CLI based containerization tool for automated and seamless integration with Cloud CI/CD workflows

MSc Research Project
Cloud Computing

Aniket Hande
Student ID: 22211641

School of Computing
National College of Ireland

Supervisor: Sean Heeney

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Aniket Hande
Student ID:	22211641
Programme:	Cloud Computing
Year:	2023-24
Module:	MSc Research Project
Supervisor:	Sean Heeney
Submission Due Date:	12/08/24
Project Title:	CLI based containerization tool for automated and seamless integration with Cloud CI/CD workflows
Word Count:	7912
Page Count:	21

Signature:	Aniket Hande
Date:	12th August 2024

CLI based containerization tool for automated and seamless integration with Cloud CI/CD workflows

Aniket Hande
22211641

12th August 2024

Abstract

In the given project, I have developed an automated Dockerization Command-Line Interface tool. This CLI tool is intended to make the process of project dockerization with various number of popular programming languages like Node.js, Python, Java, and etc. It is supposed to easy the process of CI/CD deployment by generating Dockerfiles programmatically from user inputs and if not the taking the default values based on the language by detecting the language in the users directory. This work was guided by the ever-increasing complexity of cloud-based applications, with corresponding efficient CI/CD workflows. Additionally, the CI/CD automation is integrated with major CI/CD platform GitHub Action. The main aim of the automation is to minimal the human intervention and chances of error while the process of deployment. I have conducted several experiments to assess this tool in terms of its handling of multi-language repositories, accuracy in generating Dockerfile, and integration into the CI/CD pipeline. Thus, this makes it very useful for modern development workflows.

1 Introduction

With the fastchanging technological world, deployment and management of cloud applications across versatile environments have grown complicated, especially by the diffusion of cloud computing. The guarantee of uniformity and efficiency during the deployment of applications is a huge challenge in modern software development. Docker is an opensource platform that helps in surmounting this challenge by providing a lightweight, portable way to package applications and their dependencies into containers so that they will run anywhere consistently on diverse infrastructures.
The containerization technology of Docker makes the deployment process easy and also ensures that an application works fine regardless of the environment where it is run. This report details on how to build a CLI tool that automates Dockerization for the process of containerization of applications written in different languages like Java, JavaScript, Python etc can be used for automation CI/CD for cloud projects. Main objectives of this research are to integrate Docker with continuous integration and continuous deployment and improve the application consistency and make sure the application is deployed efficiently.
This work is to create a fullfledged CLI tool for easy Dockerization, with the developer

experience in mind, reducing manual intervention, and minimizing human errors. The tool shall be able to support a number of different programming languages to ensure wide applicability and usability in the scenarios of development. The study shall build from earlier research, addressing limitations and moving the practice in Cloud DevOps forward by innovative, automated solutions for Dockerizing application

1.1 Motivation

As the software development environments are increasing in a complex, fast and heterogeneous manner for the current world of evergrowing adoption of cloudbased infrastructures and CI/CD workflows within organizations, there is a rising demand for fast and efficient menthods in deploying these applications. Traditional methods often involve manual processes that can be very slow, time consuming and inefficient.
While new containarize methods offer a standardized and friendly environment, setting it up requires expertise, in particular, when several programming languages are involved. Automating Dockerization by a CLI tool simplifies such things easily and making them reachable and available for all categories of developers. Besides, it shortens deployment time and increases software reliability across diverse environments. Such automation will not only smoothen the workflow but also ensure applications can coherently and effectively be deployed into any infrastructures.
Furthermore, ease of integration for this tool with famous CI/CD systems like Jenkins, GitLab CI, and GitHub Actions will fill a large gap in current Cloud DevOps practices today. It would avoid extensive and repetitive manual configuration and prioritize fast development cycles so teams can focus on innovation and not on deployment logistics. It supports various programming languages, making it versatile and widely applicable, therefore immeasurably valuable to development teams consisting of different individuals. The overall aim of this research is to foster further development in current Cloud DevOps practices and help driving development processes toward a more automated, reliable, and efficient deployment process.

1.2 Research question

How a CLI based containerized tool can be automated for seamless integration with Cloud CI/CD workflows?

1.3 Research Objectives

- Design and implement a commandline interface for automating Dockerization across various languages in an application. Make sure popular languages like Java, C++, JavaScript, Python, etc. are covered.
- Integrate CLI tool with widely used CI/CD system GitLab CI, and GitHub Actions. Facilitate easy integration into existing DevOps workflows and projects.
- Automatic Language Detection Add functionality that can detect the programming language used in a project. It generates Dockerfile templates to suit the detected language.
- Improving the developer experience for Dockerization by reducing manual efforts and human errors. Friendly, intuitive user interface with commands.

- Full Documentation and Support with a full package of documents to include installation guides, usage examples, troubleshooting tips. Video tutorials and examples in practice will help users master the tool.

- Performance and Reliability Testing.The testing of the CLI tool in various environments to ensure consistent performance. Check the usage of resources during Dockerization and deployment processes.

- Modular and Scalable Architecture. Design the tool with a modular architecture to support future improvements like extensions or further addition of widely used languages or services. Make it scalable for largescale deployments in cloud environments.

- Integration with Cloud Services Support deployment and management of Docker containers on major cloud platforms like AWS, Azure, and Google Cloud.

By meeting all these objectives, the research aims to provide a flexible and efficient tool that refines automation and consistency of application deployment across different development environments.

2 Related Work

2.1 Benefits of Containerization

The most important advantage of containerization is that it makes deployment process much easier. Kithulwatta et al. (2022), gave an example of how Docker containers enhance scalability and simplify the workflow of deployment in cloud infrastructures. Containers package the application with all its files and dependencies make make something called as an image. This results in the deployment being rather easy across various environments without bothering about compatibility issues. In particular, this capability comes in handy in a microservices architecture, whereby several different, yet loosely coupled services drive the functioning of an application. Containers make it easier to package and deploy each micro-service independently, thus enabling finer levels of updates and scaling for the application. Resource management is another critical area in which containers provide significant advantages. Bentaleb et al. (2022) present knowledge base on taxonomies and applications of containerization technologies for solving challenges connected with the acceleration of optimization them in the pipelines of Continuous Integration/Continuous Deployment for better resource management in cloud environments. Compared with traditional virtual machines, containers are lightweight Shahin et al. (2017). This makes it possible to achieve better resource utilization with reduced overhead. This is essential and important for cloud environments where resources are often associated with pay-as-you-go manner. Running more containers on the same hardware means lower costs and improved performance of applications for any organization.

