CI/CD for Telemedicine

B.Tech Project Report

Submitted by **Bikkavolu Prasanthi**
B20CS010

Under supervision of Dr. Sumit Kalra

Department of Computer Science and Engineering
Indian Institute of Technology Jodhpur
November 2023

Acknowledgement

I extend my heartfelt appreciation to my project supervisor, **Dr.Sumit Kalra**, for his invaluable guidance, continuous support, and constructive feedback throughout the duration of this project. I also extend my gratitude to my project mentor **Mr. Manik Sejwal** .Their expertise has been a guiding light, steering me towards a deeper understanding of the subject matter and fostering my intellectual growth. I am also grateful for the support and help, which played a crucial role in shaping the project.

Declaration

I hereby declare that the work presented in this B.Tech Project Report titled **CICD for Telemedicine** Report submitted to the Indian Institute of Technology Jodhpur in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology , is a bonafide record of the project work carried out under the supervision of Dr.Sumit Kalra. The contents of this B.Tech Project Report in full or in parts, have not been submitted to, and will not be submitted by me to, any other Institute or University in India or abroad for the award of any degree or diploma.

B·Prasanthi

Signature

Name of the Student : Bikkavolu Prasanthi

Roll Number : B20CS010

Certificate

This is to certify that the B.Tech Project Report titled **CICD for Telemedicine** Report, submitted by *Bikkavolu Prasanthi(B20CS010)* to the Indian Institute of Technology Jodhpur for the award of the degree of Bachelor of Technology, is a bonafide record of the project work done by her under my supervision. To the best of my knowledge, the contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Signature

Name of the Supervisor: Dr.Sumit Kalra

Abstract

Continuous Integration/Continuous Deployment (CICD) for Telemedicine is now an integral part of the Telemedicine project, designed to revolutionize healthcare accessibility and delivery. This project leverages modern software development practices to ensure seamless integration, testing, and deployment of both its backend and frontend components. The CICD pipeline automates the process of updating the Telemedicine application on remote servers in response to code changes, providing efficiency, reliability, and rapid deployment.

The CICD workflow involves version control, automated testing, and deployment stages. The backend, built with Node.js and utilizing technologies such as MongoDB and Redis, is subjected to automated testing and subsequent deployment using PM2 for process management. The frontend, developed with technologies like React, undergoes testing and is deployed to an Nginx server serving as the application's public-facing interface.

Key features of the CICD pipeline include version-controlled code repositories, automated unit and integration testing, secure deployment through SSH, and automatic updates triggered by code pushes to the designated branches. The use of GitHub Actions and SSH automation tools streamlines the process, enhancing collaboration and reducing manual intervention.

This report delves into the architecture, implementation, and benefits of CICD in general and for Telemedicine , showcasing how it ensures code quality, facilitates collaboration among development teams, and accelerates the delivery of new features and improvements to end-users in the rapidly evolving field of Telemedicine.

Table of Contents

Acknowledgement
Declaration
Certificate
Abstract
1. Introduction 5
1.1. Objectives
1.2. What is CI/CD ?
2. Methodology 6
2.1. Continuous Integration 6
2.2. Continuous Deployment 7
3. Implementation 8
3.1. Tools Used 8
3.2. CI/CD 8
3.3. Continuous Integration 11
3.4. Continuous Deployment 13
4. Observations 16
5. Learning Outcomes 19
6. Conclusion 19
7. Future Work 19
8. References 20

1. Introduction

Traditional software development methodologies are not enough to fulfill nowadays business requirements. Existing approaches to integration, testing, and deployment have proven to be inefficient, leaving vulnerabilities that compromise the overall security of the process.The repetitive cycle of individual team members independently building, testing, and deploying on their local machines not only poses significant risks but also introduces redundancy and potential errors.This eventually creates more bugs when several people work on the same code. Despite exhaustive preventive measures, these incidents persist with alarming frequency.

In modern application development, it's common for multiple developers to work concurrently on various aspects of the same app. However, when all the separate branches of code are merged on a designated day, this process can be laborious, manual, and time-consuming. This is due to the possibility of conflicts arising when changes made by different developers intersect. The use of individualized local integrated development environments (IDEs) instead of a unified cloud-based IDE can exacerbate this issue.

Hence we need a robust, automated system that can seamlessly integrate, test, and deploy software.

