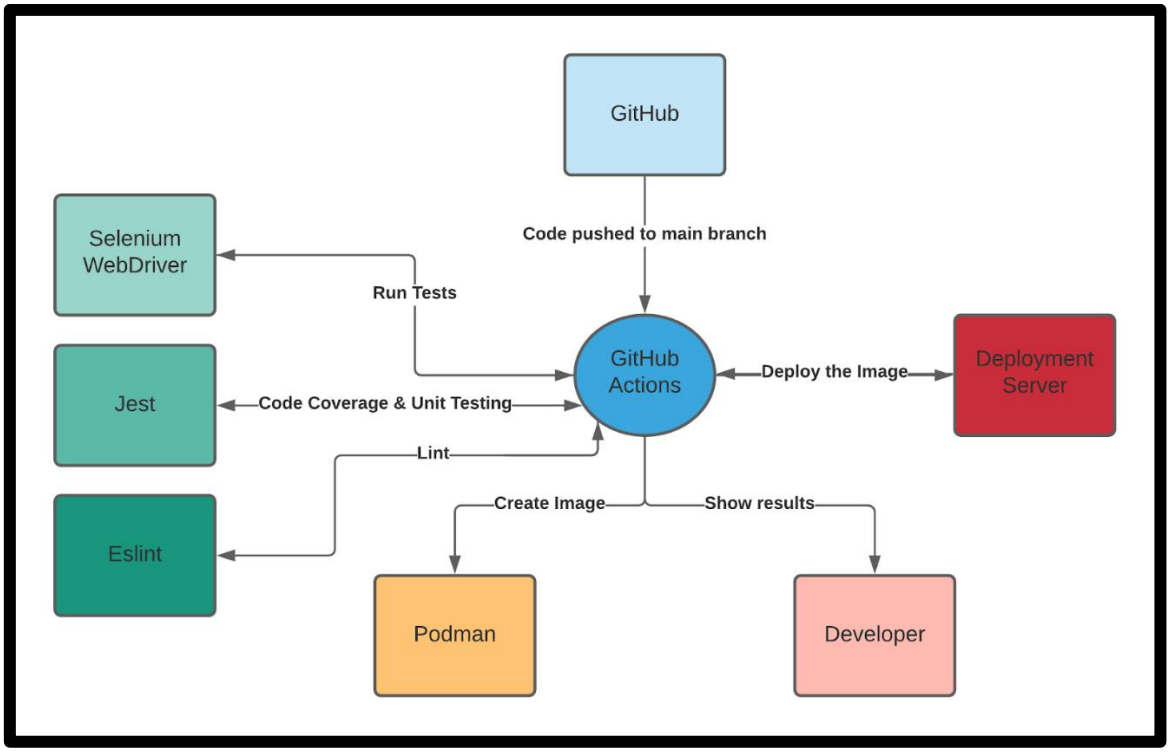
Our design is for a fully functional **CI/CD pipeline** and a **simple application** to demonstrate the use of this pipeline. This will allow developers to write, commit, and push code into a repository, and the code will continuously be built and tested.

2.1.1. High-level overview of how the system is implemented, what tools, frameworks and languages are used etc.

The application we will use to demonstrate out pipeline will be a simple application showing **Covid-19** [4] **statistics on a dashboard**. The version-control will be handled with **Git** [5] to enable collaboration of work and track modifications to the files and we will use **GitHub Actions** [6] to perform Continuous Integration. For testing, we will be using **Selenium Webdriver** [7]. For code coverage and static unit testing, we will be using **Jest** [8]. For linting, we will use **Eslint** [9]**.** For containerization, we will use **Podman.** There will not be a database, it will be replaced with an **external API - Rapid API** [10] for demonstration. Finally, the application will be deployed onto **Openshift.** This simple application will be a Webapp made with **Vue.js** [11] for the front-end, and **Node.js** [12] for the back-end.

