Bachelor’s thesis

Information and Communications Technology

2022

Vikas Singh

# DEVELOPING A CI/CD PIPELINE WITH GITLAB

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor's Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2022 | 56

Vikas Singh

# DEVELOPING A CI/CD PIPELINE WITH GITLAB

Developing software is a tedious process, especially when repetitive tasks are performed manually. This thesis discusses a better alternative to this approach, which is to automate these processes with a CI/CD pipeline. This thesis aims to enhance the application development process by integrating Continuous Integration (CI) and Continuous Deployment/Delivery (CD) methods with the application development phase. Additionally, the thesis attempts to determine whether, manual integration or auto integration is better and what kind of projects that can benefit from CI/CD.

For automation purposes, a CI/CD pipeline was implemented with GitLab which runs on a demo web-application that was created for this project. During implementation, technologies such as GitLab, React, NodeJS, Cypress, Jest, ESLint, Fly.io and Docker were used.

The outcome of this thesis is a successfully implemented CI/CD pipeline that auto builds, tests, and deploys the demo application, therefore, improving development process and increasing quality of the product.

These results are also going to be used by the Health Tech Lab TUAS, as reference case for implementing CI/CD in their work environment.

Keywords:

Continuous Integration, Continuous Development, Continuous Deployment, Docker, GitLab, React, NodeJS, Fly.io, Cypress, Jest, ESLint

# CONTENTS

## Figures

## Tables

## List of abbreviations

| | |
|---|---|
| Agile | A project development method |
| API | Application Programming Interface |
| AWS | Amazon Web Services |
| CD | Continuous Delivery/ Continuous Deployment |
| CI | Continuous Integration |
| CLI | Command Line Interface |
| DAST | Dynamic Application Security Testing |
| DevOps | Method of Development and Operations |
| DOM | Document Object Model |
| E2E | End to End |
| GUI | Graphical User Interface |
| JS | JavaScript |
| MERN | MongoDB, Express, React, NodeJS |
| MTTR | Mean Time to Resolution |
| MVC | Model View Controller |
| NPM | Node Package Manager |
| NPX | Node Package Execute |
| OS | Operating System |
| RASP | Runtime applications Self Protection |
| RAST | Runtime Application Security Testing |

| | |
|---|---|
| TDD | Test Driven Development |
| UI | User Interface |
| URL | Uniform Resource Locator |
| UX | User Experience |
| VCS | Version Control System |
| YAML | Yet Another Markup Language |

# 1 Introduction

The definition of the modern world would be imperfect without the mention of software development. If we break down today's era based on its defining forces, software development would be included amongst its driving forces, becoming a critical part of our world. Therefore, creating a never-ending demand for new ideas and innovation, backed up by a massive software industry focused on developing and maintaining these software's [1].

Due to this rising demand for IT products, new methods are required to revolutionize how software's are developed. Although we have moved far away from the traditional coding and developing styles, we are still far from the target goal. Because of this reason, a massive demand for IT professionals to automate the generic development of software codebases is seen in the job market all around the world. According to Venture beat survey, 46% of software developers see automation as a top priority [2].

In order to meet this rapidly growing demand, software companies have evolved different methodologies, such as Agile, Continuous Integration (CI), Continuous Delivery/Deployment (CD), Test Driven Development (TDD), DevOps, and more. Depending on the project size, different combinations of these methods make possible for the rapid rate development and delivery of projects. Furthermore, these tools and practices are becoming prominent as they considerably refine the scalability and automation of the software [3]. A survey shows that 65% of organizations use CI/CD to some extent [4], signifying the importance of CI/CD.

Continuous Integration (CI), Continuous Delivery, and Continuous Deployment (CD) are three frequently used methods for developing, automating, delivering, and deploying an application. CI automates testing and merging different developers' code, thus, becoming an alternative to manual and tedious processes. On the other hand, the role of CD is to take care of containerizing the application into a usable product [5]. With the extensive use of CI/CD in the software industry, keeping track of proper usage and implementation practices of these methods becomes essential.

The paper Implementation of CI/CD on automatic performance testing [6] shows the implementation of CI/CD for performance testing where the authors aim to improve the performance testing experience by automating the process using a CI/CD pipeline. Another study on CI/CD for Agile Software Projects [7] shows an implementation of CI/CD pipeline for projects using agile methodologies where the authors try to increase efficiency of agile methods with the use of a CI/CD pipeline. As a result, both these studies proves that a CI/CD pipeline, increases the quality of software development process.

This thesis focuses on integrating a CI/CD pipeline with software development in order to improve the application development and delivery process. To illustrate, a CI/CD pipeline with GitLab is implemented followed by the development of a React based demo web-application. This CI/CD pipeline auto builds, tests and deploys the demo application to a server.

The result of this implementation is hoped to be a working example case of a CI/CD pipeline with GitLab, with detailed overview of the pipeline's workflow. Additionally, the expected outcomes are hoped to be used to deduce answers to the research questions: whether manual integration or auto integration is better and which projects can benefit from CI/CD.

Lasty, this thesis contains step by step descriptive instructions of the implementation process, to be used by other developers as a reference example case of CI/CD with GitLab.

## 1.1 Implementation structure

This thesis comprises five chapters. Chapter one gives an introductory overview of the thesis idea and its importance as well as a structural outline of the implementation achieved, aiming to improve the reader's understanding.

Chapter two is focused on setting up the baseline of CI/CD, briefly defining the concept for the readers. This is followed by a discussion on the advantages and disadvantages of using CI/CD and the industrial challenges faced during implementation process. Lastly, focusing on workflow, and the significance of methods/tools like Agile, Cloud platform, and DevOps with CI/CD development.

Chapter three shows the implementation steps of the CI/CD pipeline and development steps of the demo web-application. It starts with defining a rundown of the workflow while mentioning modern techniques. Following it, there is an execution of the example case, with details including all the requirements, management, and development processes. In conclusion, the steps of deploying the test case with the GitLab are described.

Chapter four focuses on documenting the results, answering research question, and discussing challenges faced during the implementation and development.

Chapter five concludes this thesis.

# 2 Concept and Background of CI/CD

## 2.1 What is CI/CD?

CI/CD is a software development process that combines continuous integration and delivery. In other words, it’s an automated way to deploy code changes as they are made. In the context of DevOps, CI/CD refers to a set of practices, that help developers quickly build and test their applications to reduce risk and increase quality [8]. It also enables them to make quick changes without disrupting production environments.

The idea behind CI/CD is simple: Developers write code in small batches (often called “pipelines”), then run tests against each batch before merging the results into a master branch for deployment. The main goal here is to eliminate human errors by automating repetitive tasks, such as testing new builds on staging or QA environments before releasing them into production [8]. As a result, developers can spend more time writing code instead of managing infrastructure issues like deploying new versions of apps or fixing bugs caused by outdated configurations or broken dependencies between different services. This helps to ensure high-quality releases at a large scale while saving time and money on manual processes like deployments every night or every week due to outages caused by human error and malicious attacks.

### 2.1.1 CI

CI means continuous testing and integrating of work at regular intervals against pre-defined requirements such as: code quality, security vulnerabilities, performance, availability, etc. It's been around since the early days of computing and has undergone many evolutions over the years. The term CI first appeared in the book ‘Object Oriented Design with Applications written by Grady Booch in 1991 [9]. From then, it became an important part of the software development process, making integration of code easy and feasible.

Overall, it is a set of development practices that involves automating the building and testing of source code, every time changes are committed to version control systems like Git or Subversion [5]. It provides an additional level of assurance that all tests have passed before deployment. This means developers can be confident that their code works as expected when they push it live for others to use.

CI cycle can be broken down into 4 phases planning, code, build, and test [10].

Figure 1 shows the diagram of a CI life cycle.

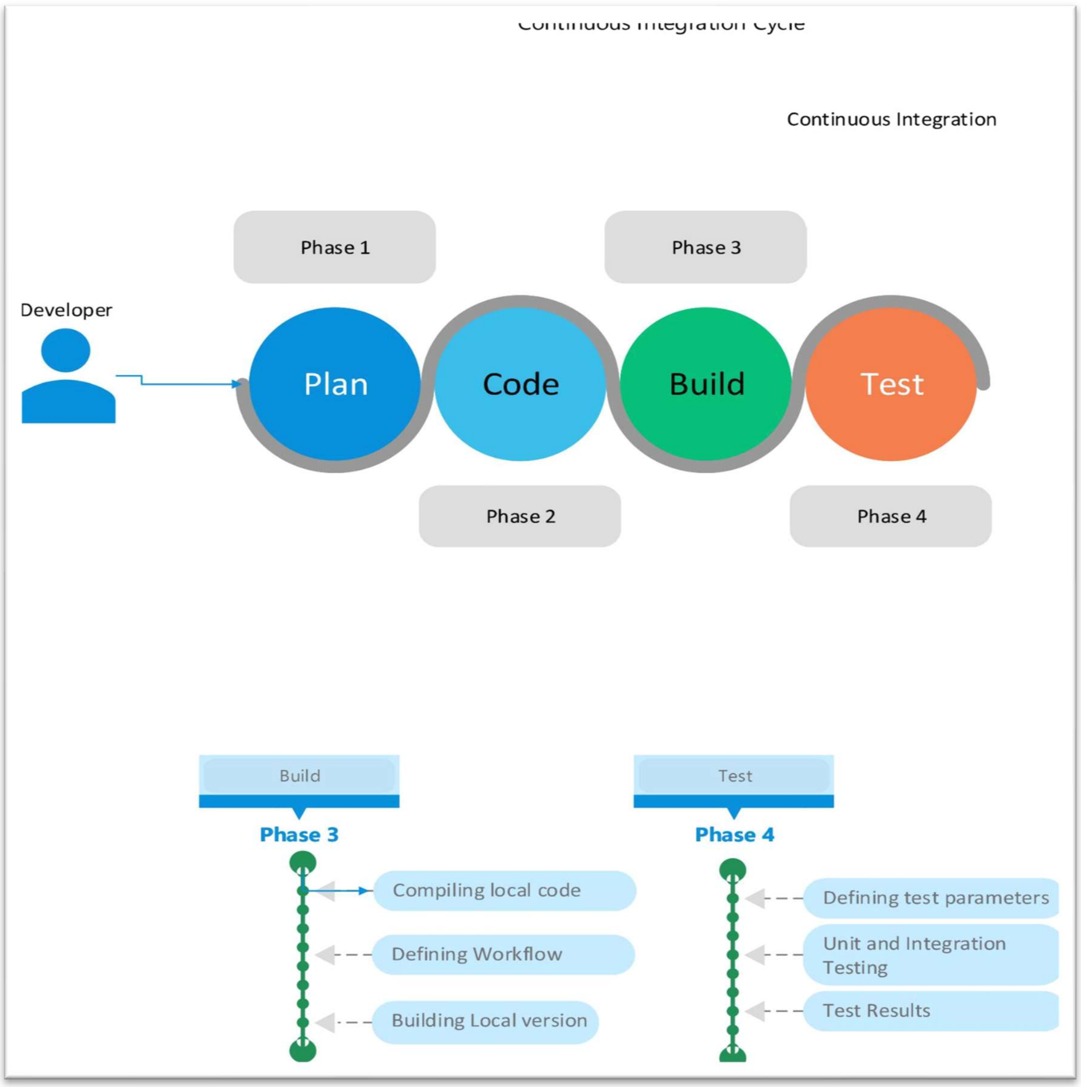


Figure 1. CI WorkFlow.

Although all 4 phases together make up a CI cycle, the last 2 are the most important ones as they are the bases of the CI pipeline and focus on automation of the execution process.

These phases can be defined as follows: -

- **Plan**

  The planning phase is the first step of a project. It includes all the necessary steps to define and prepare for the actual implementation of an idea or a product. It also includes the use of agile and scrum practices, defining the frequency of product micro-build releases [10].

- **Code**

  This phase is more focused on core coding/developing tasks. It includes making major decision like deciding languages, frameworks to be used, etc [10].

- **Build**

  The build phase consists of actions like code compiling, defining workflow, and building local code versions, making code ready for the testing phase [10].

- **Test**

  In the testing phase, integration of different developer’s work is done, after running the local builds through an automated testing pipeline. Which is accompanied by feedback and reviews on the work done [10].

These steps are defined differently from project to project, but the overall structure remains the same.

## 2.1.2 CD

Continuous Deployment or Continuous Delivery is a software delivery model that relies on automated, reliable processes and tools. It's all about delivering value to customers faster by continuously improving the quality of software through automation and process improvements [11]. It means that developers can release changes without waiting for code reviews, which makes it easier to get feedback from customers often and early. In turn, this helps to deliver higher-quality software more quickly. Since each change is tested thoroughly using an automated testing pipeline before being released, there are fewer bugs when the change gets deployed into production—less risk of introducing problems during deployment or downtime [6]. Because these deployments happen frequently, they become a vital part of an ongoing development cycle.

The life cycle of CD processes can be divided into four main phases- test, release, deploy, and operate [12].

Figure 2 shows the diagram of the CD cycle, with phase description.

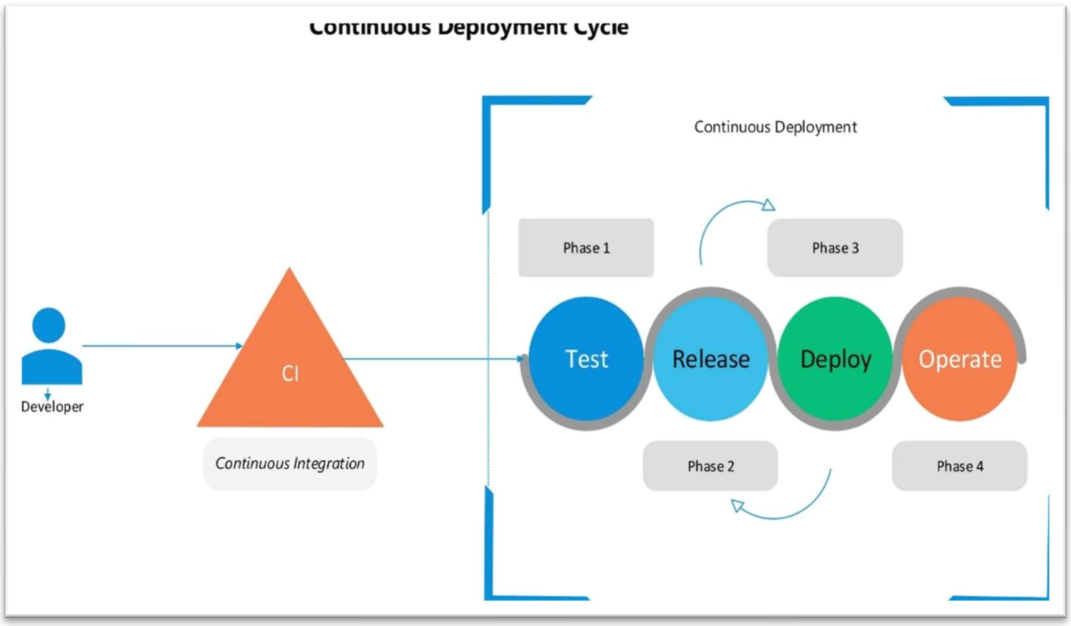


Figure 2. CD WorkFlow.