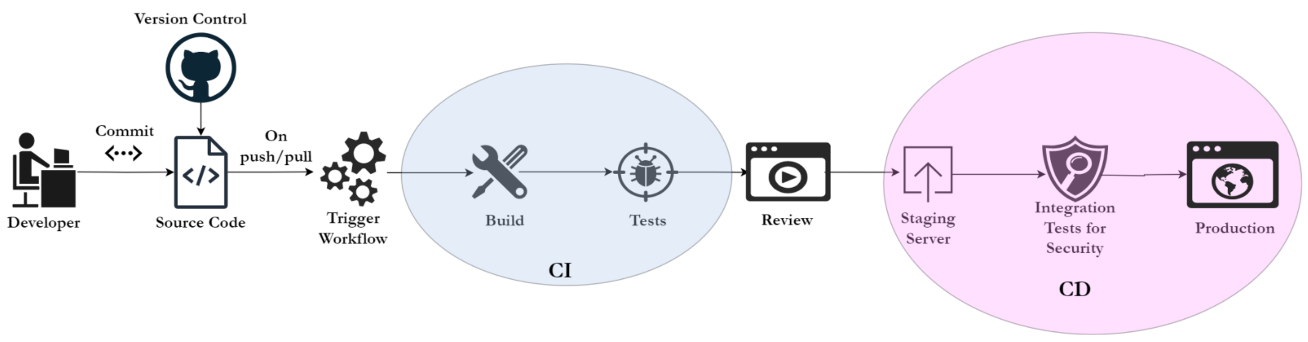
1.1 Objectives

The project "CI/CD for Telemedicine" aims to enhance the software development process by implementing Continuous Integration and Continuous Deployment (CI/CD) methodologies. The primary objectives include *enhancing efficiency* by automating integration, testing, and deployment processes, thereby *accelerating time-to-market*. It also seeks to *mitigate risks* associated with manual processes, foster *collaboration* among development teams, and establish a *continuous feedback loop* for swift issue resolution. Furthermore, the project intends to bolster *security* through automated checks, expedite software delivery, and provide scalability and flexibility to adapt to evolving requirements. Lastly, it aims to enhance auditability and traceability, ensuring compliance with regulatory requirements.

1.2 What is CI/CD ?

CI/CD falls under DevOps (the joining of development and operations teams) and combines the practices of continuous integration and continuous delivery. CI/CD automates much or all of the manual human intervention traditionally needed to get new code from a commit into production, encompassing the build, test (including integration tests, unit tests, and regression tests), and deploy phases, as well as infrastructure provisioning. With a CI/CD pipeline, development teams can make changes to code that are then automatically tested and pushed out for delivery and deployment.

CI/CD pipeline



2. Methodology

2.1. Overview

Continuous Integration and Continuous Deployment (CI/CD) is a software development methodology that streamlines the delivery pipeline from code development to production deployment. In CI, developers merge their code changes into a shared repository frequently, triggering automated builds and comprehensive testing to detect and resolve issues early. The CD process automates the release of validated code changes, first deploying to a staging environment for additional testing and validation before seamlessly transitioning to production. CI/CD practices emphasize efficiency, collaboration, and reliability by automating repetitive tasks, ensuring code quality through continuous testing, and enabling rapid and secure deployment, ultimately facilitating a more agile and responsive development lifecycle.

2.2. Continuous Integration

Continuous integration (CI) aims to alleviate these challenges by enabling developers to merge their code changes back to a shared branch or "trunk" more frequently, often on a daily basis. Following the merge, the changes undergo automatic validation through building the application and running automated tests, including unit and integration tests, to ensure that the changes haven't introduced any issues. This comprehensive testing covers everything from classes and functions to the various modules that form the entire application. If conflicts or bugs are discovered during automated testing, CI facilitates a quicker resolution and mitigation of these issues.

Continuous Integration (CI) is structured around a series of stages that collectively enhance the efficiency and reliability of the software development process. The key stages in CI encompass:

1. **Automated Building**: Following code merging, the CI pipeline initiates an automated build process. This involves compiling the source code, linking libraries, and generating executable files. Automated building ensures that the integrated code can be successfully transformed into a functional application.
2. **Unit Testing:** The next stage involves the execution of unit tests. These tests focus on individual components, such as classes and functions, to verify that each unit of code functions as intended in isolation. Unit testing helps identify any discrepancies or errors introduced during the integration process.
3. **Integration Testing:** Building on unit testing, integration testing evaluates the interactions between different units or modules within the application. This stage ensures that the integrated components collaborate seamlessly and do not introduce conflicts or issues that may arise when various modules interact.
4. **Automated Validation:** The comprehensive testing process includes automated validation of the entire application. This entails running a suite of tests that cover not only individual units and integrated modules but also the holistic functionality of the application. Automated validation aims to catch any latent issues that may emerge when components work together.
5. **Issue Resolution and Mitigation:** If conflicts, bugs, or inconsistencies are detected during any stage of testing, CI facilitates a rapid resolution. This quick feedback loop enables developers to address issues promptly, ensuring that the codebase remains stable and reliable.

Continuous Integration, through these well-defined stages, promotes a collaborative and error-resistant development environment. By automating crucial processes, developers can confidently contribute to the codebase, knowing that their changes undergo rigorous testing, fostering code quality and enhancing the overall integrity of the software.