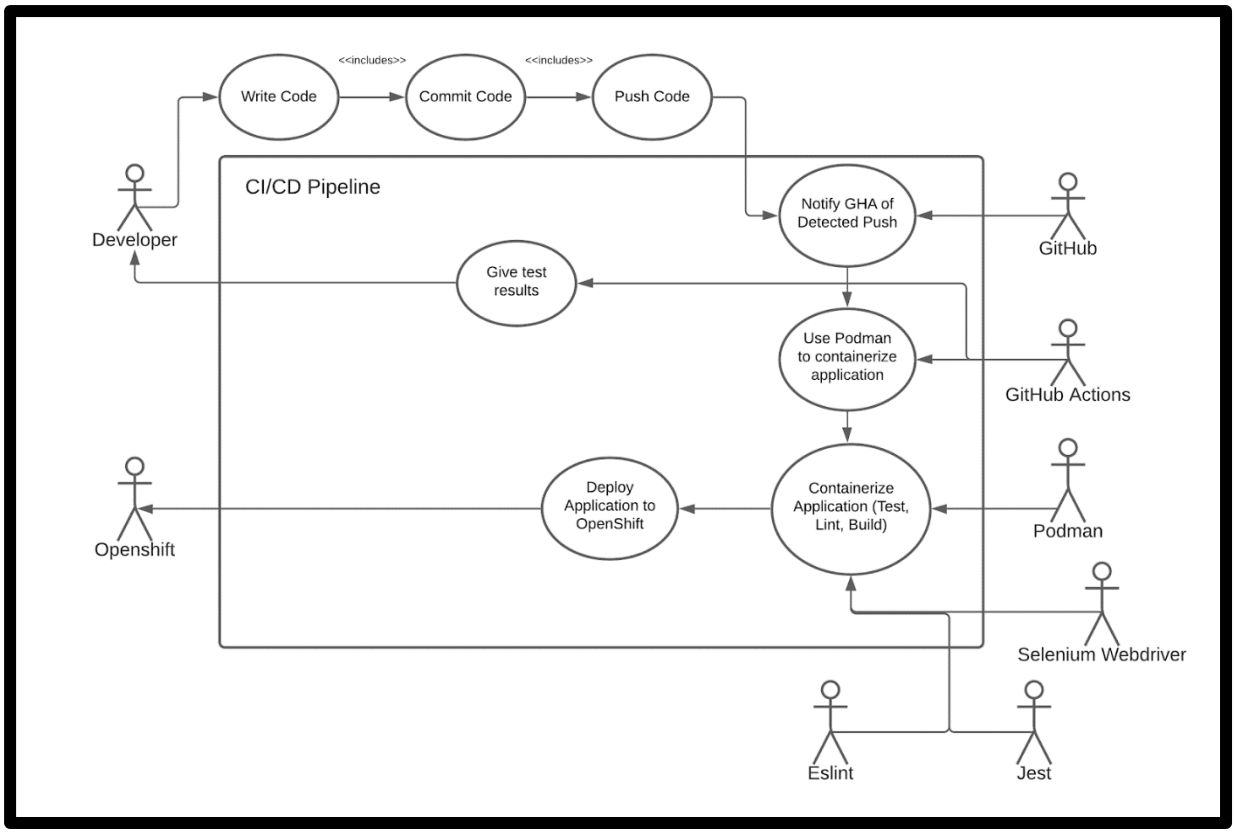
2.2. System Design Models

2.2.1. System Context



Software Design Specification

2.2.2. Use cases (from Requirements)



Name: Write, Commit, Push Code **Participating Actor:** Developer **Entry Condition:** The developer must change the code on their local machine. **Exit Condition:** The developer has successfully written, committed the code into a git repository, triggering the CI/CD pipeline. **Normal Scenario:** 1. The developer writes their code to the best of their ability. 2. The developer commits their code that they are satisfied with. 3. The developer pushes their code onto the code repository. **Error Scenario:**	**Name:** Notify GitHub Actions of Detected Push **Participating Actor:** GitHub **Entry Condition:** The developer must have pushed code to the code repository for GitHub to notify GitHub Actions. **Exit Condition:** Github Actions has been successfully notified, to kick off the CI build. **Normal Scenario:** 1. The developer pushes their code onto the code repository. 2. GitHub recognizes this push and notifies GitHub Actions. 3. GitHub Actions receives a notification from GitHub and starts the CI build. **Error Scenario:**