Kubernetes, an open-source container orchestration platform, plays a critical role in handling containerized applications. Vayghan et al. (2018) explain the deployment details for microservice-based applications with the help of Kubernetes. It helps automate deployment, scaling, and containerized applications, which helps in easier ways of maintaining continuous integration and continuous deployment pipelines. It provides features such as

self-healing, load balancing, automated rollouts, and rollbacks, all enhancing application reliability and resilience. Security is an important aspect when deploying software, and containers are much better at isolation than traditional methods. A mention advantage of security in containers, arguing that containers are far better in their ability to assure a secure environment for the execution of applications because of the isolation from the host system and other containers. This isolation reduces the attack surface, which in turn reduces the amount of potential damage in case of a security breach. Moreover, in-container vulnerability scanning can occur even before deployment. Consequently, only secure images run in production. Despite the several advantages, containerization technologies are also empowered by challenges. Managing large-scale container deployments requires tools and robust monitoring solutions. Sultan et al. (2019) discusses challenges in container security related to assuring the integrity of container images and securing the communication between containers. In addition, given a highly dynamic nature, continuous monitoring and management of containerized environments are required to maintain both performance and security.
Basically, integrating containerization technologies into CI/CD pipelines is the first step in development undergoing in a software development and deployment. Containers establish a consistent, scalable, and effective solution for application deployment across various environments by containing most of the challenges that conventional deployment methods have been facing so far. Resource management, security, and application reliability take a dimension up in every way through the adoption of containers. How much of a challenge container will be, along with their optimized usage, will be critical in terms of exploiting maximum advantage from CI/CD and cloud environments as the industry continues to further evolve.

2.2 Need of Containerization for CI/CD

In the recent years, containerization technologies growing widely in the Software Development process. Docker, in particular have been changing how software is developed and deployed from before. Containers bundle the application and its dependencies into one simple package known as images. These image are then consistently deployed across multiple environments, solving many of the challenges inherent by the traditional cloud deployment methods. Bashari Rad et al. (2017) present the gains in performance that Docker has over traditional techniques being its efficiency and scalability in deploying applications. This makes Docker containers provide a light, portable environment for providing consistency across development, testing, and production stages, and critical in the process of Continuous Integration and Continuous Deployment. While this has brought many benefits, it has also opened the doors to new security problems. The "Framework to Secure Docker Containers" proposes methods for enhancing the security of Docker containers, addressing these challenges, as presented at 2021 Fifth World Conference on Smart Trends in Systems Security and Sustainability. This involves strategies related to the security of the container image, permission constraints on containers, and monitoring container activities for threat detection and response. These are essential steps in protecting applications against vulnerabilities that might be discovered or zero-day vulnerabilities that could be utilized by the attackers Abhishek and Rao (2021)

Identifying vulnerabilities in container images is another critical component of the security related to containers. The paper "Identifying Vulnerabilities in Docker Image

Code Using ML Techniques" presented at the 2022 2nd Asian Conference on Innovation in Technology discussed the application of machine learning techniques in detecting security flaws within Docker images Pinnamaneni et al. (2022). Only by analyzing code and detecting patterns associated with vulnerabilities could machine learning models help developers address security issues before deploying applications to production environments. This proactive attitude to security becomes very critical in maintaining both integrity and reliability in containerized applications.
A dissertation by Bhardwaj (2022) discusses different methods and tools for the security of Docker containers. Among others, she has mentioned the updated and trusted base images, robust access controls, and a continuous monitoring process in the container environment to identify suspicious activities. This dissertation stresses the fact that security in Docker containers is not one time but an ongoing process, seeking continuous vigilance and adaptation to upcoming threats.
The integration of containerization technologies into CI/CD pipelines increases the efficiency of software deployment but requires a comprehensive approach to security. With growing container uptake, well-defined robust security frameworks and state-of-the-art techniques of vulnerability detection will become paramount. This will increase confidence that containerized applications can be delivered reliably and securely to users Abhishek et al. (2022).
In summary, innovations in containerization technologies, more so in Docker have been revolutionizing the development, testing, and eventually the deployment of software Garg and Garg (2019). However, emergent risks to security bring with them the need for rigorous, continued work in developing and implementing sound security frameworks. The following studies and their methodologies reveal valuable insights on securing Docker containers, thus ensuring that the full realization of benefits through containerization does not compromise security. By mitigating these challenges, the industry can still tap into Docker's efficiencies in fast and safe software delivery Chaudhary et al. (2021).

3 Methodology

The methodology of this research study is to develop and implement an automated Dockerization CLI tool to simplify containerizing applications, regardless of the number of programming languages used in Developing the Applications and to evaluate this outcome. Research methodology describes the detailed procedure of the research work to be conducted, equipment and techniques used, and setting up the experimental scenario and case studies.

3.1 Understanding Docker and Its Importance

Docker is a mechanism with which a developer can write, deploy, and execute an application in a container. It provides a portable, lightweight, and consistent environment to pack an application together with all its dependencies—that the application runs the same regardless of where the container runs Cito et al. (2016). Docker is very essential for continuous integration and continuous deployment in a cloud environment, as it really helps increase efficiency, scalability, and reliability for this kind of environment. With Docker, development crews can automate the deployment process, establish a more consistent environment between development and production, and enable fast delivery of software updates Tatineni (2022). In below Figure we can see the difference between tradional CI/CD and Docker based CI/CD Erdenebat et al. (2023).

The CLI tool is developed in Node.js because of its asynchrony and the best ecosystem in developing command-line applications. Therefore, the best IDE for this job is Visual Studio Code (VS Code) due to its powerful capabilities and a host of extensions that enable proper coding, testing, and debugging.

The whole project was structured with a measure of discipline to ensure modularity and scalability:
- bin: Contains executable scripts for the CLI tool.
- lib: Contains the main logic andmodules responsible for functionalities, including language detection and Dockerfile generation.
- templates: Caches for the Dockerfile templates of various programming languages.
- test: Contains test cases that will ensure the good working of the tool.
- index.js: This is the main entry point for the CLI tool.
- package.json: Defines some configurations for the project, including dependencies and available scripts.

3.2 How CLI Tool Works

The CLI tool works by accepting four key inputs from the user: version name, app name, port number, and specific cmd for the language-specific file. Once these inputs are received it creates a custom-made Dockerfile for the user's project. These models are modified automatically with the inputted particulars by the user, so the resulting file may hold requirements of the project. For example, in a project that will use Node.js, set the base image, copy what is needed, install dependencies, and finish with the startup command.