2.3. Continuous Deployment

Continuous Deployment (CD), the final stage in a mature CI/CD pipeline, automates the release of applications directly to the production environment, eliminating manual intervention. This means that changes made by a developer could be live within minutes, provided automated tests are successful. CD enables continuous integration of user feedback and reduces deployment risks by releasing changes incrementally. However, it requires a significant initial investment for developing comprehensive automated tests for various testing and release phases.

Continuous Deployment (CD) is a streamlined process focused on the automated and efficient release of software changes to production. The key stages in CD include:

1. **Deployment Automation:** Following successful testing, the CD pipeline automates the deployment of the application to the production environment. This stage involves the seamless transition of the validated code changes into the live system.
    a. **Staging Deployment** : Validation of code changes in a staging environment that closely mimics the production setup.

    b. **Staging Testing** : Additional testing and validation in the staging environment to confirm stability and performance.
    c. **Production Deployment** : Seamless transition of validated code changes to the live production environment.
2. **Monitoring and Validation** :
    a. **Performance Monitoring**: Continuous observation of metrics (response times, error rates, resource utilization) in the production environment.
    b. **User Acceptance Testing** (UAT): Real-world testing with actual users to ensure the application meets business requirements.

3. Implementation

3.1. Tools Used

In the Telemedicine project, **GitHub** serves as the version control system, with **GitHub Actions** selected as the primary CI/CD tool. GitHub Actions provides a streamlined, code-centric interface, enabling developers to modify workflows efficiently. For the frontend, built with **Node.js** and **React.js, Node.js** is employed in the CI process, ensuring the continuous integration of the frontend. The backend, utilizing **MongoDB** and **Redis**, integrates with **pm2** for continuous integration. Unit testing is conducted using **Jest** and **React Testing Library**, while **Selenium** with **Java and Maven** is employed for integration testing. Continuous deployment leverages **Google Cloud Platform** for production, facilitated through an SSH connection and orchestrated by **GitHub Actions**. Additionally, **nginX** is utilized as a reverse proxy, enhancing scalability, optimizing resource distribution, and providing an added layer of security.This comprehensive toolchain ensures a robust CI/CD pipeline for the Telemedicine project, enhancing efficiency, reliability, and maintainability.

3.2. CI/CD

Github Actions has been used for the main CI/CD. GitHub Actions is a CI/CD platform designed for automating various processes such as build, test, and deployment pipelines. This tool allows users to create workflows that automate the testing and building of each pull request, as well as deploy successfully merged pull requests to production environments.

GitHub Actions extends its functionality beyond traditional DevOps by enabling the execution of workflows in response to specific events within a repository. For instance, users can set up workflows to automatically assign appropriate labels whenever a new issue is created in the repository.

GitHub offers virtual machines running Linux, Windows, and macOS to execute workflows. Additionally, users have the option to utilize self-hosted runners within their own data centers or cloud infrastructure.

The components of GitHub Actions:

You can configure a GitHub Actions workflow to be triggered when an event occurs in your repository, such as a push or pull request being opened or an issue being created. Your workflow contains one or more jobs which can run in sequential order or in parallel. Each job will run inside its own virtual machine runner, or inside a container, and has one or more steps that either run a script that you define or run an action, which is a reusable extension that can simplify your workflow.

1. **Workflows:** Workflows are defined in the .github/workflows directory in a repository and a repository can have multiple workflows, each of which can perform a different set of tasks.
2. **Events :** An event is a specific activity in a repository that triggers a workflow run.
3. **Jobs :** A job comprises a series of steps within a workflow, all executed on a shared runner. Each step can take the form of a shell script to be executed or an action to be run. These steps follow a sequential order and rely on dependencies between them. As each step operates on the same runner, there is the capability to exchange data seamlessly from one step to the next. We can configure a job's dependencies with other jobs; by default, jobs have no dependencies and run in parallel with each other. When a job takes a dependency on another job, it will wait for the dependent job to complete before it can run.
4. **Runners :** A runner serves as a server responsible for executing workflows upon trigger. Each runner has the capacity to handle a single job concurrently. GitHub offers runners for Ubuntu Linux, Microsoft Windows, and macOS to facilitate the execution of workflows. Each instance of a workflow run takes place within a newly provisioned virtual machine, ensuring a fresh environment for the workflow's execution.

Hence a workflow must contain the following basic components:

1. One or more events that will trigger the workflow.
2. One or more jobs, each of which will execute on a runner machine and run a series of one or more steps.
3. Each step can either run a script that you define or run an action, which is a reusable extension that can simplify your workflow.

Understanding the workflow file:

1. name: CI-CD : *Optional* - The name of the workflow as it will appear in the "Actions" tab of the GitHub repository. If this field is omitted, the name of the workflow file will be used instead.
2. on: [push] : Specifies the trigger for this workflow. This example uses the push event, so a workflow run is triggered every time someone pushes a change to the repository or merges a pull request. This is triggered by a push to the specified branch. For example, in our workflow we used:

```
on:
    push:
        branches: ["cicd-prasanthi", "main"]
    pull_request:
        branches: ["cicd-prasanthi", "main"]
```

3. jobs: Groups together all the jobs that run in the CI-CDworkflow.
4. runs-on: ubuntu-latest : Configures the job to run on the latest version of an Ubuntu Linux runner. This means that the job will execute on a fresh virtual machine hosted by GitHub.
5. strategy:

   matrix: The matrix strategy in GitHub Actions allows us to run a job with multiple configurations in parallel, which can be particularly useful for testing across different environments or configurations.