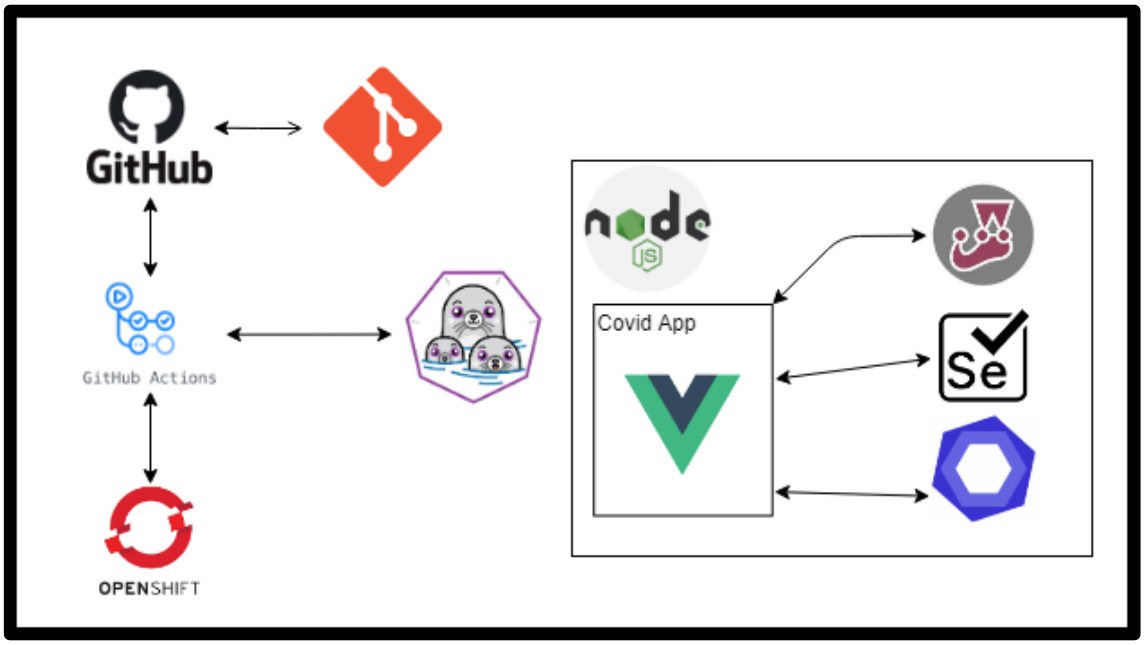
Software Design Specification

• The developer tries to push without a commit beforehand.	• GitHub notifying GitHub Actions when there was no push to the code repository.
Name: Use Podman to Containerize Application **Participating Actor:** GitHub Actions **Entry Condition:** GitHub has successfully notified GitHub Actions of a code push. **Exit Condition:** Podman will start to containerize the application. **Normal Scenario:** 1. Github Actions receives a notification from Github. 2. Github Actions notifies Podman to containerize the application. **Error Scenario:** • Trying to notify Podman without a notification of a code push.	**Name:** Containerize Application (Test, Lint, Build) **Participating Actor:** Podman, Selenium Webdriver, Jest, Eslint. **Entry Condition:** Podman was informed to containerize the application. **Exit Condition:** The application will be containerized, and tasks such as testing, linting, and building will be completed. **Normal Scenario:** 1. Podman containerises the application. 2. Testing is done with Selenium Webdriver and code coverage with Jest. 3. Linting is done with Eslint. 4. The application is built. **Error Scenario:** • Containerizing the application without code coverage from Jest.
Name: Deploy Application to Openshift **Participating Actor:** Openshift **Entry Condition:** The application has been containerized. **Exit Condition:** Openshift will manage the containers. **Normal Scenario:** 1. The application has been fully containerized and finished static unit testing. 2. The container will be handled by Openshift. **Error Scenario:**	**Name:** Give test results **Participating Actor:** Github Actions **Entry Condition:** The application has gone through testing, linting, and building. **Exit Condition:** The developer can see if the application has passed all tests and if the application has been built. **Normal Scenario:** 1. The CI/CD pipeline tests the application, and builds the application. 2. The developer gets the results from the tests and from building the application.