All the four phases defined are connected and executed in the same order. First, the CI methods are executed, resulting in a deployable product build version which is followed by the execution of CD methods.

The four phases of CD can be defined as follows: -

1. Test
   The test phase of CI/CD is comparable, with execution starting by performing different tests, including a unit test to an integrated test. Overall, this process aims to refine the code quality and create a runnable product version [12].

2. Release
   This phase focuses on documenting the committed changes and handling the release version of the product [12].

3. Deploy
   The released product from the last phase is ready to be deployed on a hosting server, presenting the final product, and ready to be used by the end-user [12].

4. Operate
   After the code is live, constant monitoring and productivity outcomes records are generated, helping further with update decisions [12].

### 2.1.3 CI/CD pipeline

In software engineering, a pipeline includes a sequence of processing factors (such as processes, threads, coroutines, functions), organized so that the output of every detail is the input of the next [13].

- Pipeline in CI/CD

The CI/CD pipeline is a set of processes used to manage and deploy software. If carried out properly, it decreases guided mistakes and beautifies the comment loops during the SDLC, permitting groups to supply smaller chunks of releases in a shorter time [14].

For the implementation of the pipeline, a YML-based script needs to be developed and deployed, containing all the required details about the run time environment and actions.

Figure 3 shows the structure of the CI/CD pipeline.

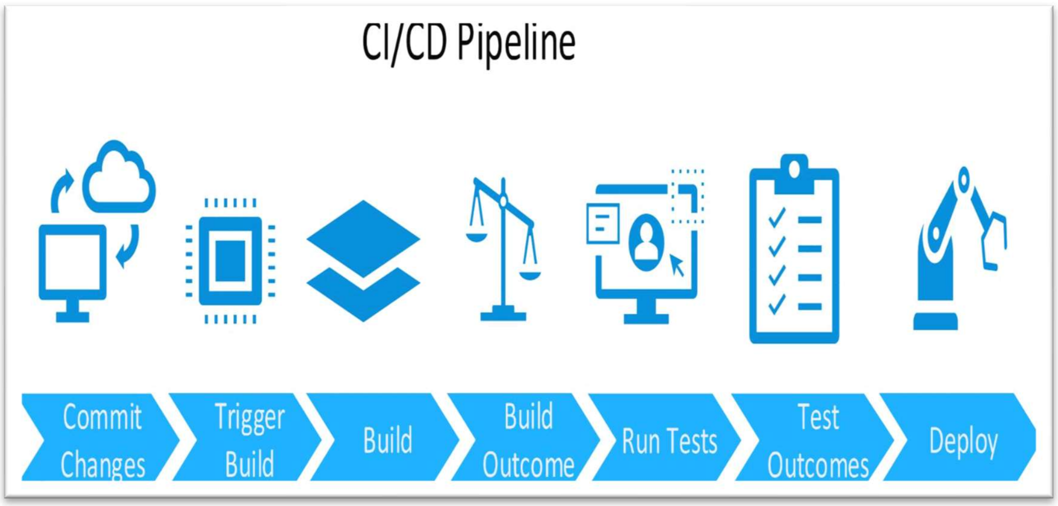


Figure 3. CI/CD Pipeline Structure.

The CI/CD pipeline is divided into four main phases Commit, Build, Test, and Deploy [13].

First, the developers decided on a version control environment like GitHub, GitLab, etc. organizing and structuring the base of the code. Further, the development is done either individually or with collaboration, depending on project to project [13]. Initiation of the pipeline can be customized based on many actions like every pull request (a command used to download the code from Version Control system to local machine), every push request (a command used to add local code changes to remote Version Control System), and so on.

Once the pipeline starts, the virtual environment command execution script (YML Script) is added to the VCS, which auto executes the build for the project [10].

After the build is complete, it undergoes predefined tests that are added to the code by the developers.

In the next step, the product goes for deployment after every check pass.

## 2.2 Advantages of CI/CD

Now that CI/CD pipeline is defined, it's the right time to mention the advantages of using CI/CD.

- Quality Code

  One technical gain of non-stop integration is that it makes possible to combine small portions of code at one time. These code adjustments are easier and simpler to deal with than big chunks of code - leaving less repair work for later [14].

- Reduced risk

  Rapid releases open a possibility for product managers and advertising specialists to interact with the improvement process. It opens opportunities to update improvements with customers often and early – with this approach method validation is possible earlier than investing. Meaning the development process can be put on hold or completely scraped based on early production results. [15]

- Effective Mean Time To Resolution (MTTR)
  MTTR measures the maintainability of repairable skills and indicates the typical time to restore broken functionality. Or simply a method to calculate the average time to resolve the issues [14].
  Because code commits and code improvements are minimal in CI/CD compared to other methods, it becomes easy to detect faults and isolate

them. Which further can be fixed fast, leading to a stable build, eventually reducing the developing time.

- Simplify Maintenance and Updates
  With the option of zero downtime updates, the process of software upgrading gets reduced by a huge amount, benefitting both user and customer in terms of time and money [15].

- Reduce Costs
  CI/CD is all about reducing production time and resources with automation of the generic processes. This process overall leads to fewer mistakes, on-time product delivery, and reduces the load on developers, which overall reduces the production costs [15].

## 2.3 Challenges of implementing CI/CD pipeline

The use of CI/CD in the software development industry has been increasing steadily. Adopting CI/CD methods in a work environment can become challenging, as setting up a CI/CD pipeline is time consuming and requires great amount of work [16]. So, it becomes important to understand the common challenges faced in implementing a CI/CD pipeline.

- Environment Limitations
  Testing any kind of software is expensive, and software development teams, in particular testing team, usually find themselves restricted by the number of available resources (tools and technologies) due to a limited budget, eventually hindering the testing process. In such scenarios, developers are left with only one option that is to share the CI/CD pipeline run environment amongst several projects [17]. Shared run environment leads to a poorly deployed CI/CD pipeline, as scenarios like parallel testing (multiple tests are performed parallelly and not one by one) failures could be seen more often.

- Security Tool Integration

  Securing a software is necessary for user privacy and preventing data leaks. Majority of the tools having a Command Line Interface (CLI) are mergeable into a CI/CD pipeline [18]. But it becomes challenging to correctly position security tools in a CI/CD pipeline, as a single misplacement might cause a breakdown in the pipeline and make error handling difficult [16].

  The results in Figure 4, of the survey conducted by 451 Research on the topic of 'What are the most significant application security testing challenges inherent in CI/CD workflow?' shows that 61% of the developer's main concern was lack of automated and integrated security testing tools in a CI/CD workflow.

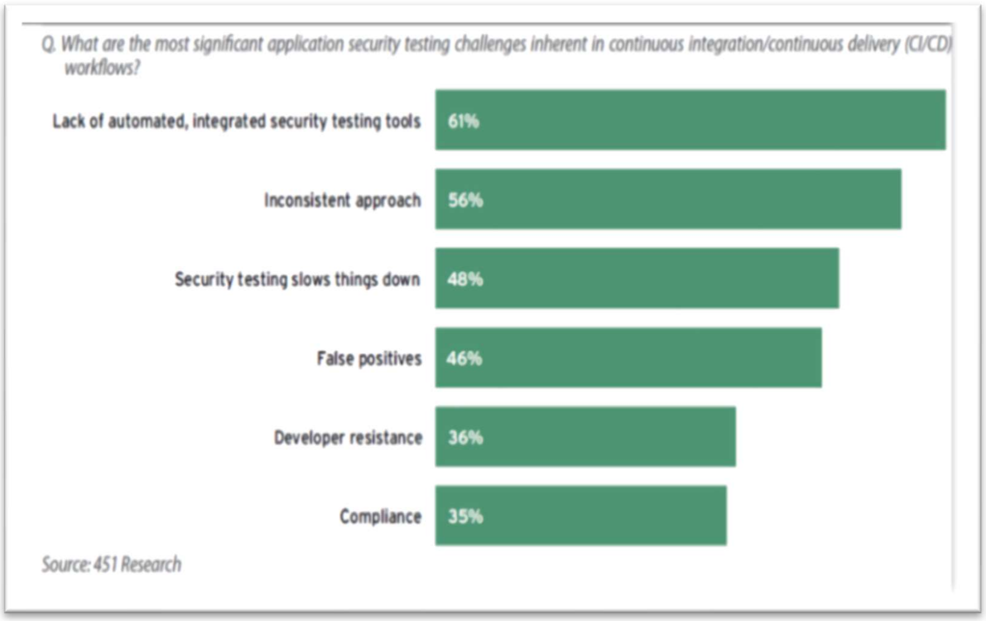


Figure 4. Most significant security challenge CI/CD [16].

- Inefficient pipeline deployment

  One of the large issues viewed in the software industry is the lack of training in CI/CD pipeline building [17]. Although, a properly developed pipeline makes life easy for a developer, still building it is a complex and time-consuming process.

- Implementation of ongoing/old projects
  Changing the flow of an ongoing project is met by a lot of challenges, especially if the base methodologies of the product need to be redone [18]. Any solution to resolve this requires a lot of resources and manpower, specifically with older projects containing thousands of code lines. Thus, becoming a challenge for developers.

## 2.4 CI/CD tools

The tools used in CI/CD processes are important to ensure the completion of work on time. These tools automate the build and deployment processes for different projects by integrating them with a CI/CD pipeline. It also allows developers to create automated tests before they deploy their code into the production environment. After successful execution of a CI/CD pipeline, it will generate reports about the status of the project, making tracking application progress easier. This way monitoring the entire project's performance at any given point in time is accessible without any involvement from IT team members or managers.  Similarly, there are many other kinds of tasks that require specific development tools, and it is important to use the proper tool for each task.

There are many Versions Control System (VCS) that comes with integrated CI/CD pipeline functionality. Some of the famous one's are Jenkins, GitLab, GitHub, Bitbucket and Circle CI. For this thesis project, GitLab is used, keeping in mind its compatibility with web-applications and its use by the Health Tech Lab TUAS.

- GitLab
  Another famous and open-source VCS is GitLab, developed in 2011[19].It is used to build applications in any language and supports Docker

containers, including Ruby on Rails (RoR) apps. GitLab CI/CD also has a plugin architecture that extends its functionality with custom plugins. Around 7.7% of software development companies worldwide use GitLab. For instance, Cisco, Intel, The Walt Disney Studio, etc. use GitLab [20].

Tools used for testing in a CI/CD pipeline are: -

- For testing Application Programming Interface (API)
  API handles the communication between the application and the server. Over the years, we have seen a dense network development of this communication process. So, executing proper testing is becoming significantly important. Tools like SoapUI, 3scale (Red Hat's), swagger, etc. are a few tools that can easily be embedded with a CI/CD pipeline [21].

- For testing User Interface (UI) / User Experience (UX)
  UI/UX testing is related to user testing, one of the dark areas where chances of code breaking are high. By defining a strong Graphical User Interface (GUI) test, code breaking issues on the side of the user are isolated. Some of the step-by-step testing tools are Selenium, Appium, and Cypress [22].

- For Application Security
  In recent years, cyber security attacks have launched concerns in developer society, pushing focus towards application security [23]. Security tools can be embedded into the CI/CD pipeline and run-instructions can be added to the auto-run script. Some of the well-known security tools are Dynamic Application Security Testing (DAST) tool, Runtime Application Security Testing (RAST) tool, Runtime Applications Self Protection (RASP) tool, and Amazon Web Services (AWS) security tool [24].

## 2.5 Agile method and CI/CD

Agile is a project management methodology, which promotes breaking down of projects in small phases and development on at a time. In short, a collection of techniques, values, and principles that help teams deliver working software frequently [25]. They have three key characteristics: customer-driven, iterative (emphasizing the importance of small batches of work), and flexible (encouraging collaboration over strict adherence to plans). These methods are considered amongst the best ways of developing software and are widely used in the software industries.

Agile and CI/CD methods often can be confused as one, but they are two different processes with cross compatibility. When used together, they are called Agile CI/CD [26], combined enabling organizations to deliver the software's in an agile manner.

- Difference between agile and CI/CD
  CI/CD methods are more focused on automation of the application development lifecycle, making possible rapid development and deployment. Whereas agile is more focused on developing processes to deliver products in small chunks on regular basis [27].

- Importance of agile CI/CD
  These two methods complement each other in many ways. Specially CI, with its rapid development, integration, and testing acts as a base line for Agile method. Another focus of Agile is small incremental delivery, which is also a highlight of a CI/CD pipeline [26].

## 2.6 CI/CD and DevOps

DevOps is a set of practices that helps to develop and deploy software faster. It is the merger of development and operations, where developers are given more autonomy over their work while the operational staff takes on a greater share of responsibility for the business [28]. DevOps aims to improve software delivery speed by reducing the time from idea to launch. This means early feedback processing, which allows developers to make changes before they become too costly or cumbersome. In addition, DevOps allows the automation of processes so that they run smoothly without human intervention. All of this helps with continuous improvement at different levels within organizations, from planning to deployment and maintenance [29].

To summarize, DevOps is a superset of CI/CD [28]. Meaning, DevOps work-cycle is similar to a CI/CD work-cycle. Figure 5 shows a DevOps roadmap cycle.