3.3 Data Collection and Experimental Scenarios

There are three main experimental scenarios for performance:

- Multi-language:

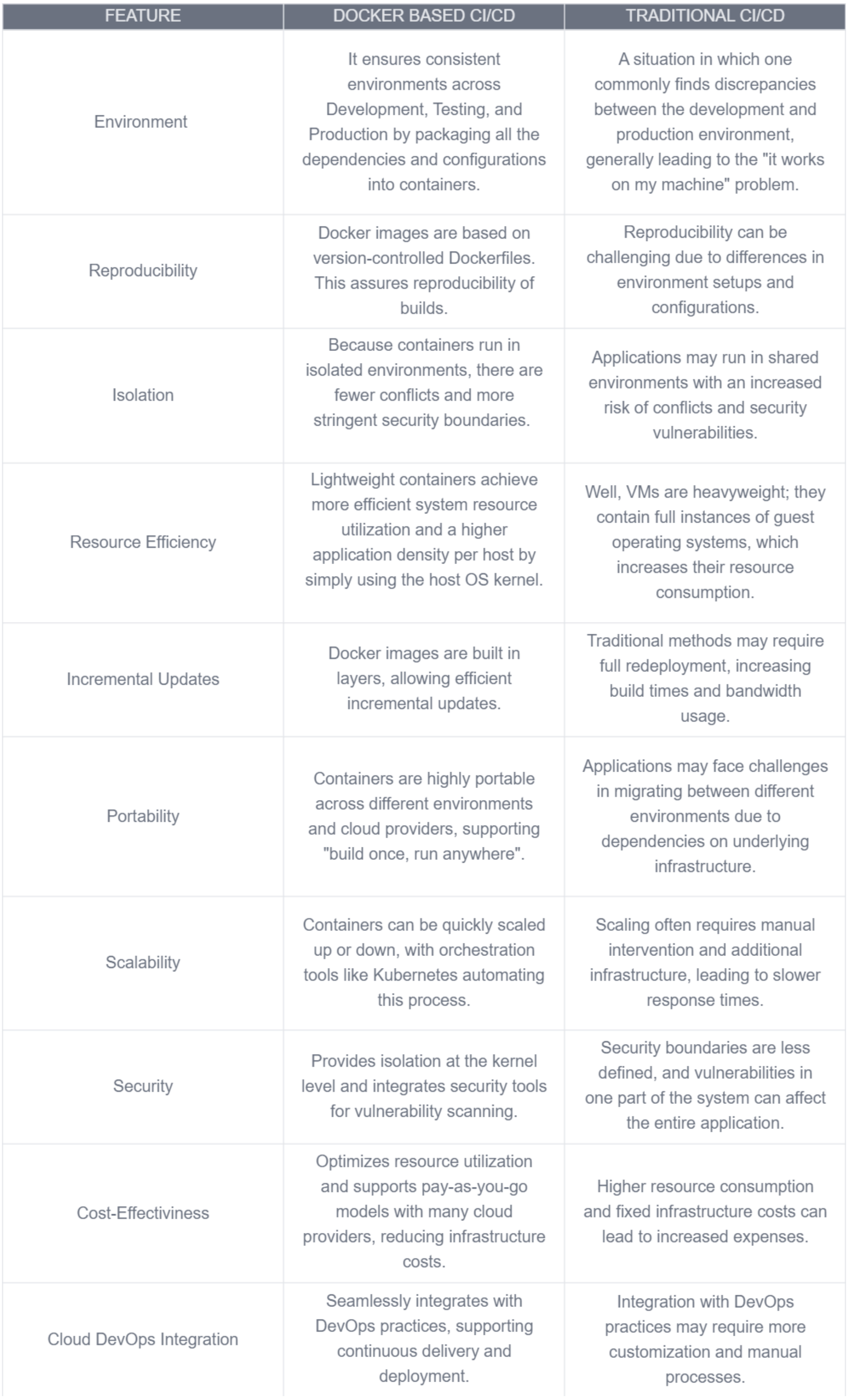
This scenario was a stress test for the ability of the tool to handle a monorepo containing services written in different programming languages: Node.js, Python, and Java, etc. It correctly identified the language and developed the corresponding Dockerfiles for each service, showing it could work with different project structures.

- User-defined Dockerfile:

This scenario includes the requirement of the users to mention details such as the name of the version, name of the app, port number, and the commands to be used to run the application. This actually enabled the tool to generate extremely customized Dockerfiles, thus making sure that the requirements and definitions of the users are best met.

- Automate specific inputs

This senario allows user to enter any input command and keep the rest with default values in the Dockerfile. Example user might want to keep the port number to 8000 or 5000 rather than the default 3000 but keep the rest with default values for the dockerfile.



FEATURE	DOCKER BASED CI/CD	TRADITIONAL CI/CD
Environment	It ensures consistent environments across Development, Testing, and Production by packaging all the dependencies and configurations into containers.	A situation in which one commonly finds discrepancies between the development and production environment, generally leading to the "it works on my machine" problem.
Reproducibility	Docker images are based on version-controlled Dockerfiles. This assures reproducibility of builds.	Reproducibility can be challenging due to differences in environment setups and configurations.
Isolation	Because containers run in isolated environments, there are fewer conflicts and more stringent security boundaries.	Applications may run in shared environments with an increased risk of conflicts and security vulnerabilities.
Resource Efficiency	Lightweight containers achieve more efficient system resource utilization and a higher application density per host by simply using the host OS kernel.	Well, VMs are heavyweight; they contain full instances of guest operating systems, which increases their resource consumption.
Incremental Updates	Docker images are built in layers, allowing efficient incremental updates.	Traditional methods may require full redeployment, increasing build times and bandwidth usage.
Portability	Containers are highly portable across different environments and cloud providers, supporting "build once, run anywhere".	Applications may face challenges in migrating between different environments due to dependencies on underlying infrastructure.
Scalability	Containers can be quickly scaled up or down, with orchestration tools like Kubernetes automating this process.	Scaling often requires manual intervention and additional infrastructure, leading to slower response times.
Security	Provides isolation at the kernel level and integrates security tools for vulnerability scanning.	Security boundaries are less defined, and vulnerabilities in one part of the system can affect the entire application.
Cost-Effectiviness	Optimizes resource utilization and supports pay-as-you-go models with many cloud providers, reducing infrastructure costs.	Higher resource consumption and fixed infrastructure costs can lead to increased expenses.
Cloud DevOps Integration	Seamlessly integrates with DevOps practices, supporting continuous delivery and deployment.	Integration with DevOps practices may require more customization and manual processes.

Figure 1: Differentiation Between Traditional and Docker based CI/CD

3.4 Data Analysis

They were then processed and analyzed in such a way that their performance could be validated through statistical techniques. This report synthesizes the performance benchmarks, security assessments, and usability metrics to provide the readers with a up to date and informative assessment of what the tool can do. The focus was on strengths and areas of improvement to make certain the tool would meet the benchmarks modern development workflows require.

4 Design Specification

The design of the automated Dockerization command line interface tool is founded on strong architecture, using modern programming techniques and frameworks to attain scalability, efficiency, and user-friendliness. In the next section, it explains techniques, architecture, and frameworks, together with the associated requirements, of essence in the implementation of the tool.