Software Design Specification

• Trying to deploy an application that is not containerized yet.	**Error Scenario:** • Giving test results when no tests and building has occured.

2.2.3. System Architecture



Our system architecture is displayed above and shows how the different components of our project work cohesively as part of a system. We use a git repository on GitHub to store our source code, which is mainly written using Javascript, using Vue.js and Node.js. GitHub Actions recognizes any push to this repository and uses Podman for containerization. After the application is containerized, carry out steps such as building the code, testing the code, and linting it. This is with the help of Selenium Webdriver, Jest, and Eslint.

2.2.3.1 Architectural Decisions

Architectural Decision	Git
Problem Statement	Multiple version control solutions to be chosen from.
Motivations	To enable collaboration of work and track modifications to the files
Alternatives	Subversion
Justification	Distributed, no assets large enough to validate the use of subversion
References	https://git-scm.com/

Software Design Specification

Architectural Decision	Web application (JS, Node.js)
Problem Statement	Multiple applications to be chosen from.
Motivations	To develop a dummy application used in testing the effectiveness of pipelines.
Alternatives	Ruby on Rails, React, Typescript
Justification	Application is not within scope to validate the use of additional frameworks (React/Typescript), could also simulate the agile environment with unplanned changes.
References	https://nodejs.org/en/

Architectural Decision	No database
Problem Statement	Inclusion of database to be chosen from.
Motivations	To decide application structure
Alternatives	Postgres, MySql, MongoDB, GraphQL
Justification	Not within requirements and unnecessarily added complexity, could easily be replaced with Rapid API for demonstration.
References	https://rapidapi.com/marketplace

Software Design Specification

Architectural Decision	Selenium Webdriver
Problem Statement	Multiple testing frameworks to be chosen from.
Motivations	To test code quality
Alternatives	Jest, Puppeteer
Justification	It is extensive (and less limiting), open-sourced, and Neil has extensive experience with it.
References	https://www.selenium.dev/documentation/en/webdriver/

Architectural Decision	GitHub Actions
Problem Statement	Multiple pipelining solutions to be chosen from.
Motivations	To support CI/CD use cases of application development.
Alternatives	Travis, Jenkins, DevOps
Justification	Built-in support by GitHub with documentations covering most of our use cases.
References	https://github.com/features/actions