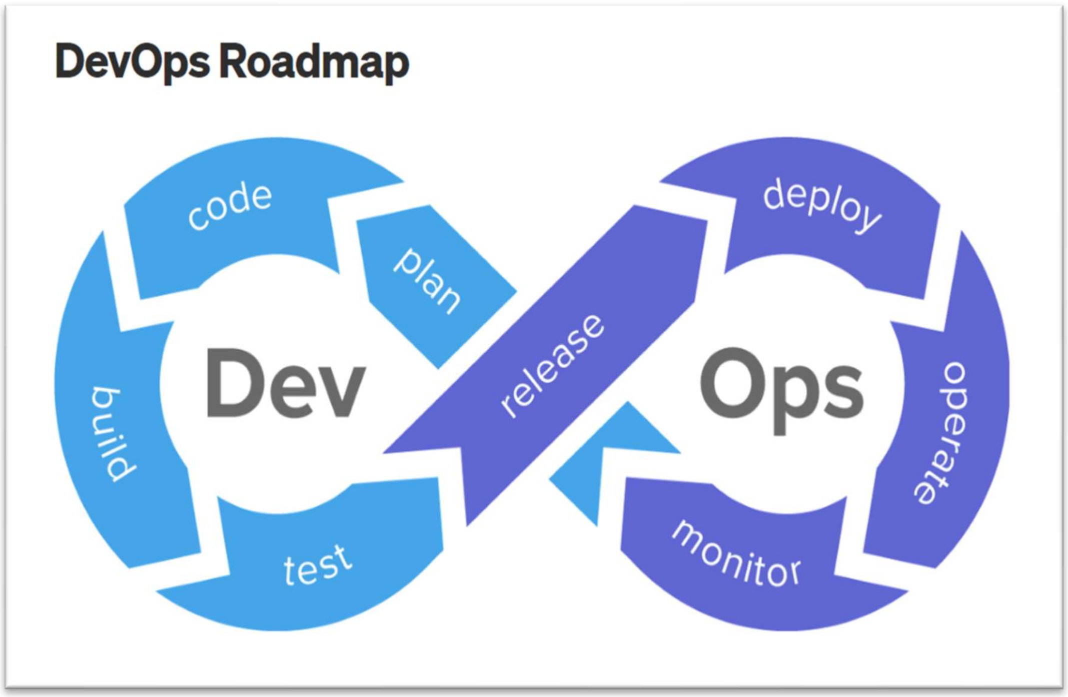


Figure 5. DevOps cycle [18].

Due to similar looking work-cycles, it becomes important to differentiate between DevOps and CI/CD. Table 1 shows the differences between DevOps and CI/CD.

Table 1. Differences between DevOps and CI/CD [27].

| **DevOps** | **CI/CD** |
|---|---|
| 1. Set of processes, ideas, and technologies, that combine the process of development and operations | 1. Set of implementation practices, to enable rapid integration, development, and deployment |
| 2. Focuses on reducing the communication/collaboration gap between developers and operators | 2. Focuses on product development |
| 3. For implementation whole office environment needs to be changed according to DevOps practices | 3. Tools like GitLab, CircleCI, etc are used for the implementation without changing the environment |
| 4. The main stages are: - CI/CD, Continuous Testing, Continuous development, Continuous monitoring, and Continuous feedback | 4. The stages are: - commit, build, test, and deploy |

In terms of software development, the use of CI/CD goes hand in hand with DevOps. When used together, they improve the development workflow and reduce communication gaps between developers, company, and customers.

## 2.7 Trends and growth of CI/CD

CI/CD pipeline was released in 2011, since then it has revolutionized the software developing industry, with its new and effective ways of developing software’s [30]. As technology is constantly improving and the focus on automation is rising, many software companies have already moved towards implementing CI/CD practice in their work environment, at least to some extent.

A survey by the developer nation shows CI/CD adoption treads by mobile, web and desktop developers, from the year 2016-2019. Indicating a stable to constant growth quarterly. Figure 6 shows the results of the survey.

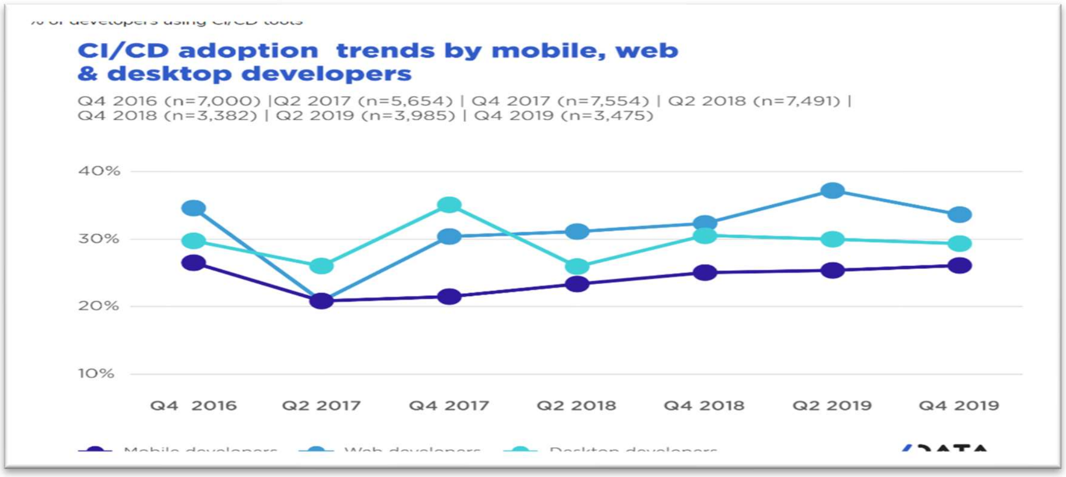


Figure 6. Trend of CI/CD in application development industry [31].

Not only this, but the DevOps market, in general, has increased in the last few years. According to Global Market Insight, ‘DevOps Market size exceeded USD 7 billion in 2021 and is expected to register over 20% gains from 2022 to 2028’ [32]

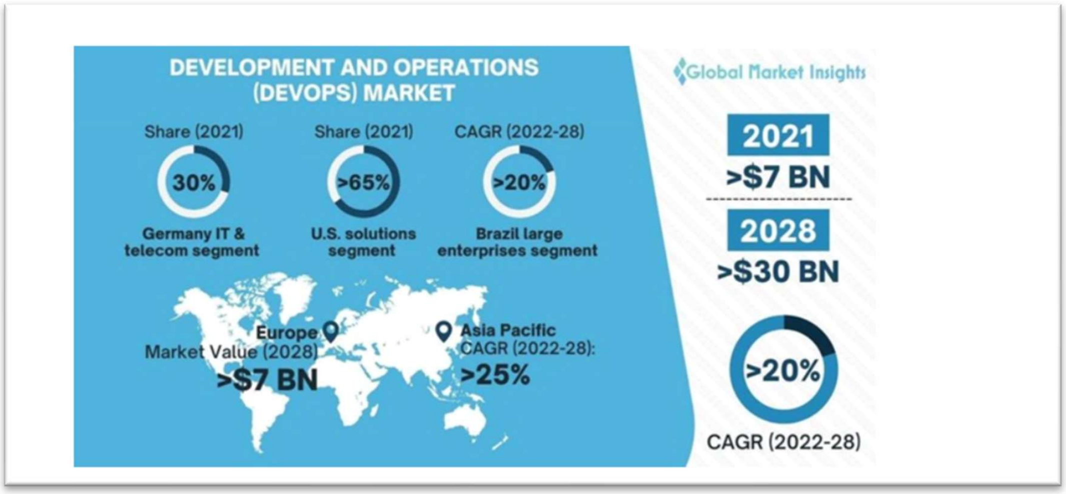


Figure 7. Growth of DevOps market [32].

All these trends, points out the importance of CI/CD in modern technology-oriented world.

# 3 Implementing a CI/CD pipeline and developing Our Travel Gallery (demo web-application)

## 3.1 Workflow design of development and implementation

Now that the definition of CI/CD and its usage in the software development industry is defined, the next stage is to create a brief execution plan followed by a step-by-step execution.

The workflow of development and implementation comprises of four stages.

1. Developing Our Travel Gallery web-application (demo application) with React, NodeJS, ExpressJS and MongoDB (MERN stack). The objective of this stage is to provide a demo, planned and designed for CI/CD pipeline implementation. Completion of this stage will lead to a fully functional Our Travel Gallery application.

2. Testing Our Travel Gallery application, which is achieved by defining unit, integration, and end-to-end (E2E). Only the most common test types are defined in order to maintain the simplicity of implementation.

3. Setting up GitLab VCS for Our Trave Gallery application.

4. Setting up a CI/CD pipeline that includes creating an executable Yet Another Markup Language (YAML) script, containing detailed instructions for integration and deployment processes.

## 3.2 The demo application development steps

This section contains a complete process breakdown of the workflow from development to deployment of Our Travel Gallery web-application.

### 3.2.1 Defining Our Travel Gallery application

This CI/CD pipeline implementation includes the development of a web-application with MERN stack. Our Travel Gallery is the name of the application developed during the process. It’s a web-application that acts as a general platform to create a collection of travel images for an individual or a group. Aiming to solve problems of image management faced by groups of friends/family/colleagues which is storing part of memory as images. The application functionality is kept at a minimum, focusing more on build quality and less on quantity. This demo application covers the most common scenarios many software developers face while implementing a CI/CD pipeline.

The ways this application brings people together which are described below.

- Users can create an account and log in as well as log out to Our Travel Gallery.
- Options of forming groups/private space.
- Users can upload a collage of images displayed in a general view area only accessible/visible to people with restricted access.
- Images can be downloaded, edited, or deleted.
- Images are accessible 24/7.

The aim of development at this point is not to develop the entire application but to start with the first version and use it as a test-case prototype to set up CI/CD pipeline infrastructure. Furthermore, promoting DevOps practices from start to finish in a work environment.

### 3.2.2 Development stages of Our Travel Gallery

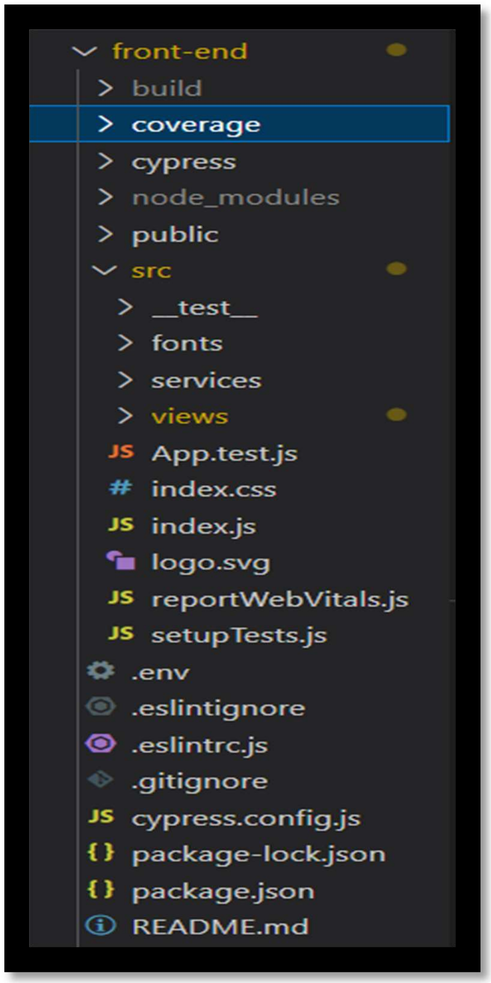
As mentioned in the previous section, the focus of development at this point is to get started with the first version of the demo application. The functionalities focused on are: -

- Registration/Login/Logout
- Availability of the gallery to both registered and non-registered users
- Logged-in users can add photo collages

There are three parts to this application which described below.

1. **Front-end**

The front-end of the application is developed using React, which is a JavaScript library. React provides integration compatibility with several tools, bringing the development process to the next level of flexibility and ease. React supports browser-based rendering, reducing time for heavy data loading, which improving application response time [34]. Our demo application is initialized using the 'create-react-app' command, which auto-creates project structure and downloads the basic necessary packages to start React applications.



```
front-end
  build
  coverage
  cypress
  node_modules
  public
  src
    __test__
    fonts
    services
    views
    App.test.js
    index.css
    index.js
    logo.svg
    reportWebVitals.js
    setupTests.js
  .env
  .eslintignore
  .eslintrc.js
  .gitignore
  cypress.config.js
  package-lock.json
  package.json
  README.md
```

Figure 8. Front-end Folder Structure for the demo application.

The file structure of the front-end application is shown in the figure 8, with the main component breakdown as follows: -

- The build directory is a compilation of the front-end. It is a sum-up of the client side for the production build.
- Coverage and Cypress directories are for testing, discussed in the testing section 3.2.5.
- Node_modules contain all the downloaded libraries required by the project.
- The public folder consists of static HTML, CSS, and JavaScript files that make up the outermost structure of the application.
- The src directory contains all the user-developed view files, styling sheets, scripts, and test files. App.test.js is a subfile of this directory where all the unit tests for the front-end are written.
- .env file handles environment variables
- .eslintrc.js and .eslintignore are for lint testing, which will be discussed later
- .gitignore contains instructions for the version control
- cypress.config.js contains instructions for Cypress
- package-lock.json locks the installed library versions, producing the same result every time the app is initialized
- package.json is the heart of the project, containing different kinds of metadata like: - installed dependencies, script run commands, entry points, and many more.
- Readme.md have instructions about setting up the application.