6. steps: Groups together all the steps that run in the job. Each item nested under this section is a separate action or shell script.
7. - uses: actions/checkout@v4 : The *uses* keyword specifies that this step will run *v4* of the actions/checkout action. This is an action that checks out our repository onto the runner, allowing us to run scripts or other actions against our code (such as build and test tools). We should use the checkout action any time your workflow will use the repository's code.
8. - uses: actions/setup-node@v3

   with: node-version: '14' : This step uses the actions/setup-node@v3 action to install the specified version of the Node.js. (This example uses version 14.) This puts both the node and npm commands in our PATH.

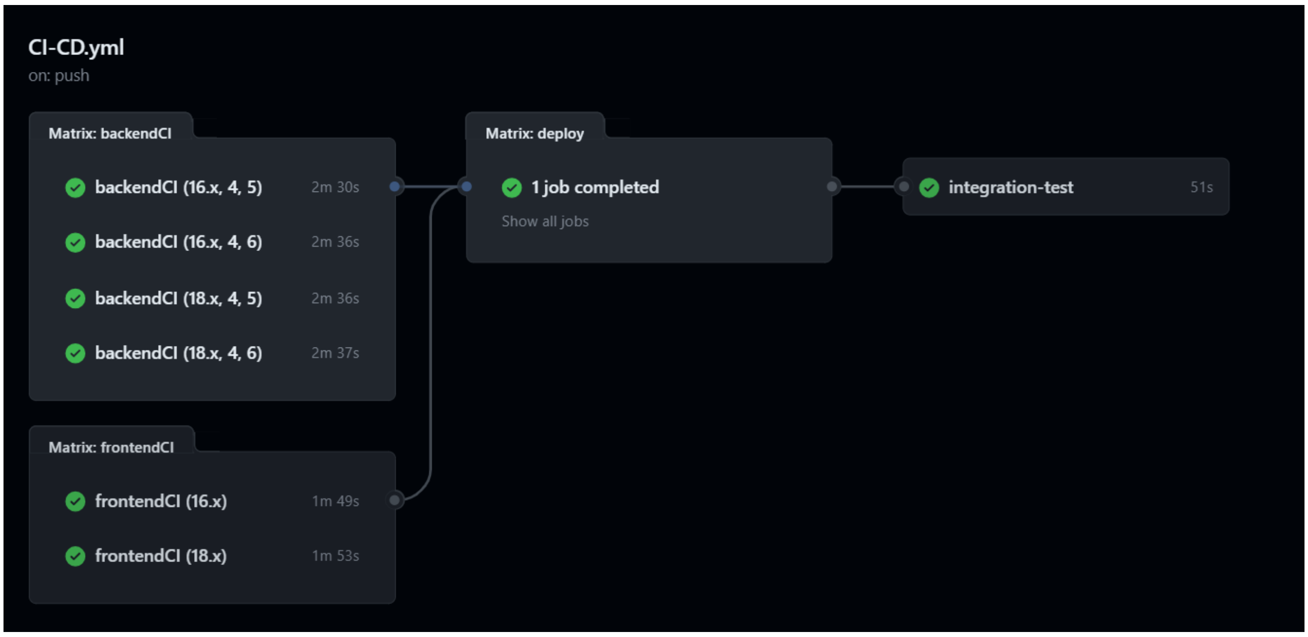
9. - run: npm install : The *run* keyword tells the job to execute a command on the runner. In this case, we are using npm to install all the required packages.

Creating dependent jobs:

By default, the jobs in our workflow all run in parallel at the same time. If we have a job that must only run after another job has completed, we can use the needs keyword to create this dependency. If one of the jobs fails, all dependent jobs are skipped; however, if you need the jobs to continue, we can define this using the if conditional statement.

Visualizing the workflow file:

In this diagram, you can see the workflow file we created and how the GitHub Actions components are organized in a hierarchy. Each step executes a single action or shell script.



3.3. Continuous Integration

Continuous Integration (CI) in GitHub Actions is a development practice that involves automatically testing and building code changes on an event to the repository. The primary goal is to detect and fix issues early in the development process. In our GitHub Actions workflows (frontendCI and backendCI), we've set up CI for the frontend and backend components separately.

Building Frontend as part of CI:

In the context of Continuous Integration (CI) for the frontend, our GitHub Actions workflow (frontendCI) ensures the seamless integration of code changes by automating the build process and executing essential tests. This practice allows us to catch potential issues early in the development cycle.