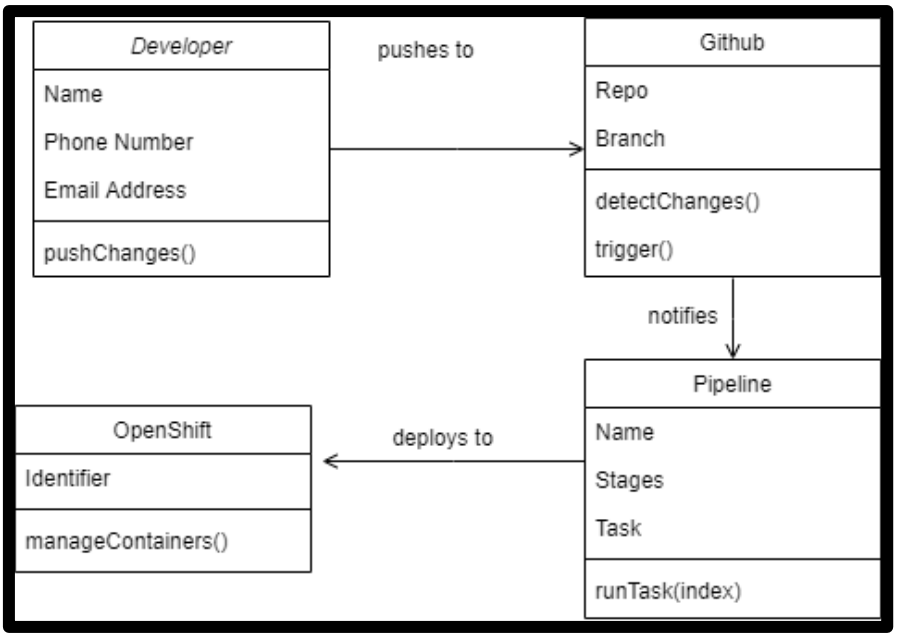
Software Design Specification

Architectural Decision	Vue.js
Problem Statement	Multiple frontend framework to be chosen from.
Motivations	To integrate the frontend framework for easier development.
Alternatives	React, Next.js
Justification	Vue is the middle group from Vanilla JS and React JS, while still providing many benefits and out-of-the-box tooling.
References	https://vuejs.org/

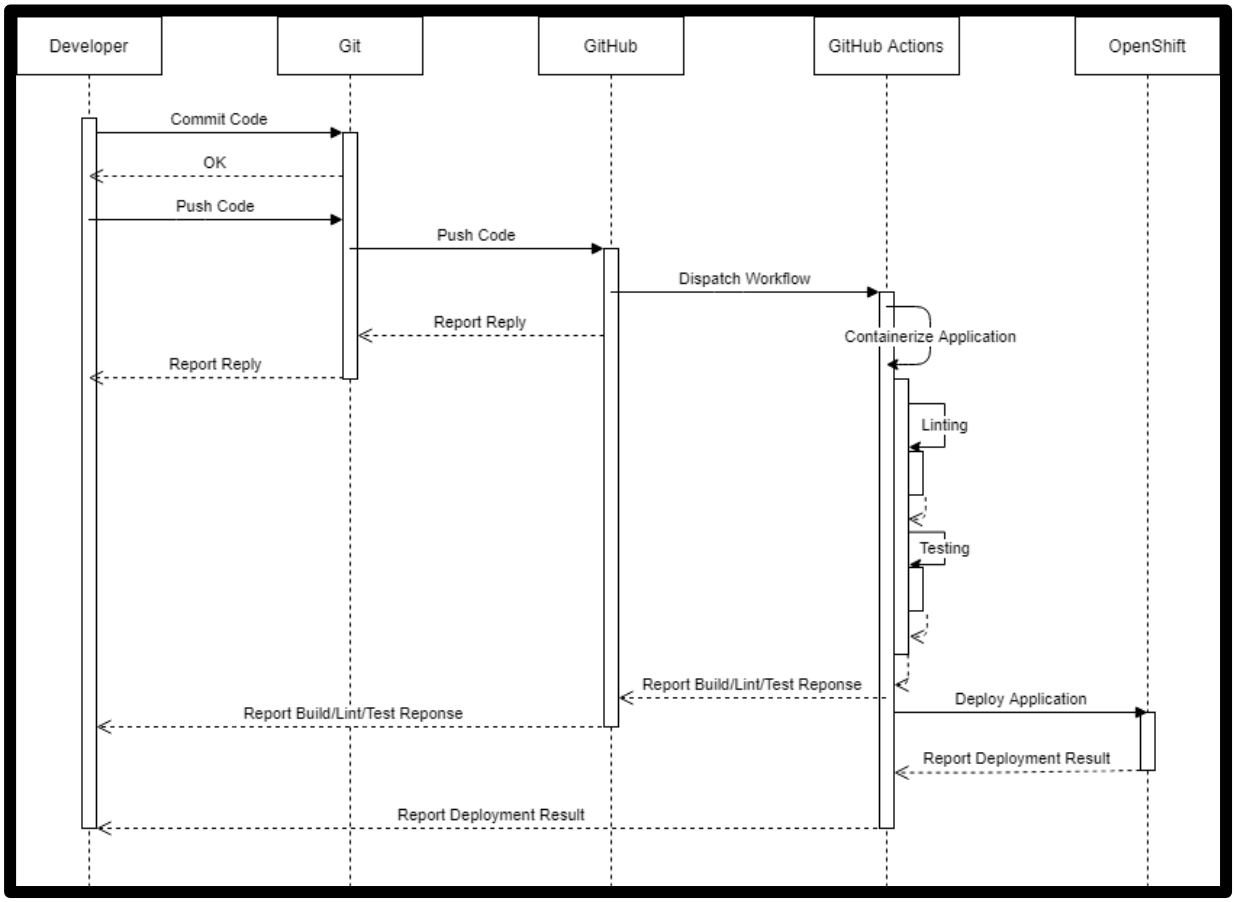
Architectural Decision	Jest
Problem Statement	Webdriver is not suitable for code coverage.
Motivations	Code coverage test.
Alternatives	Cypress, Mocha, Chai
Justification	Jest has the lowest technology overhead and highest capability
References	https://jestjs.io/