Next comes the working structure of the application with respect to the data travel structure: -

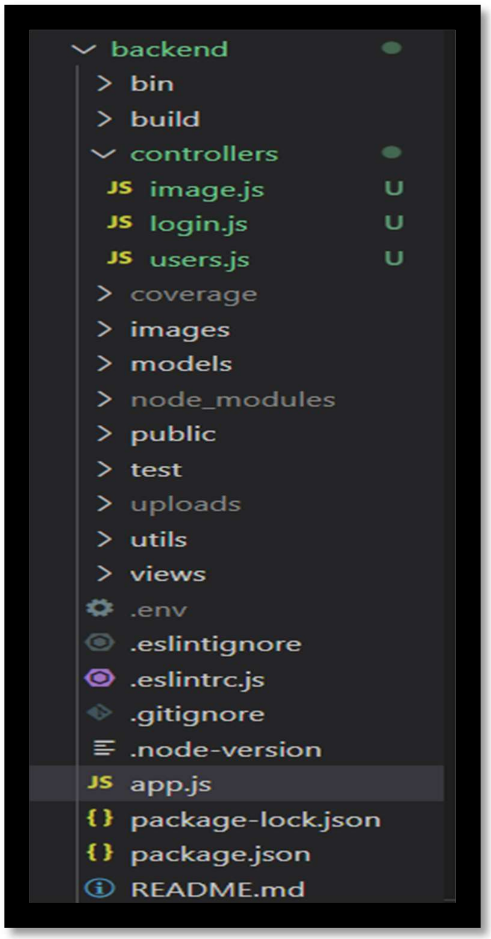
- User navigation consists of 3 pages: Gallery, Add, and login/logout.
- The gallery is a public view area with a collage display for the users, accessible by both registered and non-registered.

  Add page has the option of adding a photo collage, which is only available for logged-in users.

## 2. Back-end

The server-side of the Our Travel Gallery application is built using Node.js, an open-source JavaScript-based runtime environment built on Chrome's V8 JS engine, and Express.js, an open-source web application framework that allows developers to build robust web applications quickly and easily [34]. Together, making it possible to develop applications entirely based on JavaScript.

Node 16.13.0 and Node Package Manager (NPM) 8.12.1 are the versions used for developing this project. To initiate, 'npm init' command is used, which auto-creates the package.json and package-lock.json and sets up the entry points.



```
backend
  bin
  build
  controllers
    image.js          U
    login.js          U
    users.js          U
  coverage
  images
  models
  node_modules
  public
  test
  uploads
  utils
  views
  .env
  .eslintignore
  .eslintrc.js
  .gitignore
  .node-version
  app.js
  package-lock.json
  package.json
  README.md
```

Figure 9. Back-end folder structure of the demo application.

The back-end file structure shown in the figure follows an Model View Controller (MVC) architecture. The main structure breakdown is as follows: -

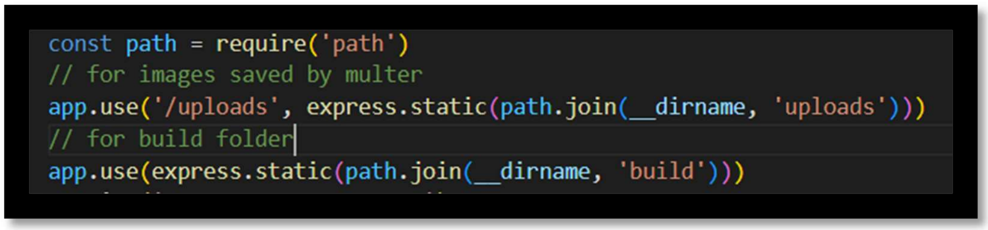
- The bin directory contains the www file, an alternative to index.js, which acts as a setup manual for the node run environment.
- The build folder is a production version of the front-end.
- The controller directory contains all the functions that control the API endpoints with logic.
- Coverage, test, and image directories are for testing (discussed later).
- The model directory contains database models, helpers, and handlers.
- The node_module directory is the same as in the front-end.
- Public directory contains static files.
- The uploads folder contains all the images uploaded by the users to the gallery.
- The Utils folder consists of middleware and logger functions which acts as helpers for controller procedures.
- env file contains environment variables.
- .eslintrc.js and .eslintignore are for the lint test.
- App.js is the entry point into the application, linking all the files into a runnable program.

  Package.json and Package-lock.json contain all the metadata for the node server.

On the back-end, there are three main routes (they are navigation options available for users, usually referring to amount of page available in an application), each one for handling user registration, user login, and image get/post request. Additionally, there are middleware functions (middleware function are helper function, with access to API's request and response) for user login checks and request/error loggers.

The Multer tool is used for image-handling purposes. It creates static disk storage for storing user uploaded images. Only the location of these images is saved in the database. The uploads folder is provided as a static path directory to the express server, which enables accessing the images using Unified Resource Locator (URL) of type- 'http://localhost:<port>/<image address in uploads folder>'.

Figure 10. shows the commands to connect the uploads directory as a static path, appended to the app.js file, which auto allocates static files from the build folder to URL endpoints.



```
const path = require('path')
// for images saved by multer
app.use('/uploads', express.static(path.join(__dirname, 'uploads')))
// for build folder
app.use(express.static(path.join(__dirname, 'build')))
```

Figure 10. Adding static path to back-end.

3. **Database**

MongoDB is used for storing data of the Our Travel Gallery application. It's an open-source database that provides free data storage for low-level usage and avails premium membership options for high-level usage. Additionally, it’s a NoSQL database (NoSQL database is not build around rows and columns) used to store and retrieve data in the same way as relational databases (built around rows and columns) which makes data manipulation, storing and transferring easy [35].

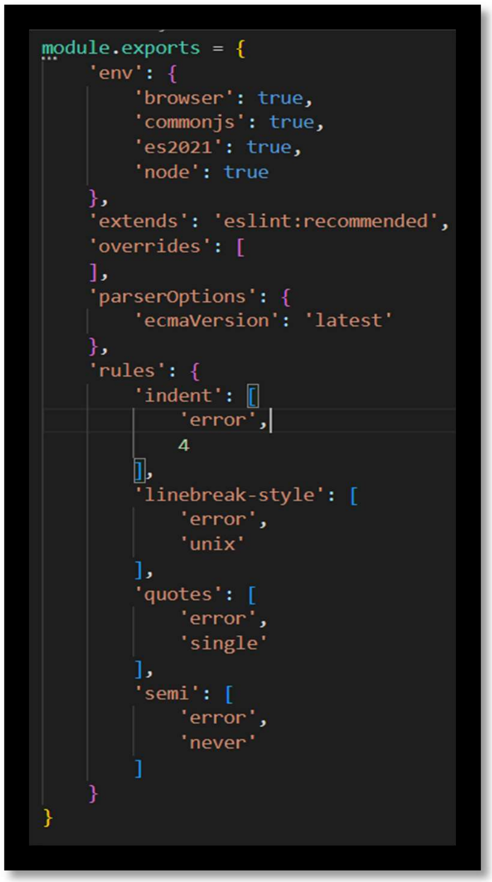
### 3.2.3 Testing Our Travel Gallery

Testing is the process of checking, verifying, and validating applications to fix/remove possible breakdowns and errors. There are several ways to test an application, from structural tests to end-user experience tests, but they can be filtered out based on the application's functionality and requirements. List of tests performed on this website are: -

## Lint test

Eslint is a tool for enforcing consistent JavaScript styles and fixing issues like problematic code patterns or code inconsistency [36]. Its setup consists of a '. eslintrc.js' file with a defined set of rules modifiable according to personal needs. Additionally, it's compatible with the CI pipeline.

Eslint is configured on both the front-end and back-end of Our Travel Gallery application, installed as a developer dependency using the command 'npm install –save-dev eslint'. Developer dependency is an option provided by NodeJS in the package.json file. Tools required only for the development version, are saved as devDependencies to the package.json file. Once the application goes for production, these tools are ignored and therefore are not downloaded to the servers, thereby saving space, and rendering time.



```
module.exports = {
    'env': {
        'browser': true,
        'commonjs': true,
        'es2021': true,
        'node': true
    },
    'extends': 'eslint:recommended',
    'overrides': [
    ],
    'parserOptions': {
        'ecmaVersion': 'latest'
    },
    'rules': {
        'indent': [
            'error',
            4
        ],
        'linebreak-style': [
            'error',
            'unix'
        ],
        'quotes': [
            'error',
            'single'
        ],
        'semi': [
            'error',
            'never'
        ]
    }
}
```

Figure 11. Eslint file data.

To initiate eslint for a project, the command ‘npx eslint –init’ is used. Once initiated, it launches a questionnaire regarding the required configuration and creates a ‘.eslintrc.js’ file based on input answers.

The next step is to add a script run command for linting. The command from Prgoram 1, is added to the package.json file under 'scripts'.

Program 1. ESLint run command

```
"eslint": "eslint './**/*.{js,jsx}'"
```

Now, the ‘npm run eslint’ command starts the linting test on the project. File/folder names mentioned in the ‘.eslintignore.js’ file don't undergo a linting test.

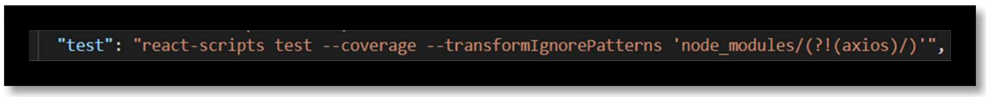
**Unit test and Integration test**

Jest is a testing framework for JavaScript that helps write code that is easy to read, reason about, and maintain. It directly interacts with the Document Object Model (DOM) tree to check for code changes and compares the results using DOM snapshots. Chai commands are integrated with Jest, giving access to Expect and Should methods, making change comparison easy [37].

Jest comes integrated with React library, meaning a separate installation is required only for the back-end folder.

- Jest with front-end - ‘@testing-library/jest-dom’,‘@testing-library/user-event’,‘@testing-library/jest-dom’ and ‘babel-jest’ are some of the NPM packages used with Jest for testing our application.

  Figure 12 command is added to the package.json file under ‘scripts’.

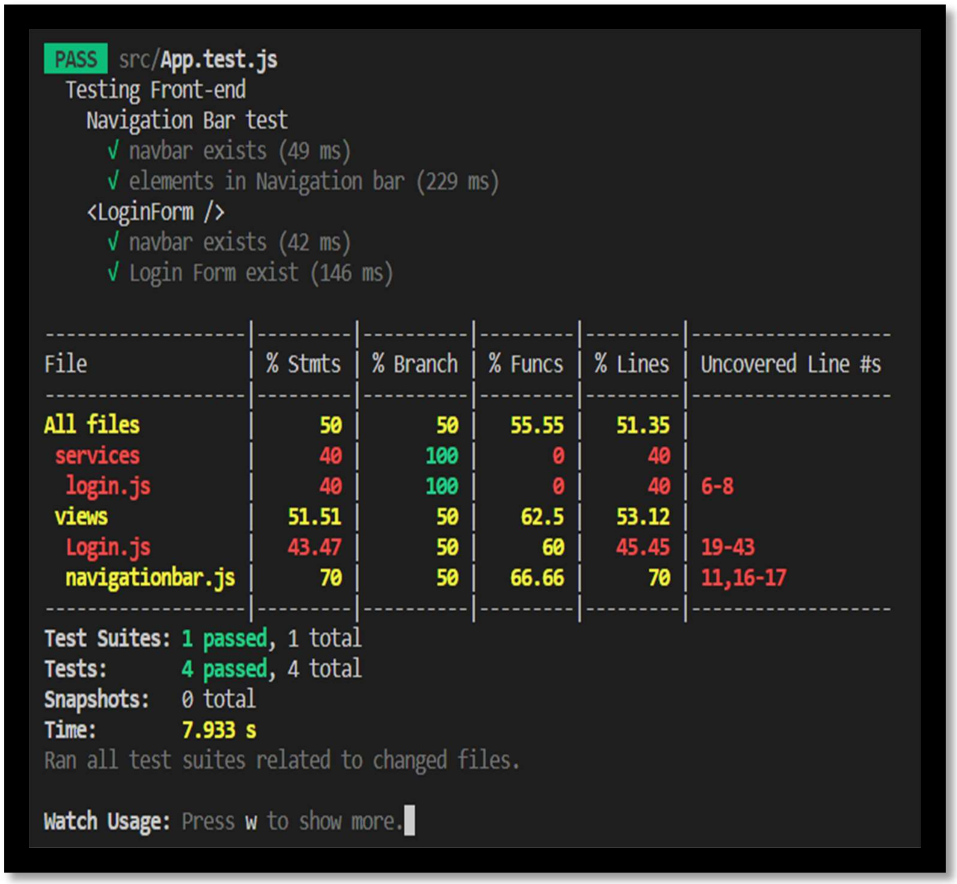


```
"test": "react-scripts test --coverage --transformIgnorePatterns 'node_modules/(?!(axios)/)'",
```

Figure 12. Test run command for demo application.

Executing the command ‘npm run test’ in the terminal starts the testing. It runs on all the files with the extension ‘.test.js’, and checks for defined tests in them. ‘—coverage’ creates the coverage folder, which has data on test coverage. Furthermore, test commands ignore the node_module directory to reduce execution time. So, any package required by the test cases is added using “—transformIgnorePatterns ‘nodemodules/(?!(<packagename)/)’”.

Figure 13 shows the testing result from Our Travel Gallery front-end.