The Install Dependencies and Update Dependencies steps ensure that the frontend dependencies are correctly installed and updated. The subsequent steps build frontend and run necessary tests.

The *Build Frontend* step in the workflow compiles the frontend code using the specified Node.js versions. Following this, the Testing Frontend step utilizes the Jest testing framework and React Testing Library (RTL) to perform unit tests on the frontend codebase. Jest is employed for JavaScript testing, while RTL is specifically designed for testing React components.

Front-end component testing:

For front-end component testing, the *Testing Frontend* step employs Jest and React Testing Library (RTL). Jest is a widely-used JavaScript testing framework that provides a comprehensive suite of testing utilities. It excels at unit testing by capturing snapshots of React components and ensuring that subsequent renders match the established snapshots. Jest also facilitates the creation of test suites and the execution of test cases, contributing to the overall robustness of our frontend components.

React Testing Library (RTL) complements Jest by offering a user-centric testing approach. RTL focuses on testing components from the perspective of user interactions, encouraging developers to write tests that closely resemble real-world scenarios. This ensures that our frontend components not only function correctly but also provide a positive user experience.

Frontend Continuous Integration Workflow Steps:

1. Git Checkout:
   uses: actions/checkout@v3: Checks out the code from the repository.
2. Clear npm Cache:
   run: npm cache clean --force: Clears the npm cache to ensure a clean installation.
3. Remove existing node_modules:
   run: rm -rf node_modules: Removes the existing node_modules directory to start with a clean slate.
4. Use Node.js:
   uses: actions/setup-node@v2: Sets up the specified version of Node.js, allowing caching of npm dependencies.
5. cache: 'npm' and cache-dependency-path: ./frontend/package-lock.json ensure efficient dependency caching.
6. Install Dependencies:
   run: npm install --legacy-peer-deps --save-dev-deps: Installs npm dependencies with legacy peer dependencies support.
7. Update Dependencies:
   run: npm update --legacy-peer-deps: Updates npm dependencies with legacy peer dependencies support.
8. Build Frontend:
   run: npm run build-linux: Builds the frontend component.
9. Testing Frontend:
   run: npm run test: Executes tests for the frontend component.

Checking Backend,Backend component testing with environment :

The backend component of our project is subjected to rigorous testing in a dedicated GitHub Actions workflow (backendCI). This workflow not only builds and sets up the backend environment but also initiates tests to verify the functionality of backend components.

The *Install Dependencies* and *Update Dependencies* steps ensure that the backend dependencies are correctly installed and updated. The subsequent steps start the backend server and run necessary tests. Unit testing for the backend is integrated into the build process, contributing to the overall reliability of our backend components.

Hence these 2 jobs(frontendCI, backendCI) run in parallel at the same time.

Backend Continuous Integration Workflow Steps:

1. Git Checkout:
   uses: actions/checkout@v3: Checks out the code from the repository.
2. Remove existing node_modules:
   run: rm -rf node_modules: Removes the existing node_modules directory.
3. Use Node.js:
   uses: actions/setup-node@v2: Sets up the specified version of Node.js.
4. MongoDB Setup:
   uses: supercharge/mongodb-github-action@v1.10.0: Sets up MongoDB using GitHub Actions.
5. Start Redis:
   uses: shogo82148/actions-setup-redis@v1.29.0: Starts Redis using GitHub Actions.
6. Install Dependencies:
   run: npm install --legacy-peer-deps --save-dev-deps: Installs npm dependencies with legacy peer dependencies support.
7. Update Dependencies:
   run: npm update --legacy-peer-deps: Updates npm dependencies with legacy peer dependencies support.
8. Start Backend Server:
   run: npm run start &: Starts the backend server in the background.
9. Test backend and Stop server.

3.4. Continuous Deployment

Continuous Deployment (CD) is the practice of automatically deploying code changes to a production environment after successful testing and integration. In our provided GitHub Actions workflow (deploy), we've implemented a CD for our project using Google Cloud Platform (GCP) for deployment.

Let's break down the code and explain how the deployment process works in detail:

Deployment Workflow Steps:

1. Checkout Code: uses: actions/checkout@v2: Checks out the code from the repository.
2. Print Deployment Trigger: run: echo "Deployment Triggered!": Prints a message indicating that the deployment has been triggered.
3. Setup Node.js Environment:
   uses: actions/setup-node@v1: Sets up the Node.js environment with the specified version.
4. SSH and Deploy Node App:
   uses: appleboy/ssh-action@v1.0.0: Uses the SSH action to connect to the deployment server.
5. host, username, key, port: SSH connection details retrieved from GitHub secrets.
6. script_stop: true: Stops the script on any error.
7. script: The deployment script executed on the remote server.
8. Changes directory to the project (/home/prasanthi_1/Telemedicine/).
9. Stashes any local changes.
10. Pulls the latest changes from the main branch with a recursive strategy.
11. For the backend:
    Removes existing node_modules and package-lock.json.
12. Installs dependencies and updates packages.
13. Restarts the PM2 process manager for the backend (pm2 restart 0).
14. For the frontend:
    Removes existing node_modules and package-lock.json.
15. Installs dependencies and updates packages.
16. Builds the frontend for Linux.
17. Cleans up the NGINX server's HTML directory.
18. Copies the built frontend files to the NGINX server's HTML directory.
19. Restarts the NGINX server (sudo systemctl restart nginx).