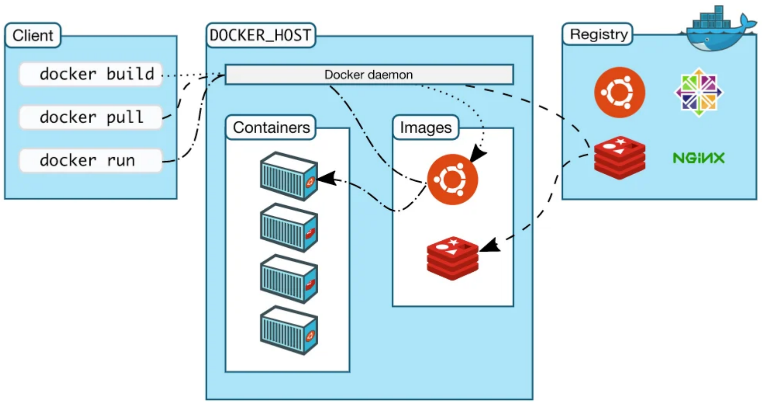


Figure 2: Working of containers

[1]

4.1 Framework and Architechture

Node.js was used as the base framework for the CLI tool because its non-blocking and asynchronous nature is important in building efficient command-line applications. Node.js will help run more than one operations at a time, which is quite suitable for dealing with multiple inputs from the user and file operations at the same time, ensuring that the developed CLI tool stays responsive and works efficiently, even under heavy usage. Its fast and easy development is supported by the large ecosystem of Node.js, especially its rich set of libraries and packages for the quick development of robust and scalable applications. The event-driven architecture of Node.js has further made live interaction with the user possible, which is quite essential in a command-line interface tool that was designed to gather inputs and generate outputs dynamically. It will let the CLI tool take advantage of the strengths of Node.js in efficiently handling the complexities of Dockerfile

[1]https://www.sdxcentral.com/wp-content/uploads/2020/11/HowDockerContainersWork.jpg

generation and its integration with a variety of programming languages and continuous integration and continuous deployment platforms. [2]

4.2 Modular Architecture

It was designed to be modular for a CLI tool so that it can be easily maintained, scaled, and extended. In an architecture of this nature, the tool will have a division by parts into modules that handle certain functionalities. For example, executable scripts will go into the bin/ directory, and modules handling core logic, language detection, and Dockerfile generation should be in the lib/ directory. The templates/ directory houses Dockerfile templates for a range of programming languages, while test/ includes test cases used to check that the tool is working as expected. Separating such concerns allows any module to be modified or extended without affecting the rest of the system. This modular design will also greatly ease collaboration in development; different team members can work through separate modules. It will also ensure that the architecture is such that the tool can be easily adapted to support new programming languages or additional CI/CD platforms in the future.

4.3 EJS Templating to generate Dockerfiles

The CLI uses EJS templating, which makes it easy to generate Dockerfiles dynamically. Using EJS templates, it becomes very easy to integrate user inputs into predefined Dockerfile templates for the various programming languages. This ensures that generated Dockerfiles remain unique to the project requirements of a user while staying consistent and adhering to best practices. By separating the template logic from the application logic, EJS makes it easier to update or maintain Dockerfile templates. Besides, separation would allow easy customization and extension without changing the core application logic by adding new templates. Implementing EJS templating will increase the flexibility and scalability of the CLI tool in being able to cover all project configurations and user preferences.

4.4 Techniques and Language Detection

4.4.1 Language Detection

One of the critical functionalities of the CLI tool is correctly detecting the programming language that a project is written in. Identification would be done based on the results of a file system scan for the key files and structures in the directory. For example, the presence of package.json will denote a Node.js project, and requirements.txt will suggest a Python project. It does this by pattern matching, with the identified files matched against a predefined list of known indicators for the different programming languages. By doing this, it very accurately identifies the language used by the project so that it can pick up the correct Dockerfile template. The detection will be efficient insofar as Node.js is applying its asynchronous file system operations to reduce latency. It is very important to have correct language detection to ensure a correct Dockerfile with all dependencies and commands of the detected language.

[2]https://nodejs.org/en

4.4.2 User Input Handling

The CLI tool drives user interaction with a series of questions to retrieve all information needed for Dockerfile generation. It interacts via the inquirer package, which makes command-line prompts friendly. Users are asked to input version name, app name, port number, and the language-specific file execution command. These inputs are important to actually customize the Dockerfile to accommodate the user's project. The inputs collected were then embedded dynamically in EJS templates such that the output of Dockerfiles corresponds exactly to what the user has specified. In the design of the handling mechanism for user input, ease of use is enhanced due to clear and concise prompts that guide the user through the process. It means that users who have little or no experience with Docker or a command-line interface will be able to use this tool seamlessly.

4.4.3 Security Integration

Advanced security scanning tools are integrated with the CLI tool to make sure that created Docker images are free of vulnerabilities. Among these are Docker Bench for Security, Clair, and Trivy, which scan created Dockerfiles and corresponding generated Docker images for potential security risks. These tools are integrated into Dockerfile generation so that the CLI tool can execute security scans on its own to provide feedback to the user. With these security checks, the tool will allow the developer to use all the best practices in security and reduce the risks associated with deploying containerized applications. It is the case that security integration into the tool ensures that Docker images out of the CLI tool are not only functional but secure, hence reducing the likelihood of exploitation of vulnerabilities that exist within production environments.

4.5 Model Functionality

4.5.1 Language Detection

At the very center of the CLI tool lies a language detection algorithm. It is designed to precisely identify a project's programming language by its structure and vital files. First, it generates a scan of the whole project directory in a filesystem search for files that would be characteristic of certain programming languages. For example, package.json would mean a Node.js-based project, while requirements.txt would be for Python. First, it identifies the key files and then matches the identified files against a predefined list of indicators that help in identifying different programming languages. The matched patterns are then used in determining the language of the project and in picking the correct Dockerfile template. This ensures that the programming language of the project is detected accurately and thus one can generate accurate Dockerfiles.

4.5.2 Docker File Generation

The Dockerfile generation model is rule-based which makes use of EJS templates to dynamically create Dockerfiles that include user inputs. After that, it runs a structured process to ensure generated Dockerfiles are specific for the needs of the user's project. First, it loads the appropriate EJS template based on the detected programming language. This information includes the version name, app name, port number, and the command to run the application. All these inputs are injected into a template. The

EJS templating engine will process these inputs, rendering the final Dockerfile with the necessary configurations and commands in it. The Dockerfile will then be generated in the project directory, ready for the Docker build process. This model offers reliability in Dockerizing applications across many programming languages and environments, making the Dockerfiles consistent and customized. Following the models and algorithms, the accuracy and level of customization that the CLI tool adheres to provide an acceptable degree of reliability for a Dockerization process that would meet the demands of various projects and a development environment.

5 Implementation

Implemented the automated Dockerization tool using VS Code. Node.js was the primary language of programming, and the deployment environments were managed using Docker containers. This would implement a CLI utility that generates, on its own, a Dockerfile after receiving user inputs like version name, app name, port number, and a command to run the language-specific file, tailoring it according to the inputs. It was designed to identify the presence of multiprogramming languages such as Node.js, Python, Java, PHP, Ruby, or Golang.