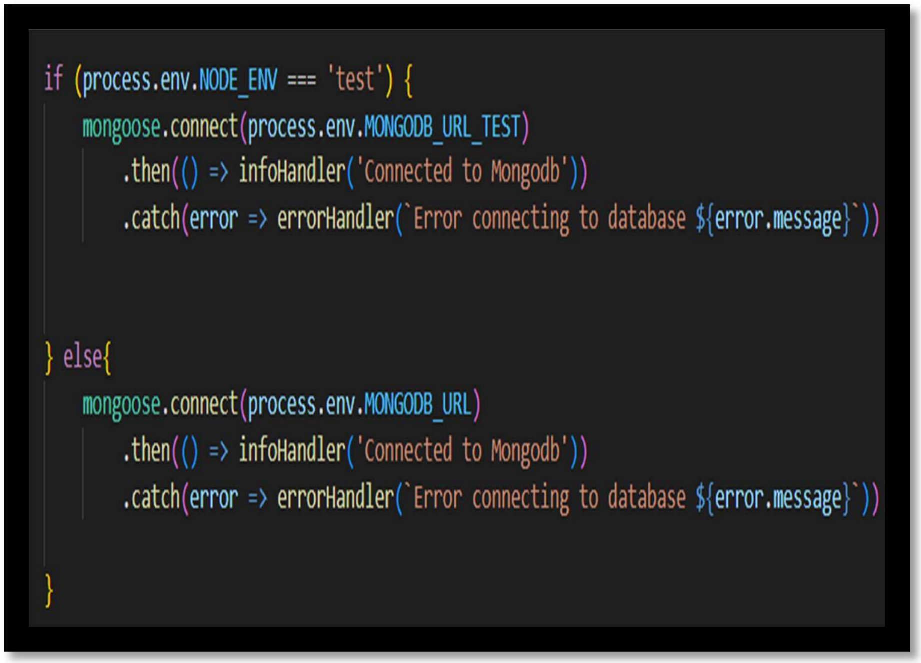
```
PASS  src/App.test.js
  Testing Front-end
    Navigation Bar test
      √ navbar exists (49 ms)
      √ elements in Navigation bar (229 ms)
    <LoginForm />
      √ navbar exists (42 ms)
      √ Login Form exist (146 ms)

-------------------|---------|----------|---------|---------|-------------------
File               | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------|---------|----------|---------|---------|-------------------
All files          |      50 |       50 |   55.55 |   51.35 |
 services          |      40 |      100 |       0 |      40 |
  login.js         |      40 |      100 |       0 |      40 | 6-8
 views             |   51.51 |       50 |    62.5 |   53.12 |
  Login.js         |   43.47 |       50 |      60 |   45.45 | 19-43
  navigationbar.js |      70 |       50 |   66.66 |      70 | 11,16-17
-------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        7.933 s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Figure 13. Test Results from front-end of demo application.

- Jest with back-end

Jest does not come integrated with NodeJS. It is installed separately as a developer dependency with the command ‘npm install –save-dev jest’. Furthermore, to test HTTP requests, the supertest package is installed.

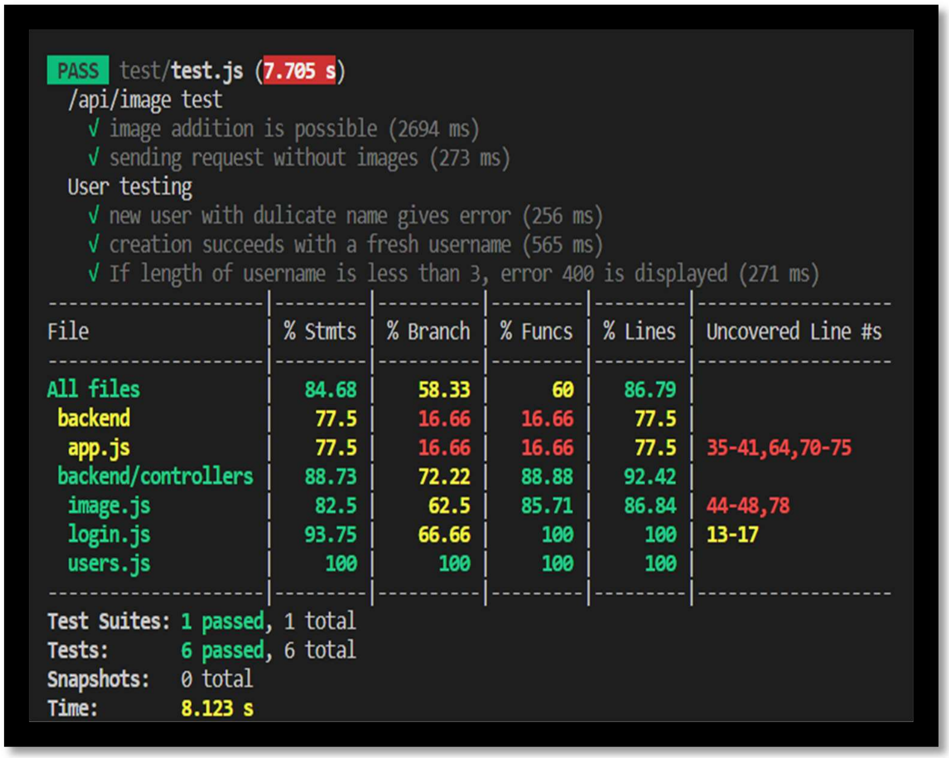
Testing with the back-end is similar to client-side testing, with the difference being in the setup, where back-end testing is more focused on API testing. Additionally, a separate database is used when testing the back-end. The image below shows two mongoose database connections, selected depending on the env variable from the run commands.



```
if (process.env.NODE_ENV === 'test') {
    mongoose.connect(process.env.MONGODB_URL_TEST)
        .then(() => infoHandler('Connected to Mongodb'))
        .catch(error => errorHandler(`Error connecting to database ${error.message}`))


} else{
    mongoose.connect(process.env.MONGODB_URL)
        .then(() => infoHandler('Connected to Mongodb'))
        .catch(error => errorHandler(`Error connecting to database ${error.message}`))

}
```

Figure 14. Mongoose conditional Connection.

There are six tests, that are performed in back-end. Figure 15 shows results of the test.



```
PASS  test/test.js (7.705 s)
  /api/image test
    √ image addition is possible (2694 ms)
    √ sending request without images (273 ms)
  User testing
    √ new user with dulicate name gives error (256 ms)
    √ creation succeeds with a fresh username (565 ms)
    √ If length of username is less than 3, error 400 is displayed (271 ms)
---------------------|---------|----------|---------|---------|-------------------
File                 | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------|---------|----------|---------|---------|-------------------
All files            |   84.68 |    58.33 |      60 |   86.79 |
 backend             |    77.5 |    16.66 |   16.66 |    77.5 |
  app.js             |    77.5 |    16.66 |   16.66 |    77.5 | 35-41,64,70-75
 backend/controllers |   88.73 |    72.22 |   88.88 |   92.42 |
  image.js           |    82.5 |     62.5 |   85.71 |   86.84 | 44-48,78
  login.js           |   93.75 |    66.66 |     100 |     100 | 13-17
  users.js           |     100 |      100 |     100 |     100 |
---------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        8.123 s
```

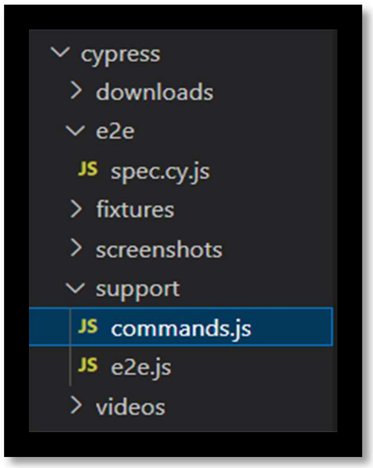
Figure 15. Test result from back-end.

## End to End test (E2E)

Cypress is a JavaScript-based end-user testing tool. It provides a user-friendly interface with options for choosing a runtime environment [38]. Set up for cypress is added to the front-end of the application.

Cypress package is installed with the command ‘npm install –save-dev cypress’. Executing the command ‘npx cypress open’ starts the interface and sets up the runtime structure. It is important to mention that both the front-end and back-end should be up and running before executing cypress tests. Files auto-created from this command are: -

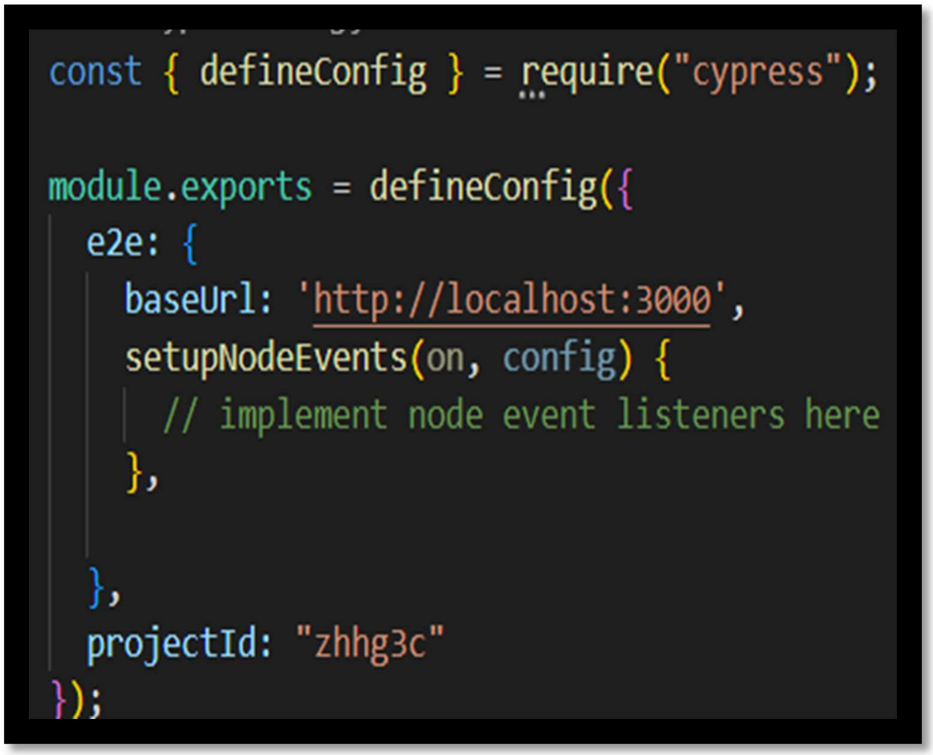
- Cypress directory- main files inside this directory are spec.cy.js and commands.js, containing test cases and helper functions.



```
cypress
  downloads
  e2e
    spec.cy.js
  fixtures
  screenshots
  support
    commands.js
    e2e.js
  videos
```

Figure 16. Cypress file structure.

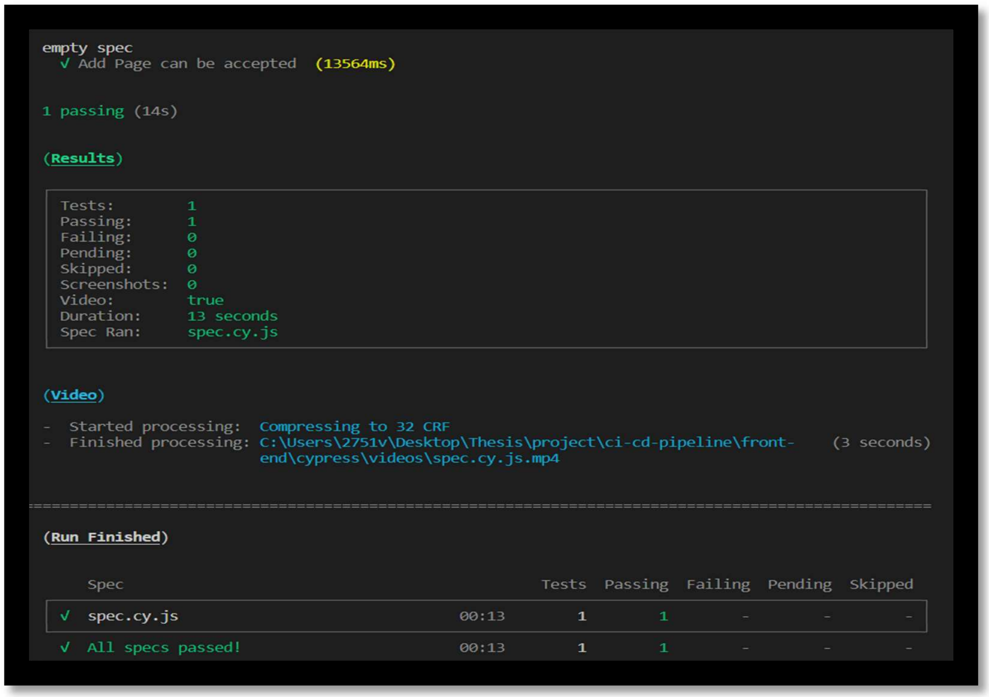
- Cypress.config.js - contains all the setup instructions like baseUrl and projectId to track the run outcome (Figure 17).



```
const { defineConfig } = require("cypress");

module.exports = defineConfig({
  e2e: {
    baseUrl: 'http://localhost:3000',
    setupNodeEvents(on, config) {
      // implement node event listeners here
    },

  },
  projectId: "zhhg3c"
});
```

Figure 17. Cypress.config.js file.

For E2E test, cypress auto logs in the user, add images, and check if they are displayed in the gallery. Figure 18 shows E2E test results.



```
empty spec
  √ Add Page can be accepted  (13564ms)

1 passing (14s)

(Results)

┌──────────────────────────────────────────────────────────────────────────────────────────┐
│ Tests:        1                                                                          │
│ Passing:      1                                                                          │
│ Failing:      0                                                                          │
│ Pending:      0                                                                          │
│ Skipped:      0                                                                          │
│ Screenshots:  0                                                                          │
│ Video:        true                                                                       │
│ Duration:     13 seconds                                                                 │
│ Spec Ran:     spec.cy.js                                                                 │
└──────────────────────────────────────────────────────────────────────────────────────────┘


(Video)

-  Started processing:  Compressing to 32 CRF
-  Finished processing: C:\Users\2751v\Desktop\Thesis\project\ci-cd-pipeline\front-    (3 seconds)
                        end\cypress\videos\spec.cy.js.mp4


====================================================================================================

(Run Finished)