Deployment Flow:

1. The workflow is triggered after successful completion of the frontendCI and backendCI workflows.
2. The deployment server's SSH details are fetched from GitHub secrets.
3. The script connects to the server, updates the codebase, installs dependencies, and restarts the necessary services.
4. The deployment process ensures the production environment is updated with the latest changes from the main branch.

This comprehensive workflow automates the deployment process, enhancing the efficiency and reliability of delivering code changes to the production environment.

Integration Testing Workflow:

The integration-test workflow is designed to perform integration tests on the deployed application. It relies on the deployment workflow (deploy) to ensure that the latest changes are deployed to the production environment before executing the tests. Let's break down the workflow steps:

Integration Testing Workflow Steps:

1. Checkout Code:
   uses: actions/checkout@v2: Checks out the code from the repository.
2. Set up JDK 17:
   uses: actions/setup-java@v3: Sets up the Java Development Kit (JDK) version 17 using the Temurin distribution and caching Maven dependencies.
3. Set up Chrome:
   uses: browser-actions/setup-chrome@v1: Sets up the Chrome browser for testing, specifying the beta version.
4. The installed Chromium version and path are printed for reference.
5. Build with Maven:
   run: mvn clean install: Cleans the project and installs dependencies using Maven.
6. The working directory is set to testing/integration_testing.
7. Run Selenium Tests:
   run: mvn exec:java -Dexec.mainClass=integration_testing.i_t: Executes the Selenium integration tests.
8. If the tests pass ($? -eq 0), a success message is printed, and the workflow exits with code 0. If the tests fail, an error message is printed, and the workflow exits with code 1.

Integration Testing Flow:

1. The workflow is triggered after the successful completion of the deploy workflow, ensuring the latest changes are deployed.
2. The code is checked out, and the Java development environment is set up.
3. Chrome is configured for testing.
4. Maven is used to build the project in the testing/integration_testing directory.
5. Selenium tests are executed, and the success or failure of the tests determines the outcome of the workflow.

This integration testing workflow contributes to the overall robustness of the application by validating its behavior in a real-world environment.