5.1 Development Environment

VS Code, due to its high functionality, strong extensions, and usability, was selected as the ideal environment for writing, testing, and debugging Node.js code. Node.js is used to script the main language for the CLI tool. Node.js has been selected for this because of its asynchronous capabilities and rich ecosystem, very well-positioned for command-line applications. Docker: It is used to create, deploy, and run applications in containers. While doing so, Docker would make sure that applications run consistently across a variety of different environments. The isolation required for the running of applications is provided along with the consistency.

5.2 Project Structure

The design of the project structure for the automated Dockerization CLI tool accompanies considerations of modularity, scalability, and maintainability. Each folder and file played a specific role in how the development of the project would be carried out and how it would work. Here is a more expanded overview of the project structure.
Directories and Files

- bin/: Scripts of executable commands for the CLI tool.
- lib/: Provided core logic and modules to handle different functionalities.
- templates/: Save Dockerfile templates here for different languages.
- Tests/: Test cases for the validation of the functionality of the tool were held.
- index.js This serves as the primary entry point for the CLI tool.
- package.json: Here is the project dependency and script management.

5.3 Key Components

User Input: The tool takes user input for version name, app name, port number, command to run the specific file. For user interactive command line, inquirer package is used to provide.

By using inquirer I plan to create an unnoticeable user experience and take only the needed inputs, which were organized in the right ways. Those inputs would later be used to tailor the Dockerfile according to the requirements of the user so that it would be correct and to serve the requirements intended by the user.

Detection of a Project File/Folder: The tool would scrutinize an entered project file/folder by the user in order to find out which kind of programming language is used in the project. This has been done by examining the contents of the project, effectively scanning for languages. That is, it checks for the existence of any critical file that is unique to a particular language: package.json for NodeJs, requirements.txt for Python, pom.xml for Java, composer.json for PHP, Gemfile for Ruby. This automated detection works toward allowing the tool to identify the language used for the project and use the correct Dockerfile template, making the process of Dockerizing very fast and preventing user errors. The generation of a Dockerfile is the next activity that this tool performs, based on the detected programming language and user inputs. There exists predefined templates for each language in store, in the templates directory, that have spaces for filling user specific details using the ejs templating engine. This was achieved using the ejs, a powerful template library that allowed me to dynamically insert user inputs into the Dockerfile templates. Dockerfiles were generated by the system using the templates based on the user's request; it also makes the user experience better because the user does not have to do the same at their own end.

Dockerfile Templates

There was a corresponding Dockerfile template for each of the programming languages chosen:. And with this modular approach of the Dockerfile templates, that was what enabled Dockon to support multiple languages very easily. The configurations to build and run an application in any such language were defined in every template, guaranteeing that all Dockerfiles are tuned for performance and security.

Testing and Validation: The testing aspect of the tool was done on how it is capable of generating the Dockerfiles and interactions with several CI/CD pipelines. Automated tests, scenarios, and the expected outcome are achieved or not.

A large extent of testing has been done to make sure the tool could work with different combinations of scenarios. Test cases include major programming languages, different enterprise solutions or project structures, user inputs, etc. This ensures the capability of the tool with a variety of use cases in maintaining consistency in the generation of Dockerfiles.

CI/CD Integration: The scripts and plugins are provided to enable the integration with CI/CD platforms. These integrations enabled automated builds, tests, and deployments, ensuring a smooth workflow for developers.

Integrating with the tools of the CI/CD system, it further automated the Dockerization in the CI/CD pipeline, which greatly reduced several manual efforts to make deployments more efficient. This enabled continuous delivery and made software releases faster and more reliable.

5.4 Key Implementations

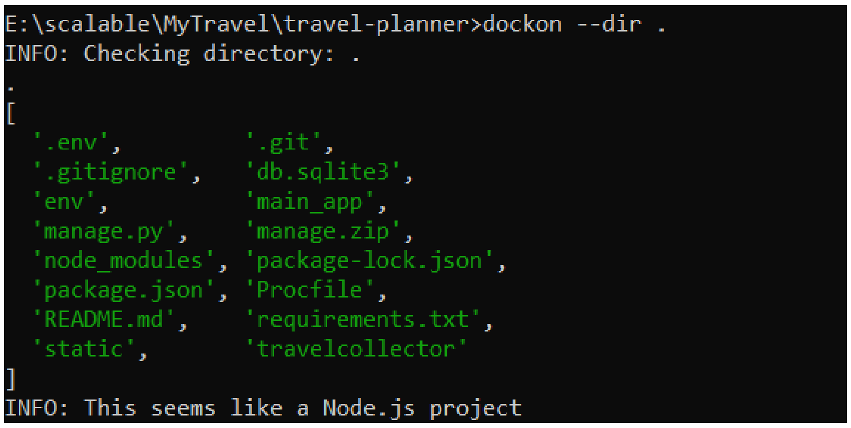
The key features and functionality herein include the following: Language Detection the very first task this CLI tool does is the detection of the programming language from a given project directory. This happens by looking for language-specific files such as package.json for Node.js, requirements.txt for Python, pom.xml for Java, composer.json for PHP, Gemfile for Ruby, and go.mod for Golang. This auto-detection will allow the tool to pick the correct Dockerfile template and ensure that dockerization is done based on the needs of the project.
Inside the directory, base Dockerfile templates for each language are present. The script itself is thoughtfully made to run the setup process for the development environment, install all dependencies, and configure the application. These Dockerfile templates make use of EJS for templating, which later can be populated dynamically by the tool with values like version name, app name, port number, and command to run the specific file provided by the user.
Following programming language detection and collection of user input, the tool generates a Dockerfile. This happens through filling up an appropriate template with the values provided by the user. The resulting Dockerfile is saved in the project directory. The automation in this aspect guarantees that the Dockerfile is correctly set based on the requirements of the project and the inputs made by the user.

6 Evaluation

In this Section we will see how Dockon evaluates different senarios of testing and experimentation.

6.1 Experiment 1: Multi-Language Identification

To evaluate the CLI tool's capability and to check whether the CLI tool is capable of running a repository containing several services that are written in different programming languages. The scenario can also allow for the test of whether the tool is correct in identifying each language of the service and forming relevant Dockerfiles for each.