     Spec                                              Tests  Passing  Failing  Pending  Skipped
┌──────────────────────────────────────────────────────────────────────────────────────────┐
│ √  spec.cy.js                               00:13        1        1        -        -        - │
└──────────────────────────────────────────────────────────────────────────────────────────┘
  √  All specs passed!                        00:13        1        1        -        -        -
```

Figure 18. Cypress test run result.

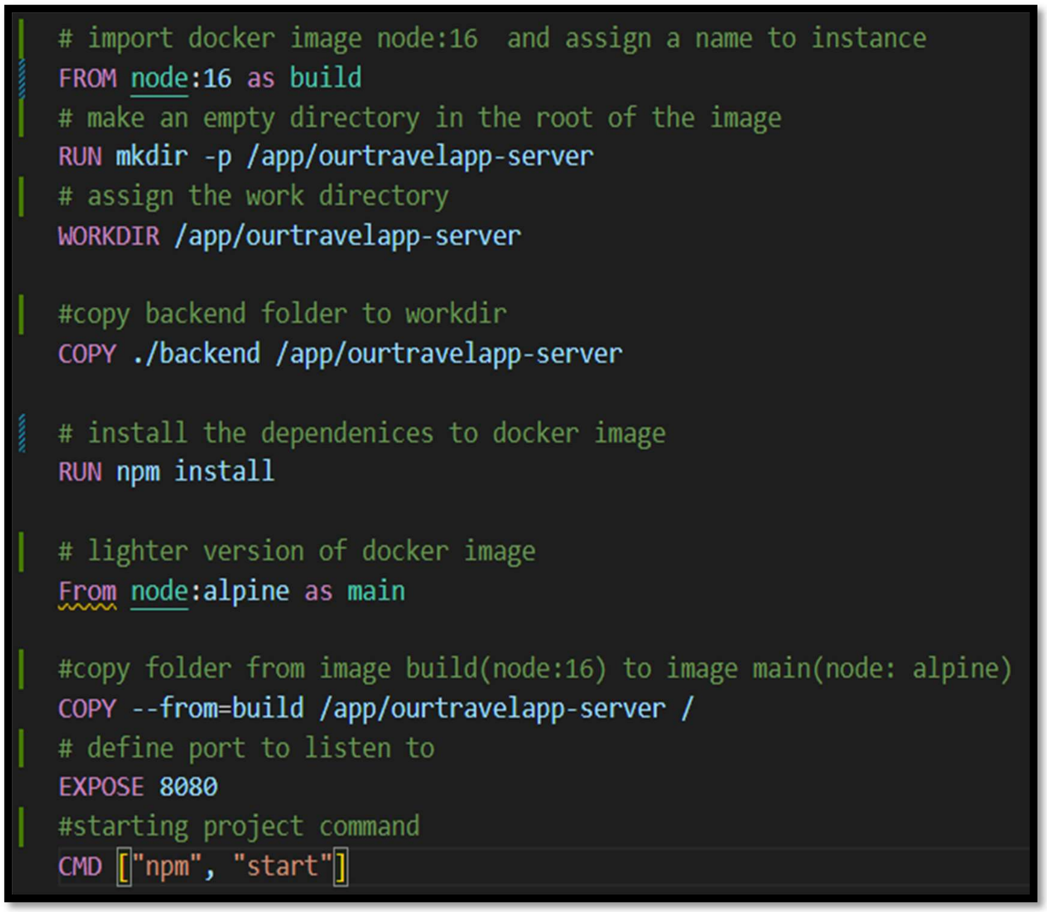
## 3.3 Implementing CI/CD pipeline

### 3.3.1 Creating Docker file and setting up server

Fly.io is an open-source hosting platform, which is used for deploying the Our Travel Gallery application. At the start, we need to install the flyctl command-line utility. For windows, the following command installs the flycts files, which enables the use of flyctl commands in the terminal.

```
iwr https://fly.io/install.ps1 -useb | iex
```

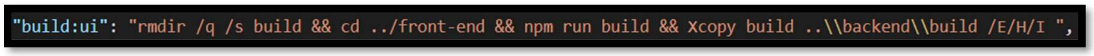
Fly.io only accepts applications wrapped in docker images. So, the addition of a docker file to the root of the project is required. Figure 19 shows the content of the docker file.



```
# import docker image node:16  and assign a name to instance
FROM node:16 as build
# make an empty directory in the root of the image
RUN mkdir -p /app/ourtravelapp-server
# assign the work directory
WORKDIR /app/ourtravelapp-server

#copy backend folder to workdir
COPY ./backend /app/ourtravelapp-server

# install the dependenices to docker image
RUN npm install

# lighter version of docker image
From node:alpine as main

#copy folder from image build(node:16) to image main(node: alpine)
COPY --from=build /app/ourtravelapp-server /
# define port to listen to
EXPOSE 8080
#starting project command
CMD ["npm", "start"]
```

Figure 19. Docker File commands.

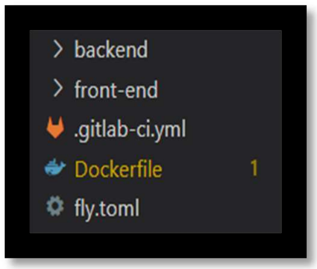
Before deploying the application, a command (Figure 20) is used to build a version of the front-end which is then added to the back-end.



```
"build:ui": "rmdir /q /s build && cd ../front-end && npm run build && Xcopy build ..\\backend\\build /E/H/I ",
```

Figure 20. Build command front-end.

Running the command ‘flyctl login auth’ in the root directory terminal logs the user in. Lastly, the command ‘flyctl launch’ creates a project on the fly server and helps with setting up the fly.toml file locally, based on the docker files created before. Every time the application is deployed, fly.io checks for fly.toml file in the root of the directory, later executing steps and commands from the docker file. Now the project is ready to be deployed with the pipeline. Figure 21 shows the final file structure.



```
> backend
> front-end
.gitlab-ci.yml
Dockerfile          1
fly.toml
```

Figure 21. Final File Structure.

### 3.3.2 Setting up version control with GitLab

As mentioned before, GitLab is the VCS used for storing code of the Our Travel Gallery and implementing CI/CD pipeline. Steps that were followed to get started with GitLab are: -

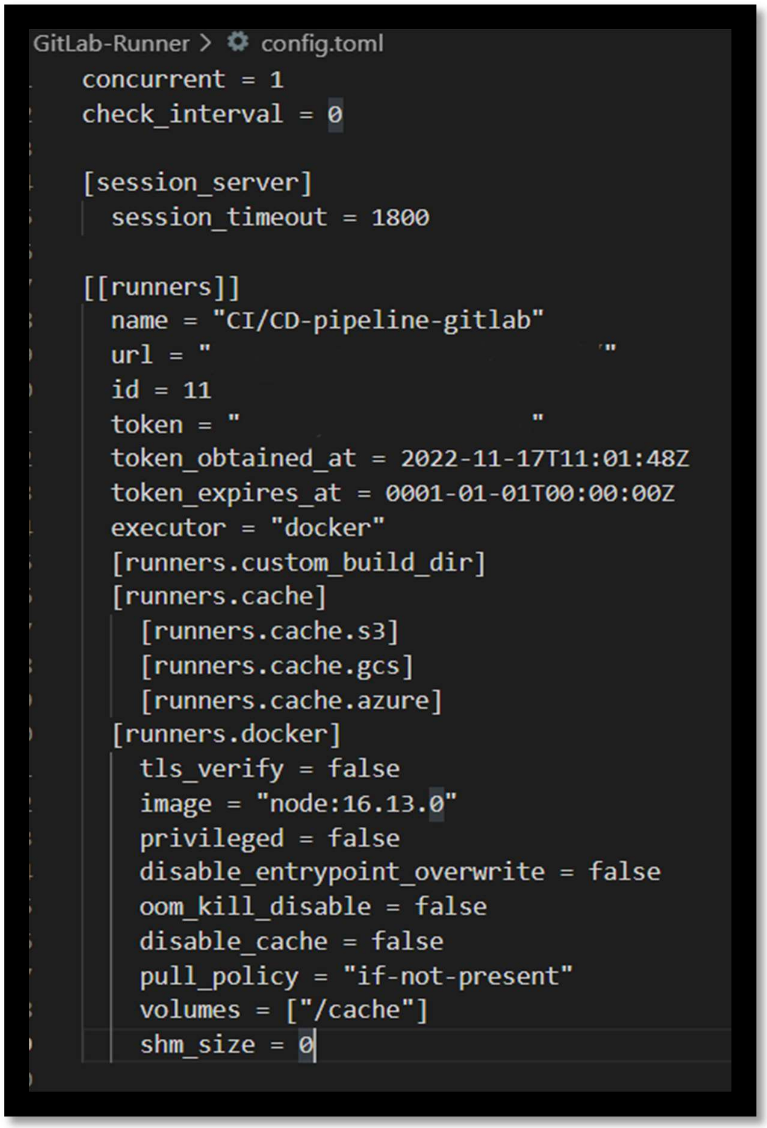
- Installing git on local machine
- Creating GitLab account
- Creating project repository in GitLab
- Learning Git Command Line Interface (CLI)
- Adding demo applications environment variable and CI/CD pipeline environment variables to GitLab's environment variables storage.
- Pushing the code to remote GitLab repository

### 3.3.3 CI/CD pipeline

In a GitLab pipeline script, tasks defined are separated into jobs and stages. While jobs consist of all the executable components of a pipeline, stages generally refer to divisions of a pipeline workflow. These scripts are executed by runners- GitLab instances which executes CI pipeline job over various machines, sending back results and data. The runners are of two types: shared runners and specific runners.

The application uses a specific runner type for pipeline implementation purposes. Before starting, runners need to be installed on a local machine and registered with GitLab. The application uses docker runner which is installed on Windows Operating System (OS). Upon a runner installation, a GitLab-Runner folder is created with contents: gitlab-runner.exe(execution file) and config.toml (setup file).

Config.toml file is used to customize and control the runner's environment.Figure 22 shows contents of a config.toml file.



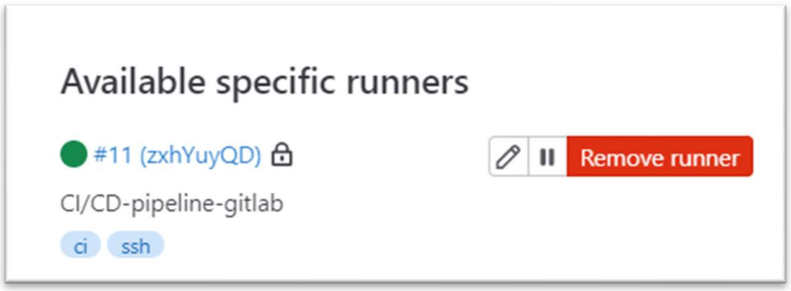
GitLab-Runner > config.toml

```
concurrent = 1
check_interval = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "CI/CD-pipeline-gitlab"
  url = "[redacted]"
  id = 11
  token = "[redacted]"
  token_obtained_at = 2022-11-17T11:01:48Z
  token_expires_at = 0001-01-01T00:00:00Z
  executor = "docker"
  [runners.custom_build_dir]
  [runners.cache]
    [runners.cache.s3]
    [runners.cache.gcs]
    [runners.cache.azure]
  [runners.docker]
    tls_verify = false
    image = "node:16.13.0"
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    pull_policy = "if-not-present"
    volumes = ["/cache"]
    shm_size = 0
```

Figure 22. config.toml file.

The Docker container needs to run locally to access the docker runner in GitLab. Additionally, all the runners are tagged. Tag names are allocated to jobs to point towards the runner which needs to run the job.

GitLab displays the status of the available runners (Figure 23).


Available specific runners

● #11 (zxhYuyQD)

CI/CD-pipeline-gitlab

ci ssh

Figure 23. Active GitLab runner.

The .gitlab-ci.yml is a file for compiling GitLab CI/CD pipeline run commands.

Environment variables from inside the .yml file and GitLab storage are accessed with '$<variablename>'.

Terminologies used in the .yml file are:

- stages- list of stages which runs in a queue
- variables- defines global variables, which are used in .yml file
- default- setting up a default configuration for the pipeline
- script- runnable commands
- stage- refers to one of the stages defined in stages term
- tags- direct the pipeline on which runner to use
- artifacts- a storage, used to store and move data between jobs
- path- a path to a file in the project directory
- expire_in- expiry time for artifacts
- when- run conditions for job execution
- dependencies- used to define dependencies of a job
- needs- defines requirements before running a job
- allow_failure- if true, continues running pipeline after a job failure

- before_script- list of scripts that needs to be executed before a job

Figure 24 shows stages and respective jobs for Our Travel Gallery .yml.

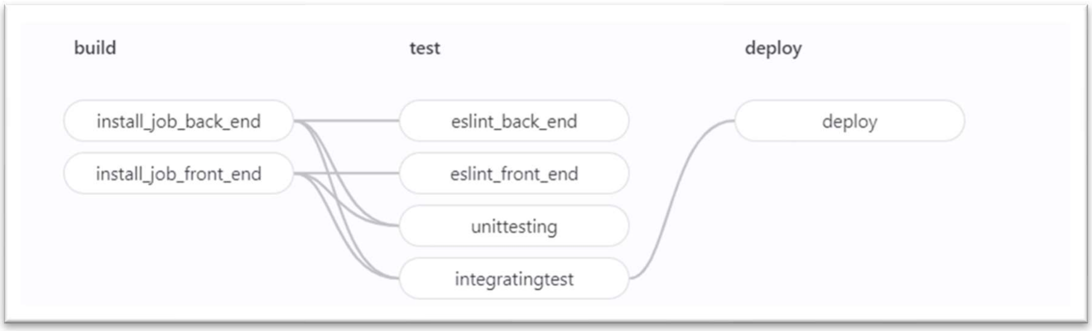
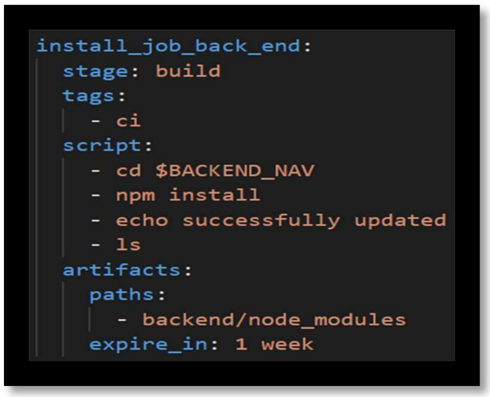


Figure 24. CI/CD pipeline Jobs and connections.

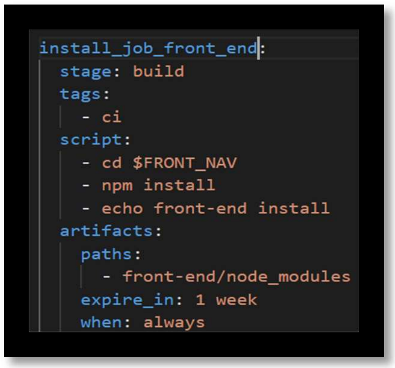
The pipeline has three stages: -

1. **Build**- In this stage, project dependencies are installed, built, and stored as artifacts. In the .yml file there are two jobs defined, one each for front-end and back-end of the application. These jobs are marked with 'stage: build', so the pipeline runs them under the first stage. After that, the term 'tags' allocates the specified runner which executes the command written under the term 'script'. These commands point towards the front-end and the back-end folders, followed by the commands to install the dependencies and saving them as artifacts. Figure 25 and 26 shows our build stage jobs.