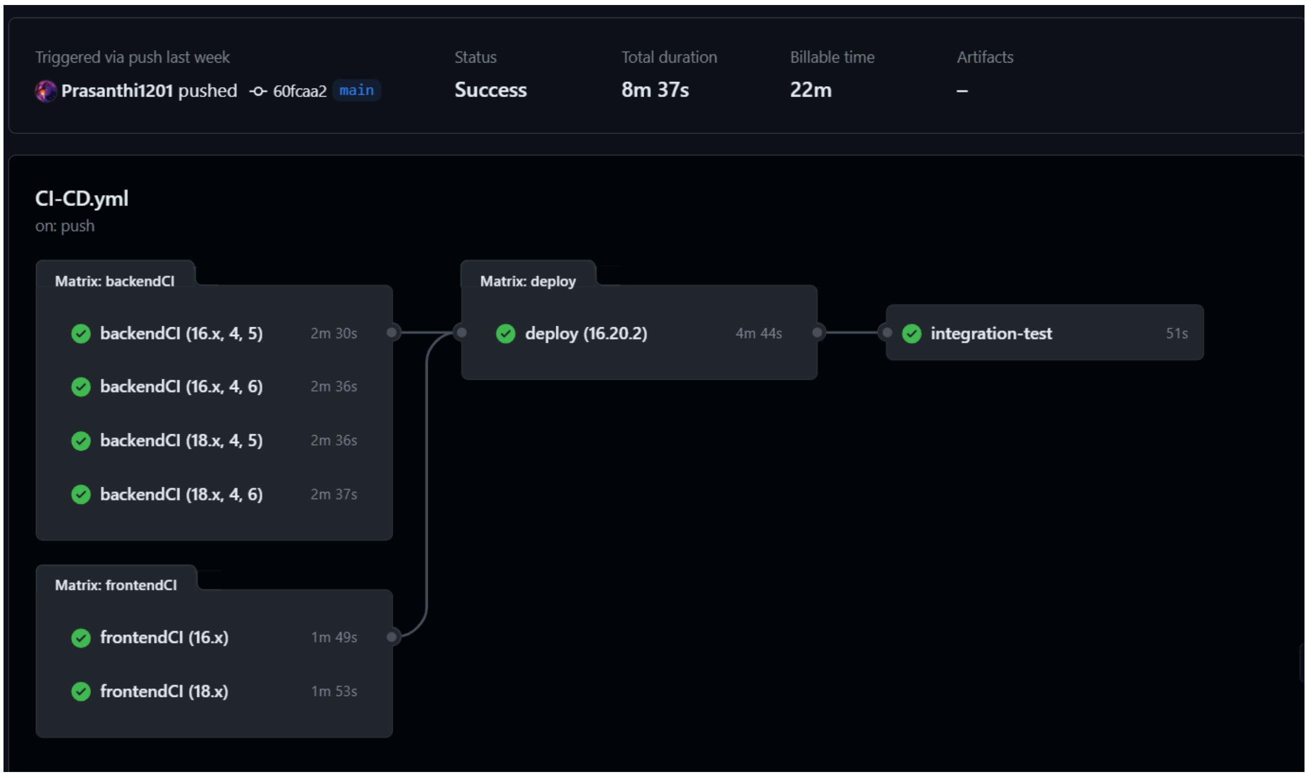
4. Observations

- Efficient Continuous Integration (CI):
    - The implementation of CI for both frontend and backend components ensures that code changes are promptly tested and integrated.
    - Usage of GitHub Actions allows for automated builds, dependency installations, and testing, contributing to a streamlined development process.
- Testing Strategies:
    - Front-end testing is conducted using Jest and React Testing Library (RTL), providing a comprehensive suite for unit testing React components.
    - Integration testing for the entire application is implemented using Selenium with Java, ensuring end-to-end functionality.
- Multi-Stage Continuous Deployment (CD):
    - Continuous Deployment is carried out in multiple stages, starting from the staging server and then progressing to the production environment.
    - Google Cloud Platform (GCP) is leveraged for production deployment, showcasing flexibility in deployment options.
- Reverse Proxy Configuration with Nginx:
    - Nginx is utilized as a reverse proxy, enhancing security and performance by serving as an intermediary between the application server and clients.
    - The reverse proxy setup with Nginx plays a crucial role in handling incoming requests and directing them to the appropriate backend services.
- GitHub Actions Workflow Coordination:
    - Workflow dependencies are well-defined, with the integration-test job dependent on the deploy job, ensuring that integration testing occurs after successful deployment.
    - The use of matrix strategies in workflows allows for parallelization, optimizing the execution of jobs across different configurations.
- Security Considerations:
    - The use of SSH keys stored in GitHub Secrets for deployment indicates a commitment to security best practices.
    - Continuous testing and integration contribute to the identification and mitigation of security vulnerabilities at an early stage in the development process.
- Diversity in Technology Stack:
    - The project incorporates a diverse technology stack, including Node.js, React.js for the frontend, MongoDB, Redis for the backend, and Java for Selenium-based integration testing.
    - The choice of technologies reflects a thoughtful selection based on the project's requirements.
- Documentation and Readability:
    - The provided workflow configurations are well-documented, enhancing readability for developers who may need to understand or modify the CI/CD processes.

    - Descriptive comments and clear naming conventions contribute to the overall maintainability of the workflows.
- Optimized Build and Deployment Scripts:
    - Build and deployment scripts are optimized, employing efficient practices such as caching dependencies to minimize redundant operations.
    - The use of PM2 for managing Node.js processes and restarting the backend server demonstrates a robust approach to process management.
- Scheduling and Parallel Execution:
    - The workflows are triggered on specific events, such as pushes to the main branch, ensuring that CI/CD processes are initiated at relevant times.
    - The parallel execution of jobs in workflows optimizes resource utilization and contributes to faster pipeline completion.

Overall, this project showcases a well-structured CI/CD pipeline with a focus on testing, security, and efficient deployment strategies. The utilization of GitHub Actions in conjunction with a diverse technology stack enhances the project's resilience and adaptability to evolving requirements.

The implementation of the CI/CD workflow in the Telemedicine project brings significant time efficiency to the development and deployment processes. In a manual scenario, where each step of integration, testing, and deployment would require individual attention and coordination, the entire process could take *several hours*, if not days as mentioned in the Figure below. However, with the automated CI/CD workflow, these tasks are seamlessly orchestrated, reducing the overall time required to a *mere 10 minutes*. This remarkable acceleration in the development lifecycle not only boosts productivity but also allows for swift adaptation to changing requirements, ensuring that the project remains agile and responsive to evolving needs.