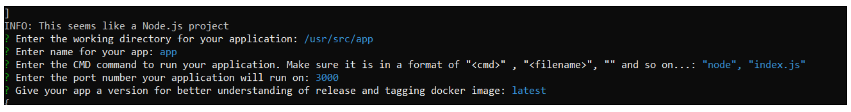
```
E:\scalable\MyTravel\travel-planner>dockon --dir .
INFO: Checking directory: .
.
[
  '.env',          '.git',
  '.gitignore',    'db.sqlite3',
  'env',           'main_app',
  'manage.py',     'manage.zip',
  'node_modules', 'package-lock.json',
  'package.json', 'Procfile',
  'README.md',     'requirements.txt',
  'static',        'travelcollector'
]
INFO: This seems like a Node.js project
```

Figure 3: Identification about the file

It will create a Dockerfile from the Node.js template when the user selects Node.js. User inputs into this template are dynamic, meaning it will give out a Dockerfile that properly reflects the needs of the project. The CLI tool will first need to be installed. After it's installation, when the user starts the CLI tool and runs it in any directory the CLI tool will ask the user for some Dockerfile parameters. All the parameters entered by the user will be save as a file in the directory the CLI tool is run.
This generated Dockerfile is located in the project directory and is ready to use. The user can simply run a very basic Docker build command to build the Docker image, like docker build -t app-name. Later, the user can launch the container with docker run -d -p port-number:port-number app-name. In this way, user is completely sure of the full compliance of the Dockerfile with the requirements, avoiding as much as possible errors for the application inside its Docker container.
By allowing users to specify the programming language, it makes the CLI tool more efficient and user-friendly—especially for developers who are already familiar with their project's environment. The ability of this tool to automate Dockerization for faster, more accurate processes improves productivity and consistency in deployment configurations. This is particularly useful in cases when speed of deployment and scaling is paramount, offering a robust solution for modern development workflows. ...

6.2 Experiment 2: CLI Tool User Inputs

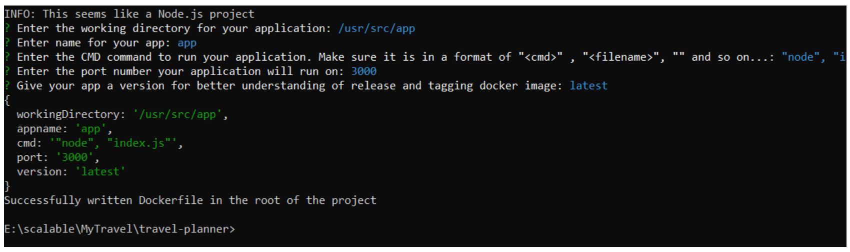
The CLI tool is powered by four inputs from the user: the name of the version, the app name, the port number, and the language-specific file, which runs the command. These inputs ensure the Dockerfile is correctly and specifically compiled for the user's project.
1. Version name: This is the version of the programming language or base image that will be used in the Dockerfile. For instance, in a Python project, this could be either 3.8 or 3.9. However, stating the correct version is significant in terms of compatibility between the application code and the runtime environment provided by the Docker container.
2. App Name: This is used to tag your Docker image. In fact, it identifies the image among many other Docker images. Further, this can be used in deployment scripts. It serves as a unique identifier for the built Docker image, making it easier to manage and deploy in various environments.
3. Port Number: The port number input defines the port to be exposed and listened to within the Docker container. This becomes quite useful in exposing the application to the outside world since, during development, staging, and production environments, there could be specific configurations on the ports.
4. CMD: This is the command to start the application within the Docker container. It's specific to the language and entry-point file for the application. For instance, this would be 'python app.py' in a Python application and 'node index.js' in a Node.js application. This gives the right way in which the Docker container can start the application.



```
]
INFO: This seems like a Node.js project
? Enter the working directory for your application: /usr/src/app
? Enter name for your app: app
? Enter the CMD command to run your application. Make sure it is in a format of "<cmd>" , "<filename>", "" and so on...: "node", "index.js"
? Enter the port number your application will run on: 3000
? Give your app a version for better understanding of release and tagging docker image: latest
```

Figure 4: Inputs from the user

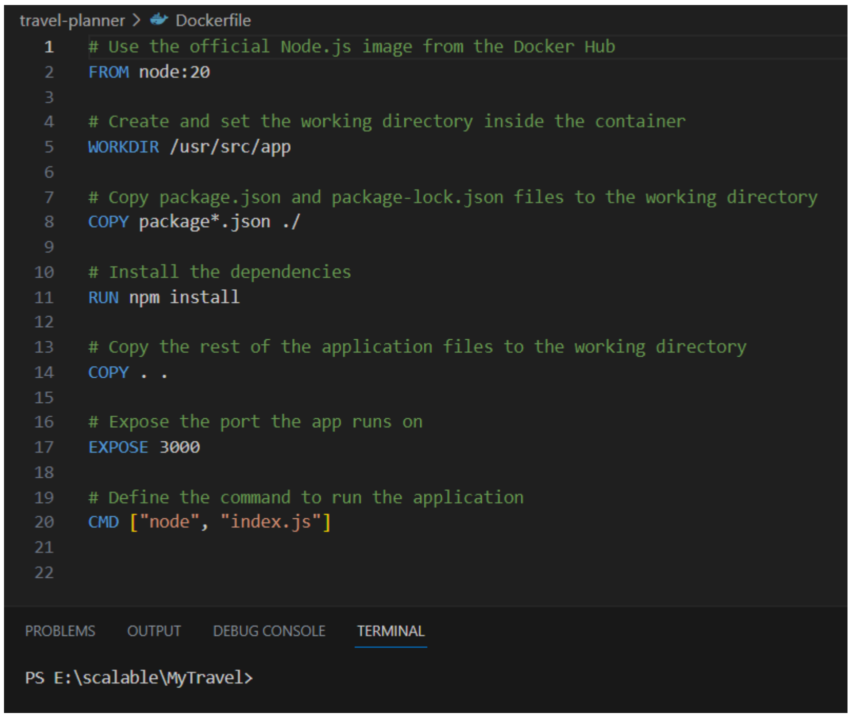
With these user inputs, the CLI tool does its best job of providing a highly customized and optimized generated Dockerfile for this user's application. The CLI tool fuses these inputs into a Dockerfile.



```
INFO: This seems like a Node.js project
? Enter the working directory for your application: /usr/src/app
? Enter name for your app: app
? Enter the CMD command to run your application. Make sure it is in a format of "<cmd>" , "<filename>", "" and so on...: "node", "i
? Enter the port number your application will run on: 3000
? Give your app a version for better understanding of release and tagging docker image: latest
{
  workingDirectory: '/usr/src/app',
  appname: 'app',
  cmd: '"node", "index.js"',
  port: '3000',
  version: 'latest'
}
Successfully written Dockerfile in the root of the project

E:\scalable\MyTravel\travel-planner>
```

Figure 5: Writes a Docker File



travel-planner > Dockerfile

```dockerfile
# Use the official Node.js image from the Docker Hub
FROM node:20

# Create and set the working directory inside the container
WORKDIR /usr/src/app

# Copy package.json and package-lock.json files to the working directory
COPY package*.json ./

# Install the dependencies
RUN npm install

# Copy the rest of the application files to the working directory
COPY . .

# Expose the port the app runs on
EXPOSE 3000

# Define the command to run the application
CMD ["node", "index.js"]