```
install_job_back_end:
  stage: build
  tags:
    - ci
  script:
    - cd $BACKEND_NAV
    - npm install
    - echo successfully updated
    - ls
  artifacts:
    paths:
      - backend/node_modules
    expire_in: 1 week
```

Figure 25. Back-end installation job pipeline.



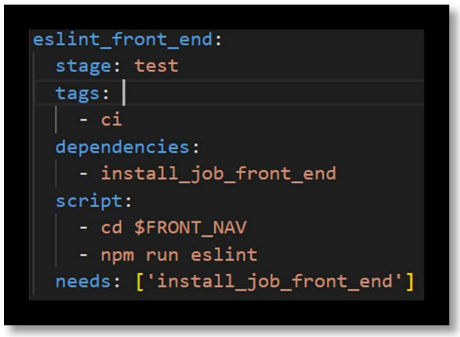
```yaml
install_job_front_end:
  stage: build
  tags:
    - ci
  script:
    - cd $FRONT_NAV
    - npm install
    - echo front-end install
  artifacts:
    paths:
      - front-end/node_modules
    expire_in: 1 week
    when: always
```

Figure 26. Front-end installation job pipeline.

2. **Test**- This stage executes the job specified for application testing. There are five jobs:  two for linting, two for unit testing and one for E2E testing. All these jobs are tagged as ‘stage: test’.

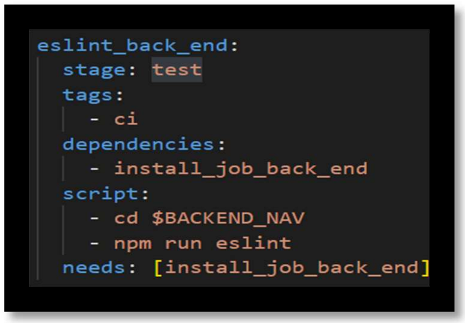
   Firstly, the linting tests are performed on front-end and back-end of the application. The overall structure of the jobs in this stage is similar to the jobs in the build stage, with addition of terms ‘dependencies’ and ‘needs’. For every job execution, the GitLab runner creates a separate instance and a new build space. To save resources and eliminate reoccurring processes, build files from the jobs are saved as artifacts into the GitLab memory. These files are transferred to other jobs by using parent job names as dependencies, eventually saving time and money.

   Figure 27 and 28 shows lint test jobs for the pipeline.



```yaml
eslint_front_end:
  stage: test
  tags:
    - ci
  dependencies:
    - install_job_front_end
  script:
    - cd $FRONT_NAV
    - npm run eslint
  needs: ['install_job_front_end']
```

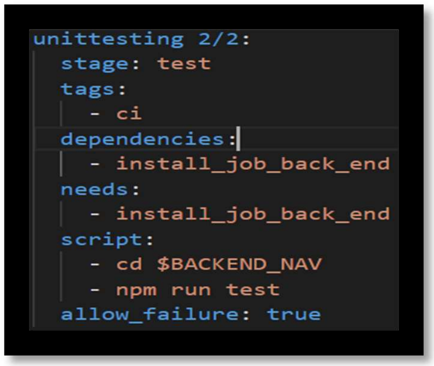
Figure 27. Linting for for front-end.



```
eslint_back_end:
  stage: test
  tags:
    - ci
  dependencies:
    - install_job_back_end
  script:
    - cd $BACKEND_NAV
    - npm run eslint
  needs: [install_job_back_end]
```

Figure 28. Linting job for back-end.

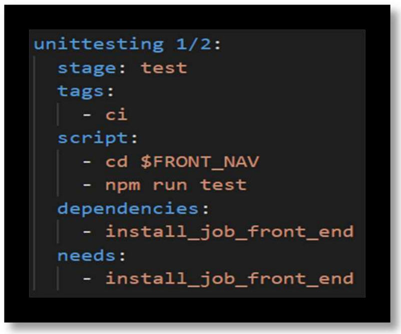
Secondly, the unit and integrated tests with Jest are performed both on the front-end and the back-end. Figure 29 shows unit test jobs for the pipeline.



```
unittesting 2/2:
  stage: test
  tags:
    - ci
  dependencies:
    - install_job_back_end
  needs:
    - install_job_back_end
  script:
    - cd $BACKEND_NAV
    - npm run test
  allow_failure: true
```

Figure 29. Demo application front-end Jest test job.

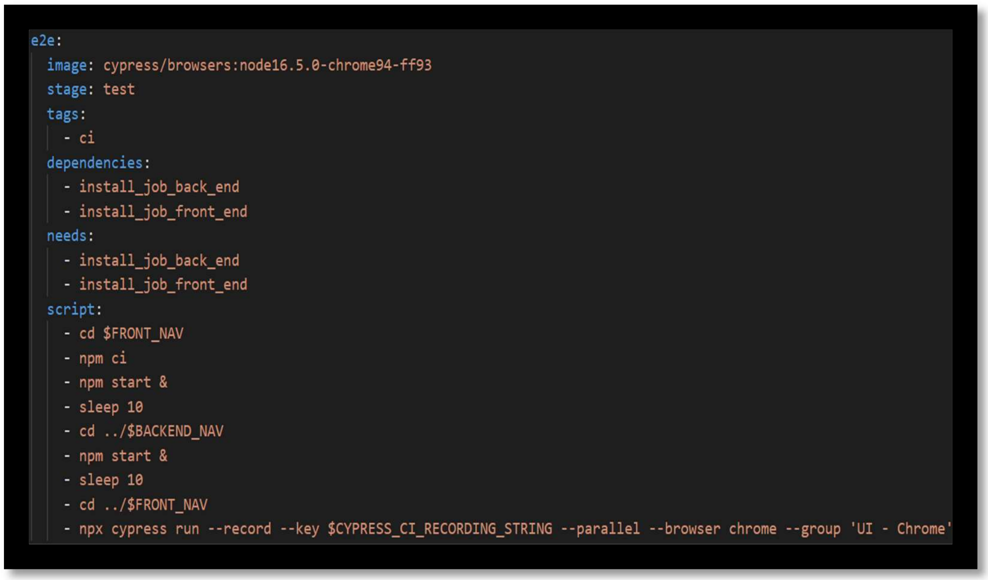
And the Figure 30 shows unit-test jobs for the front-end of the app.



```
unittesting 1/2:
  stage: test
  tags:
    - ci
  script:
    - cd $FRONT_NAV
    - npm run test
  dependencies:
    - install_job_front_end
  needs:
    - install_job_front_end
```

Figure 30. Demo application front-end Jest test job.

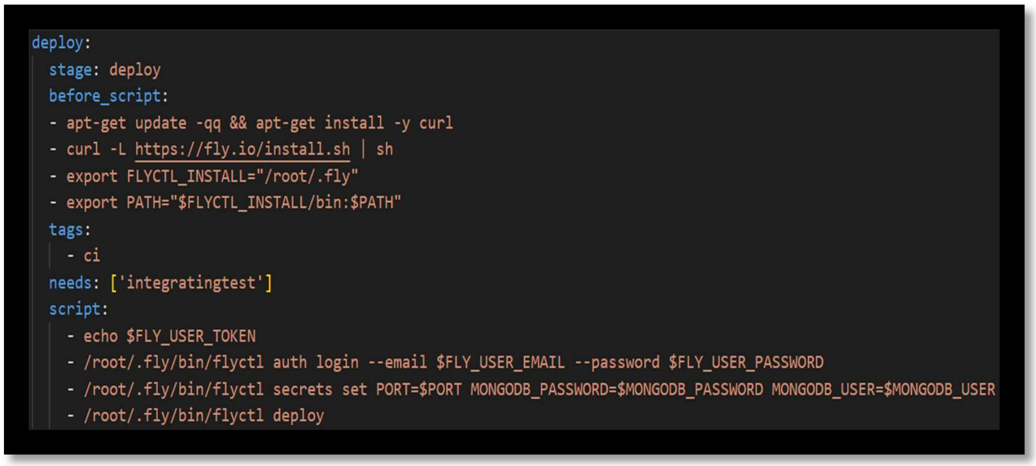
Lastly, E2E test is performed with Cypress. For Cypress a separate image needs to be added. Figure 31 shows the job for E2E testing



```yaml
e2e:
  image: cypress/browsers:node16.5.0-chrome94-ff93
  stage: test
  tags:
    - ci
  dependencies:
    - install_job_back_end
    - install_job_front_end
  needs:
    - install_job_back_end
    - install_job_front_end
  script:
    - cd $FRONT_NAV
    - npm ci
    - npm start &
    - sleep 10
    - cd ../$BACKEND_NAV
    - npm start &
    - sleep 10
    - cd ../$FRONT_NAV
    - npx cypress run --record --key $CYPRESS_CI_RECORDING_STRING --parallel --browser chrome --group 'UI - Chrome'
```

Figure 31. E2E testing job of implemented CI/CD pipeline.

3. **Deploy-** In this stage the application is deployed on the fly.io server. The commands under the term ‘before_script’ installs and adds flyctl command to the root of the docker image. At last, we set all the environment variables and deploy the image to Fly.io. Figure 32 shows the deployment job for the pipeline