Software Design Specification

2.2.4. Class Diagrams

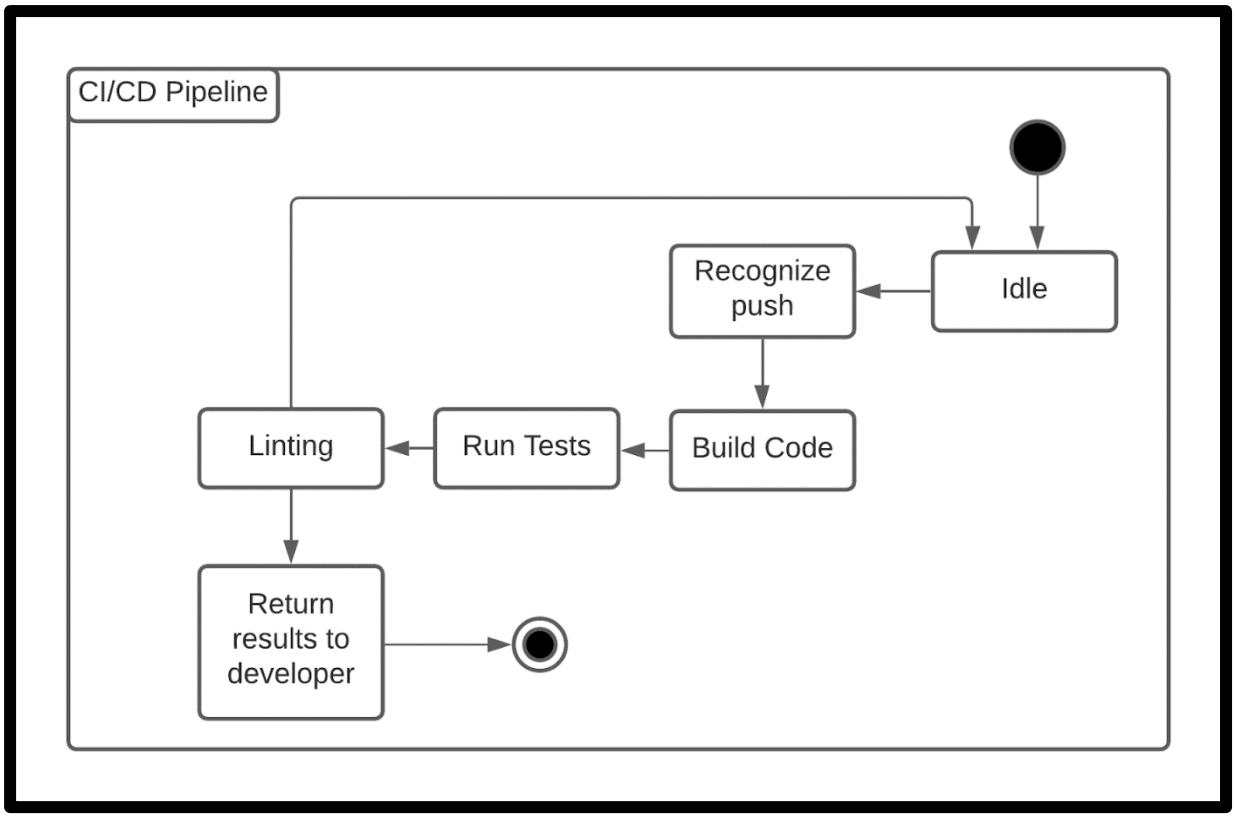


2.2.5. Sequence Diagrams



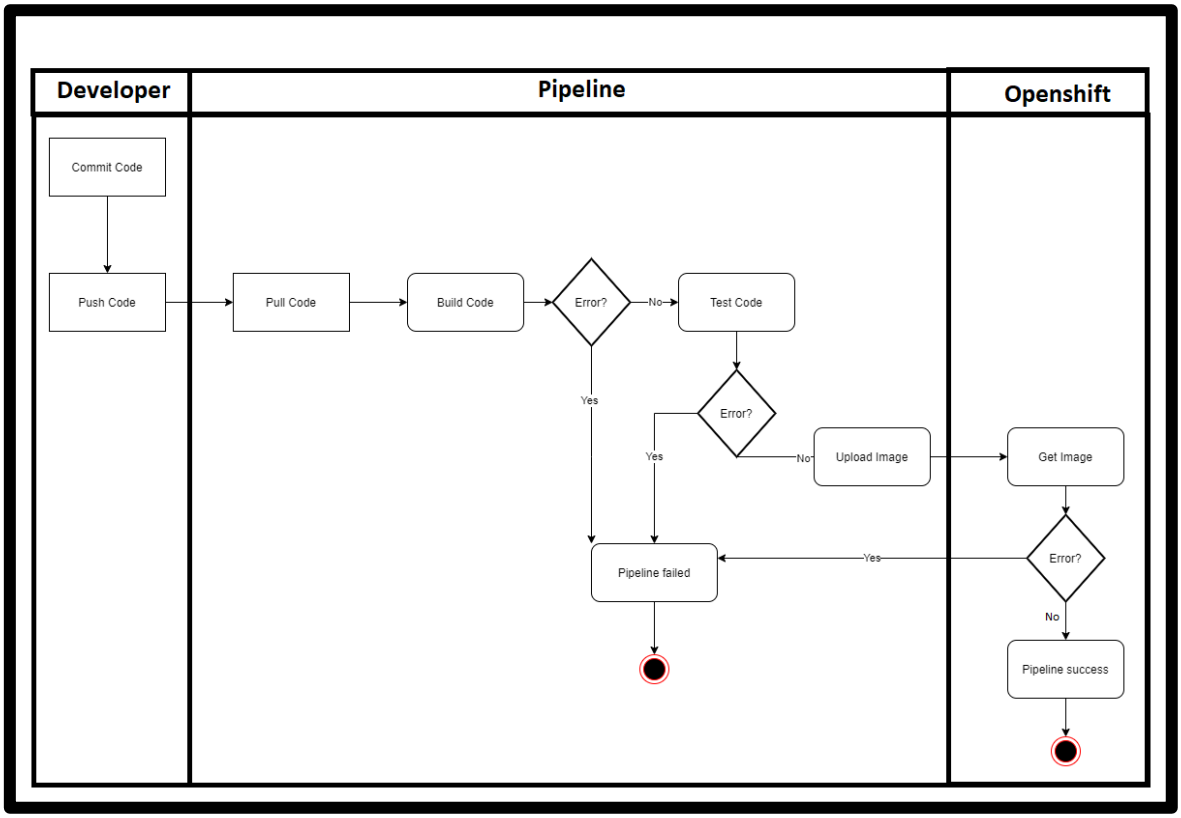
Software Design Specification

2.2.6. State Diagrams



2.2.7. Other relevant models

Dynamic Model



Software Design Specification

References

1. What is DevOps? - Amazon AWS
2. Podman
3. Red Hat OpenShift
4. COVID-19
5. Git
6. GitHub Actions
7. Selenium Webdriver
8. Jest
9. Eslint
10. Rapid API
11. Vue.js
12. Node.js

Other useful resources

- https://learn.openshift.com
- https://www.openshift.com/blog/deploying-to-openshift-using-github-actions
- https://redhatspain.com/openshift-pipelines/
- https://developers.redhat.com/blog/2019/02/21/podman-and-buildah-for-docker-users/
- https://developers.redhat.com/developer-sandbox