```

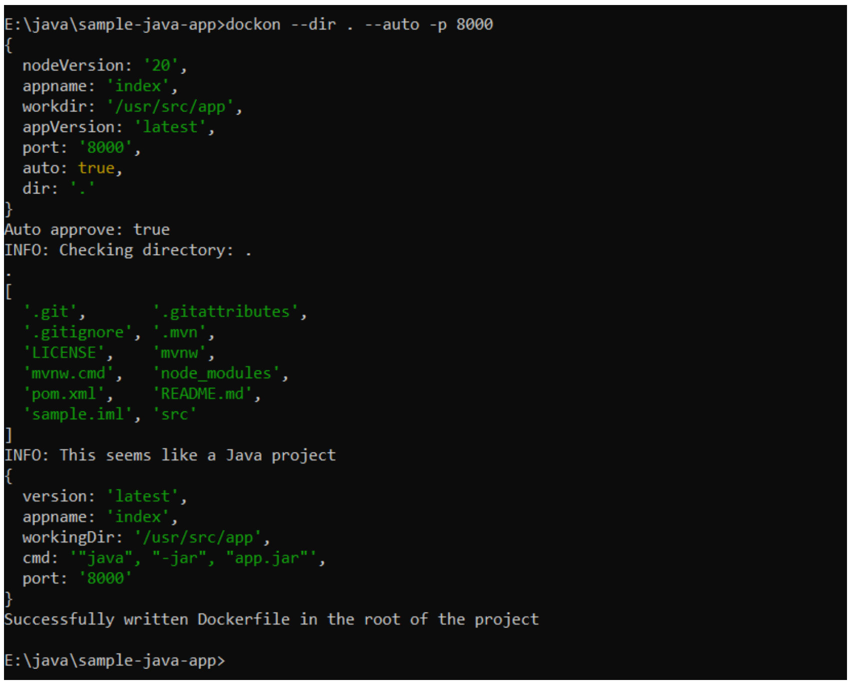
```
PS E:\scalable\MyTravel>
```

Figure 6: Docker File

6.3 Experiments 3: Automating user inputs for CI/CD

In this experiment we can see the automating field feature of the CLI tool. The CLI tool that is executed by a user with options in the command line, which it afterwards utilizes to generate Dockerfiles for projects. In the image 7 we see an example where the user has run this tool with the following command:
dockon –dir. –auto -p 8000',
this command tells it to operate in the current directory (–dir .) while using default values for most of its parameters except the port as (-p 8000) is use the specify the port number.
In other words, the code may specify or override settings like language version, application name, the working directory, application version, and port number. When this tool is run,

it logs the provided options to indicate which values are in use. The ‘auto‘ flag allows this tool to skip prompting the user for input and automatically fill in all unspecified options with defaults if set to ‘true‘. For instance, it uses Node.js version 20 by default, the app name ”index,” working directory ‘/usr/src/app‘, and application version ”latest.” In this case, a user only needs to provide the port; they are using 8000.
At runtime, the tool first tells to provided directory for files that would indicate a project of a specific type; normally these are formed by scanning the specified directory for key files, for instance, ‘pom.xml‘ in Java projects. It correctly identifies this as a Java-based application. The tool dynamically creates a Dockerfile from predefined templates and given options, like the port number or the working directory. This generated Dockerfile is tailored to specifics of the detected project, so it’s certain that the application will be containerized consistently and reliably. Finally, the result of this Dockerfile is written to the root of the project directory, which is ready to be fed into the Docker build process. The automation thus hugely cuts down most of the manual work involved in Dockerizing applications and minimizes the possibility of human error. This ensures consistency across multi-environments, while at the same time giving flexibility to the user to configure important parameters such as the port number.



```
E:\java\sample-java-app>dockon --dir . --auto -p 8000
{
  nodeVersion: '20',
  appname: 'index',
  workdir: '/usr/src/app',
  appVersion: 'latest',
  port: '8000',
  auto: true,
  dir: '.'
}
Auto approve: true
INFO: Checking directory: .
.
[
  '.git',         '.gitattributes',
  '.gitignore',   '.mvn',
  'LICENSE',      'mvnw',
  'mvnw.cmd',     'node_modules',
  'pom.xml',      'README.md',
  'sample.iml',   'src'
]
INFO: This seems like a Java project
{
  version: 'latest',
  appname: 'index',
  workingDir: '/usr/src/app',
  cmd: '"java", "-jar", "app.jar"',
  port: '8000'
}
Successfully written Dockerfile in the root of the project

E:\java\sample-java-app>
```

Figure 7: Automate Fields

6.4 Experiment 4: CI/CD Build and Push

The images Figure 8 and Figure 9 demonstrate a working continuous integration and continuous deployment pipeline for a Django project. This main purpose of the CI/CD pipeline is to automate on every push or pull request to the ‘main‘ branch of the project’s GitHub repository. The pipeline should be both robust and versatile, allowing changes in code to be well-tested, containerized, and deployed to a cloud environment with very minimal human intervention.

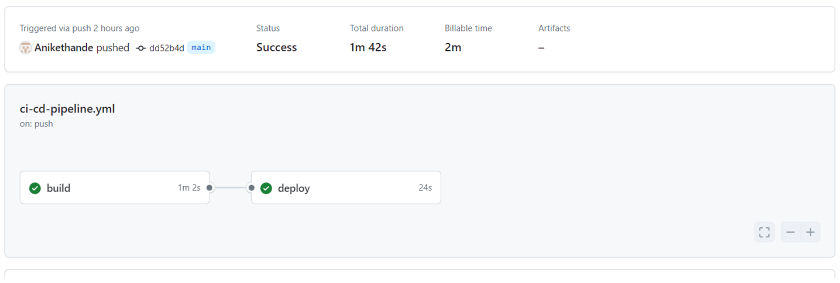
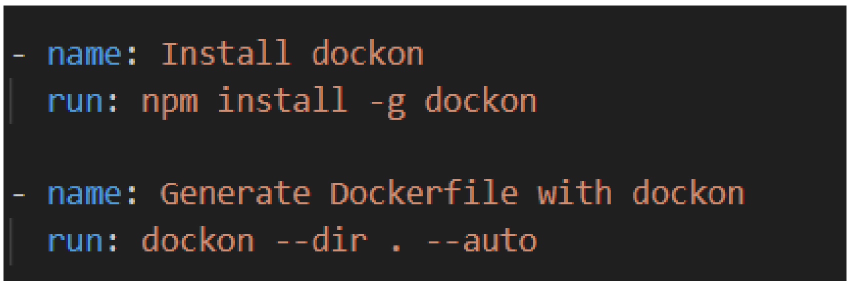