```yaml
deploy:
  stage: deploy
  before_script:
  - apt-get update -qq && apt-get install -y curl
  - curl -L https://fly.io/install.sh | sh
  - export FLYCTL_INSTALL="/root/.fly"
  - export PATH="$FLYCTL_INSTALL/bin:$PATH"
  tags:
    - ci
  needs: ['integratingtest']
  script:
    - echo $FLY_USER_TOKEN
    - /root/.fly/bin/flyctl auth login --email $FLY_USER_EMAIL --password $FLY_USER_PASSWORD
    - /root/.fly/bin/flyctl secrets set PORT=$PORT MONGODB_PASSWORD=$MONGODB_PASSWORD MONGODB_USER=$MONGODB_USER
    - /root/.fly/bin/flyctl deploy
```

Figure 32. Deployment job of implemented CI/CD pipeline.

# 4 Results, analysis and discussion

## 4.1 GitLab CI/CD pipeline run result

After the complete development of Our Travel Gallery application, the master code is pushed to GitLab, which activates the CI/CD pipeline. This pipeline implements the instructions from the .yml file (created previously) on to the master code, and during this process, creates a build application version out of the master code, tests it and deploys it to the Fly.io server. The results of the pipeline run are shown in Figure 33. The figure shows that the pipeline run's seven pre-defined jobs, two for building, four for testing and one for deploying the application. All the passed jobs are marked with a green tick and jobs that failed are marked with a red tick as seen in the figure below. Also, jobs that failed but did not stop the pipeline execution are marked with an exclamation mark.

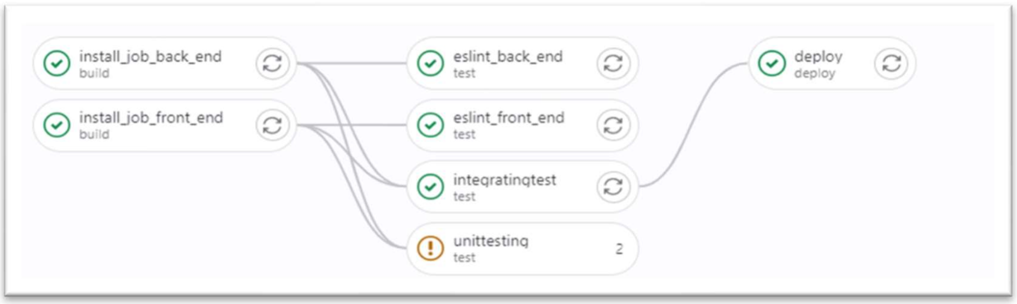


Figure 33. CI/CD pipeline job structure and dependencies.

The next step is to check the efficiency of the CI/CD pipeline. One way to test the pipeline's productivity is by checking the run time of the pipeline for one complete execution. The execution time of the CI/CD pipeline for Our Travel Gallery application was roughly 9 minutes, as shown in the Figure 34. This means developers have to wait approximately less than 10 minutes before receiving the results from the pipeline, which is quite fast. Hence, shorter the run time of a pipeline, the better it becomes.

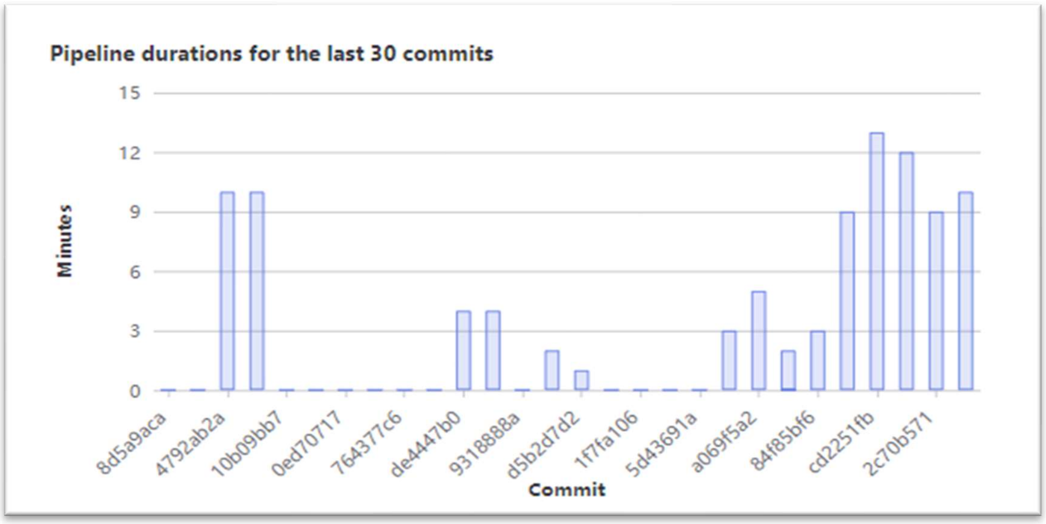


Figure 34. Time taken by CI/CD pipeline to run.

After a successful pipeline run, Our Travel Gallery application is deployed on Fly.io server. The hosted application's URL is given in the Fly.io dashboard, which is used to access the application. Figure 35 shows the successful deployment of Our Travel Gallery.

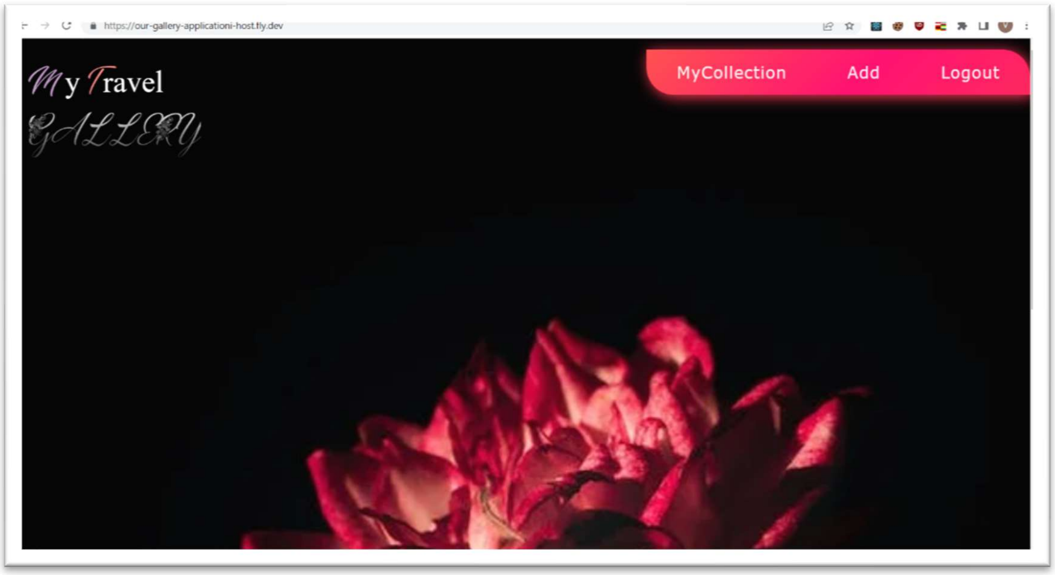


Figure 35. Deployed Our Travel Gallery.

## 4.2 Challenges faced during implementation

- Developing Our Travel Gallery web-application
  Image data is known to be hefty and handling it slows down the application’s response time. Also, an important function of Our Travel Gallery application is to deal with image rendering, uploading, and storing. Every time an image is uploaded to the application, it travels from front-end to back-end to the database and back to be rendered in the UI/UX. This caused a rendering lag of 1.6seconds in the application’s image display area. A solution to this problem (not yet implemented in this application) is to store the images in a separate cloud platform rather than the application’s database, which will significantly reduce the image rendering time and improve user experience.

- Deploying Our Travel Gallery web-application
  Pre-production, the project size was 640 MB and when added to production with docker image, it became more than 1GB. This is a problem as the size of host Fly.io server provides free storage up to 250MB. To solve this issue, two docker images were used during the deployment of Our Travel Gallery application: node:16 and node:alpha (as shown in Figure 19). Docker node:16 image comes with pre-installed packages and tools, therefore is large in size. It was used to install the application’s dependencies and to create a build version. Later, the build version was copied to the node:alpha image, which has a base size of ~5MB. This way the final production image size ended up being 246MB, which was acceptable under Fly.io server free version.

- Heroku server policy change
  Initially, the application was hosted on Heroku server which is an open-source website hosting platform. Deploying with Heroku is easy, with just a few commands, it auto build’s a deployable version of the application before hosting it. Towards the completion of the project, Heroku team

introduced new policy changes, including a removal of free deploying services, which caused hindrance in the development process. For this reason, new hosting platform Fly.io was adapted which has different structural requirements than Heroku. Fly.io requires an application to be containerized before hosting. This was the reason for adding docker image to Our Travel Gallery application.

## 4.3 Answers to research questions

- Which is better manual Integration vs auto Integration?
  From the implementation experience, it can be safely said that auto integration is better compared to manual integration. If the size of the application is small and only one developer is working, then adapting to manual integration becomes less time-consuming. In all other cases, auto integration stands out. Auto scripts run the integration process on every push or pull command and reduces the manual work.

- What kind of projects benefit from CI/CD?
  CI/CD pipeline is a useful tool, but it comes with extra work and an additional budget. Developing a CI/CD pipeline requires knowledge of different tools and technologies. Even if the tools are familiar, dynamic changes in the pipeline are required to adjust to the changing project environment. So, medium to large size projects with spare funding benefit from a CI/CD pipeline the most. Additionally, companies having self-hosted servers can benefit from CI/CD pipeline, as it cuts the project hosting cost.

# 5 Conclusion

The main goal of this thesis was to enhance the application development and delivery process by automating the generic time-consuming development processes such as application integration, testing, and deployment. This was achieved by integrating the use of CI/CD methods with the development process. The other objectives were to deliver a working example of CI/CD pipeline to the Health Tech Lab TUAS and answer two frequently asked questions by CI/CD developers. One question was which type of integration, namely, manual integration or auto integration, can be considered more useful. The other question was what types of projects which can benefit from CI/CD.

For the implementation of this thesis, technologies from Health Tech Lab TUAS were used. These technologies required the use of GitLab for building a CI/CD pipeline and React/NodeJS for building Our Travel Gallery web-application.

The outcome of the implementation is a simple and detailed CI/CD pipeline that auto builds, tests, and deploys the demo application (Our Travel Gallery). At this point, any break or job fail in the pipeline stops the execution and notifies the developer through an email. Using this process developers can act fast and fix the issues which leads the pipeline to failure, eventually, improving collaboration and co-ordination between developing team, which leads to a better-quality product.

This thesis is not an actual guide to building a CI/CD pipeline, but an example case, with detailed step by step instructions that can be referred to for implementing a CI/CD pipeline on GitLab.

## References

1. Bureau of Labor Statistics, U.S. Department of Labor, *Occupational Outlook Handbook*, Software Developers, Quality Assurance Analysts, and Testers, at https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm (visited *November 28, 2022*).

2. Staff, V.B. (2021) Automation is key to growing software development, execs say, VentureBeat. VentureBeat. Available at: https://venturebeat.com/business/automation-is-key-to-growing-software-development-execs-say/ (Accessed: December 11, 2022).

3. *Continuous delivery market size, share: 2022 - 27: Industry forecast* (2022) *Continuous Delivery Market Size, Share | 2022 - 27 | Industry Forecast*. Available at: https://www.mordorintelligence.com/industry-reports/continuous-delivery-market (Accessed: December 12, 2022).

4. Schneckenberg, D. *et al.* (2021) 'Value creation and appropriation of software vendors: A digital innovation model for cloud computing', *Information & Management*, 58(4), p. 103463. doi: 10.1016/j.im.2021.103463.

5. Team, J.B. (no date) *TeamCity CI/CD guide*, *JetBrains*. Available at: https://www.jetbrains.com/teamcity/ci-cd-guide/ (Accessed: November 19, 2022).

6. Pratama, M. R.  and Sulistiyo Kusumo, D.  "Implementation of Continuous Integration and Continuous Delivery (CI/CD) on Automatic Performance Testing," 2021 9th International Conference on Information and Communication Technology (ICoICT), 2021, pp. 230-235, doi: 10.1109/ICoICT52021.2021.9527496.

7. Donca, I-C., Stan, O.P., Misaros, M., Gota D. & Miclea, L. Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects. *Sensors*. 2022; 22(12):4637. https://doi.org/10.3390/s22124637

8. Team, R.H. (2022) *What is Ci/CD?*, *Red Hat - We make open source technologies for the enterprise*. Available at: https://www.redhat.com/en/topics/devops/what-is-ci-cd (Accessed: November 20, 2022).

9. Karuturi, A.B.S. (2021) *An introduction to continuous integration*, *Qentelli*. Available at: https://www.qentelli.com/thought-leadership/insights/continuous-integration (Accessed: November 19, 2022).

10. Roper, J. (2022) *CI/CD pipeline : Everything you need to know with examples*, *Spacelift*. Available at: https://spacelift.io/blog/ci-cd-pipeline (Accessed: November 20, 2022).

11. Humble, J. and Farley, D. (2015) *Continuous delivery: Reliable software releases through build, test, and Deployment Automation*. Upper Saddle River, NJ u.a: Addison-Wesley.

12. Brigginshaw, D. (2022) *Guide to CI/CD pipeline: Everything you need to know (2022)*, *Scriptworks*. Available at: https://www.scriptworks.io/blog/ci-cd-pipeline/ (Accessed: December 7, 2022).

13. Anastasov, M.. (2022) *CI/CD pipeline: A gentle introduction*, *Semaphore*. Available at: https://semaphoreci.com/blog/cicd-pipeline (Accessed: November 22, 2022).

14. Team, K. (no date) *Benefits of Continuous Integration & Delivery: CI/CD benefits*, *katalon.com*. Available at: https://katalon.com/resources-center/blog/benefits-continuous-integration-delivery (Accessed: November 22, 2022).

15. Team, J.B. (2020) *What are the benefits of CI/CD?: Teamcity CI/CD guide*, *JetBrains*. Available at: https://www.jetbrains.com/teamcity/ci-cd-guide/benefits-of-ci-cd/ (Accessed: November 22, 2022).

16. Rao, M. (2018) *Common security challenges in CI/CD workflows - dzone*, *dzone.com*. Available at: https://dzone.com/articles/common-security-challenges-in-cicd-workflows (Accessed: November 23, 2022).

17. Rajora, H. (2021) *CI/CD benefits, challenges and best practices for your team*, *TestProject*. Available at: https://blog.testproject.io/2021/04/22/ci-cd-benefits-challenges-best-practices-for-your-team/ (Accessed: December 11, 2022).

18. Choudhary, N. (2022) *Top 10 CI/CD pipeline implementation challenges and solutions*, *LambdaTest*. Available at: https://www.lambdatest.com/blog/cicd-pipeline-challenges/ (Accessed: November 22, 2022).

19. GitLab Inc. (2021) *History of Gitlab*, *GitLab*. Available at: https://about.gitlab.com/company/history/ (Accessed: November 22, 2022).

20. HG Insight Team (no date) *Gitlab agile planning - discovery.hgdata.com*, *GitLab*. Available at: https://discovery.hgdata.com/product/gitlab-agile-planning (Accessed: November 23, 2022).

21. Katalon team (2022) *Best automated API testing tools for software testing in 2022*, *katalon.com*. Katalon. Available at: https://katalon.com/resources-center/blog/top-5-free-api-testing-tools (Accessed: November 13, 2022).

22. Ashiq, F. (2022) *11 best automated ui testing tools in 2022*, *LambdaTest*. Available at: https://www.lambdatest.com/blog/top-ui-automated-testing-tools/ (Accessed: November 12, 2022).

23. Positive Technologies (2022) *Threats and vulnerabilities in web applications 2020–2021*, *Positive Technologies - vulnerability assessment, compliance management and threat analysis solutions*. Positive Technologies. Available at: https://www.ptsecurity.com/ww-en/analytics/web-vulnerabilities-2020-2021/ (Accessed: December 2, 2022).

24. Williams, L. (2022) *9 best security testing tools (2022)*, *Guru99*. Available at: https://www.guru99.com/security-testing-tools.html (Accessed: December 2, 2022).

25. Red Hat Team (2022) *What is agile methodology?*, *Red Hat - We make open source technologies for the enterprise*. Available at: https://www.redhat.com/en/topics/devops/what-is-agile-methodology (Accessed: November 26, 2022).

26. JetBrains (no date) *CI/CD in Agile Development: Teamcity CI/CD guide*, *JetBrains*. Available at: https://www.jetbrains.com/teamcity/ci-cd-guide/agile-continuous-integration/ (Accessed: December 12, 2022).

27. Steve, J. (2021) *What's the difference between agile, CI/CD, and DevOps?*, *Application Security Blog*. Available at: https://www.synopsys.com/blogs/software-security/agile-cicd-devops-difference/ (Accessed: November 16, 2022).

28. Mijacobs *et al.* (2022) *What is DevOps? - azure DevOps*, *Azure DevOps | Microsoft Learn*. Available at: https://learn.microsoft.com/en-us/devops/what-is-devops (Accessed: December 5, 2022).

29. JetBrains (2020) *Understanding CI/CD in DevOps: Teamcity CI/CD guide*, *JetBrains*. Available at: https://www.jetbrains.com/teamcity/ci-cd-guide/devops-ci-cd/ (Accessed: November 27, 2022).

30. Hall, J. (2021) *A brief history of CI/CD*, *Jonathan Hall*. Available at: https://jhall.io/archive/2021/09/26/a-brief-history-of-ci/cd/ (Accessed: December 12, 2022).

31. Korakitis, K. (2020) *DevOps CI/CD usage trends*, *Developer Nation Community*. Available at: https://www.developernation.net/blog/devops-ci-cd-usage-trends (Accessed: December 12, 2022).

32. Wadhwani, P. and Loomba, S. (2022) *DevOps Market Size & Share, global trends 2022-2028*, *Global Market Insights Inc.* Available at: https://www.gminsights.com/industry-analysis/devops-market (Accessed: December 1, 2022).

33. Muhsin, M. (2022) *AMP performance with react server-side rendering*, *LogRocket Blog*. Available at: https://blog.logrocket.com/improve-app-performance-react-server-side-rendering/ (Accessed: November 21, 2022).

34. Heller, M. (2022) *What is node.js? the JavaScript runtime explained*, *InfoWorld*. InfoWorld. Available at: https://www.infoworld.com/article/3210589/what-is-nodejs-javascript-runtime-explained.html (Accessed: December 1, 2022).

35. Tabirao, M.A. (2022) *What is mongodb and why use it for modern web applications?*, *Ubuntu*. Ubuntu. Available at: https://ubuntu.com/blog/what-is-mongodb (Accessed: December 2, 2022).

36. Gupta, S. (2021) *ESLint: What, why, when, how*, *DEV Community*. DEV Community. Available at: https://dev.to/shivambmgupta/eslint-what-why-when-how-5f1d (Accessed: December 6, 2022).

37. Vaidya, N. (2022) *Jest framework tutorial: How to use it*, *BrowserStack*. Available at: https://www.browserstack.com/guide/jest-framework-tutorial (Accessed: November 13, 2022).

38. Kinsbruner, E. (2021) *What is Cypress Testing? what it is and how to get started*, *Perfecto by Perforce*. Available at: https://www.perfecto.io/blog/cypress-testing (Accessed: December 12, 2022).