Triggered via push last week	Status	Total duration	Billable time	Artifacts
Prasanthi1201 pushed 60fcaa2 main	Success	8m 37s	22m	–

CI-CD.yml
on: push

When contrasting the implementation of CI/CD in the development process with the traditional manual development approach, despite developers investing considerable time in code verification, issues often persist when changes are pushed to the source repository, the final observations are visualized with a table as follows:

Factor	**CI/CD**	**Manual Development**
Integration	Automatic integration multiple times a day	Manual integration, typically less frequent
Testing	Automated testing with each code change	Manual testing, often done in batches
Deployment	Automated, quick deployment to multiple environments	Manual deployment, often time-consuming
Time Efficiency	Rapid development and release cycles ~ 10 mins	Slower development and release cycles ~ hours
Error Detection	Early detection and resolution of errors	Errors may be identified later in the process
Parallel Execution	Parallel execution of tasks in workflows	Sequential execution, potentially leading to longer timelines
Consistency	Consistent and reproducible builds and deployments	Variability in manual processes may lead to inconsistencies
Scalability	Easily scalable for larger projects	Manual processes may become challenging to scale
Ease of Manipulation for Entrants	Simplified workflows with code-driven configuration	Requires in-depth knowledge and manual coordination
Adaptability	Easily adaptable to changing requirements	Manual processes may require significant adjustments
Documentation	Well-documented workflows enhance readability	Relies heavily on individual knowledge and may lack documentation
Collaboration	Facilitates collaboration through shared workflows	Collaboration may be hindered due to manual dependencies
Resource Utilization	Optimized resource utilization through parallelization	Resource allocation may be less efficient
Feedback Loop	Short feedback loop with quick iterations	Longer feedback loop, impacting development speed

5. Learning Outcomes

Through the implementation of CI/CD in the telemedicine project, several key learning outcomes have been achieved. Firstly, the hands-on experience has deepened my understanding of the critical role that automated testing and continuous integration play in identifying and resolving potential issues early in the development process. The utilization of GitHub Actions has honed my skills in creating and managing efficient CI/CD workflows tailored to the project's needs.

The implementation also highlighted the importance of a collaborative and integrated development environment. Working solo, I have learned to structure workflows that enable parallel execution, ultimately accelerating the development pace and minimizing the risk of integration conflicts. Engaging with testing tools such as Selenium, Jest, and React Testing Library has strengthened my proficiency in ensuring code quality and reliability.

Additionally, the incorporation of continuous deployment practices using Google Cloud Platform has provided valuable insights into automating the release process and maintaining consistency in deployment environments. This solo project has equipped me with a robust skill set in CI/CD tools and methodologies, essential for modern software development.

6. Conclusion

In conclusion, the solo integration of CI/CD into the telemedicine project has been a transformative experience, mitigating the challenges associated with traditional manual development. The streamlined automation of integration, testing, and deployment workflows has significantly improved efficiency and cultivated a mindset of continuous improvement.

This project's journey from manual testing and deployment to the seamless automation of critical processes underscores the importance of CI/CD in solo software development. The ability to detect and rectify issues early, work efficiently in a solo environment, and maintain consistency in deployment has elevated the overall quality of the telemedicine application.

Looking ahead, the insights and skills gained from this project will undoubtedly be applied to future endeavors, solidifying CI/CD as a fundamental practice in my solo software development approach. The success of this project underscores the pivotal role that CI/CD plays in meeting the demands of modern software development, ensuring solo agility, reliability, and optimal resource utilization.

7. Future Work

The incorporation of containerization represents a promising avenue for advancing the telemedicine project's CI/CD infrastructure. Containerization technologies such as Docker offer several benefits that can enhance the scalability, consistency, and deployment efficiency of the application.

One key future endeavor involves containerizing individual components of the telemedicine application. Breaking down the application into modular containers allows for encapsulation of dependencies, ensuring consistency across various development, testing, and production environments. This not only simplifies the deployment process but also facilitates seamless collaboration and version control.

Moreover, the adoption of container orchestration tools like Kubernetes could further amplify the benefits of containerization. Kubernetes provides a robust framework for automating the deployment, scaling, and management of containerized applications. Exploring Kubernetes orchestration can optimize resource utilization, enhance application resilience, and streamline the management of microservices.

The introduction of containerization aligns seamlessly with the CI/CD pipeline, as Dockerized containers can be integrated into the existing workflows. This integration streamlines the testing and deployment processes, making it easier to maintain and update containerized components.

As part of this future work, consideration should be given to configuring container registries to store and manage Docker images securely. Additionally, assessing the potential impact of containerization on the application's overall performance and resource consumption is essential to ensure a smooth transition.

In conclusion, future work on containerization presents an exciting opportunity to elevate the telemedicine project's CI/CD capabilities. Embracing Docker and Kubernetes can contribute to improved scalability, maintainability, and consistency, aligning the development process with modern best practices in software deployment and management.

8. References

1. Sai Alekhya Ganugapati, Sandeep Prabhu, "*Unifying Governance, Risk and Controls Framework Using SDLC, CICD and DevOps*", 2023 8th International Conference on Communication and Electronics Systems (ICCES), pp.1797-1802, 2023.
2. Ana Margarida Ferreira, Miguel A. Brito, José Lima, "*Continuous Inspection of Software Quality in an Automotive Project*", 2023 18th Iberian Conference on Information Systems and Technologies (CISTI), pp.1-6, 2023.
3. https://docs.github.com/en/actions/quickstart
4. Link to demo : - https://drive.google.com/file/d/1A-Xfme50DAfxfMI4_6JUhKjpUYc4FX3r/view?usp=sharing