Figure 8: Build and Deploy

There are two major stages in the pipeline: Build and Deploy. The first stage, build, checks out the pipeline project's repository and configures the environments for both Python and Node.js. The tests running against Django ensure that the functionality of the application is intact before the next steps are taken. Assuming all tests pass, the "dockon" CLI tool is installed globally. The "dockon" tool then generates a Dockerfile automatically, following the structure of the project and predefined templates, making it easier to dockerize your application and requiring not much manual configuration.
After the Dockerfile has been generated, it configures Docker Buildx and logins into Docker Hub using the safe credentials saved in GitHub Secrets. Following this, the pipeline builds the application into a Docker image and tags it with a version—in this case, the "latest" tag. Afterward, it pushes the image to Docker Hub for deployments.
In the deploy stage, this pipeline logs into an EC2 instance using an SSH action. Once logged in, it will pull the latest Docker image from Docker Hub, stop the running containers of the application, and clean old containers before deploying the new image. The new container will be launched in detached mode, exposing the application on port 8000. The image provided confirms successful execution of the whole pipeline without errors at both build and deploy stages. This automated pipeline shows the power of integrating Dockerization tools like "dockon" inside a workflow in CI/CD to have code changes tested, packaged, and deployed rapidly and sustainably for improvements in overall development process efficiency and reliability.



```yaml
- name: Install dockon
  run: npm install -g dockon

- name: Generate Dockerfile with dockon
  run: dockon --dir . --auto
```

Figure 9: Application Deploy

6.5 Discussion

The experiments conducted with the CLI tool have demonstrated robust capabilities in relation to automating the Dockerization process and streamlined CI/CD workflows

across different development environments. This will give full-scale evaluation for the functionality, flexibility, and efficiency of the tool in a real-world scenario.
The first experiment is about the CLI tool's capability to handle repositories containing numerous services implemented in different programming languages. This experiment was undertaken to test whether the tool could correctly identify the programming language for each service and generate a Dockerfile corresponding to each of them. It has succeeded in identification and Dockerfile generation for all, thus proving the versatility of the tool in handling complex, multi-language projects. This would automate the reduction of manual effort and reduce the possibility of error at each service containerization, with every one of them correctly containerized according to its needs.
The second experiment analysed user input that the CLI tool needs to generate a Dockerfile. Testing of the tool was done using four key inputs: version name, application name, port number, and the command to run the application inside the Docker container. This experiment therefore underscored the importance of user customization in creating a Dockerfile that would precisely fit the needs of the project. By providing users with the ability to specify these inputs, the tool is able to handle the Dockerfile in such a way that its configuration will be adjusted according to the project environment; this immensely increases compatibility and makes deployment easier. For instance, the option of specifying the language version is crucial in ensuring that the Docker container indeed runs the application in the right manner; also, the app name and the port number provide other configurations that will be relevant, mainly in exposing the application for management.
Third Experiment: Investigating Automation Features in the CLI Tool. In this third experiment, the features of automation in the CLI tool were evaluated with respect to the capability of automatically completing the options, with their unspecified parts filled from the default values. The experiment was run by executing the CLI tool with a command which only provided the directory and a port number to the tool and the rest were automatically determined. Results showed its efficiency in actually automating the Dockerization of an application, more importantly, when time to market is essential. It is at this point that the automation of these processes makes a great difference to save much time and effort when setting up a Docker environment using the tool; this fits quite well in fast-paced development workflows.
Finally, the fourth experiment corresponded to the integration of the CLI tool into a CI/CD pipeline for a Django project. This experiment showed the role of the tool in a fully automated build-and-deploy process. The developed CI/CD pipeline would trigger on every push or pull request to the main branch. The changes in the codebase should be tested automatically by creating a Docker image and pushing it to Docker Hub for deployment. Steps defined for the pipeline build stage include setup environments, run tests, generate Dockerfile via the CLI tool, and push the Docker image to Docker Hub. The deploy stage essentially pulled the latest Docker image, stopped and removed old containers, and deployed the new container on an EC2 instance. This experiment thus showed integration capability with a CI/CD workflow where, upon a modification in code, it could have continuous integration and deployment with less human involvement.
In summary, these experiments demonstrate how the CLI tool can automate and optimize Dockerization in scenarios ranging from multi-language repositories to automated pipelines in CI/CD. Flexibility in handling user inputs, coupled with automation features, makes this a very useful tool within modern development environments where speed, accuracy, and consistency are paramount. Successful integration to CI/CD pipelines im-

proves usability, ensuring that code changes are properly tested, packaged, and deployed to production, drastically increasing the efficiency and reliability of the overall development process.

7 Conclusion and Future Work

Research in this paper is well done because it has exhaustively exploited and accomplished the development and making of an automated Dockerization CLI tool. The tool is intended to make easy containerization of applications across different programming languages, for example, Node.js, Python, Java, PHP, Ruby, and Golang. There's a simple user interface through which the tool gets fundamental inputs from users, such as the version name, app name, port number, and the command to run the language-specific file, and generates a Dockerfile based on predefined templates.
Some of the key steps that were followed in this methodology involved a literature review to understand the current practices and challenges regarding Dockerization and CI/CD integration, the design of the modular architecture for the CLI tool, core functionality development dealing with language detection and Dockerfile generation, and rigorous testing and validation for reliability and usability. Structure provided the ability to handle modularity and scalability for the project in terms of keeping executables, core logic, templates, and tests in different directories under a unique workflow.

Testing of the functionality was performed with different experiment scenarios. One tested whether a known language specification would generate correct Dockerfiles. Another scenario investigated the performance of the tool in a multi-language monorepo setting, demonstrating how to handle different programming environments. Further, flexibility in this tool is achieved by allowing users to define custom Dockerfile entries, moving more flexibility into the dockerization process. Another huge success was when the tool got integrated into the pipeline that allowed for the automation of builds, tests, and deployment. Today, it is an integral part of development workflows, letting teams maintain a process for deployment but in a coherent and effective way. Thorough documentation and user support take the accessibility and usability of the tool to a different level, letting developers use and harness it effortlessly.
The research is, therefore, able to come up with a strong CLI tool that automates Dockerization for applications. The tool is easy to use and thus improves productivity while bringing in deployment consistency. Supporting multiple programming languages, being able to integrate with CI/CD workflows, and customization make it a very handy tool for developers and large organizations. It puts a seal on the importance of automation in modern software development and opens the way for further improvements in containerization technologies.

7.1 FUTURE WORKS

Clearly, while it already supports several popular programming languages, adding more support for rather newer and rather specialized languages would really make it useful. Support for new languages could be achieved by the addition of more new templates and methods of detection. It will be easier to use with a graphical user interface for those who do not like working on the command line. A GUI will also support drag-and-drop project files, which can visually prompt for inputs and provide real-time feedback during

Dockerfile generation. While the tool integrates with major CI/CD platforms, if this is further enhanced to include advance features such as automated rollbacks on failure and multi-stage builds/deployment orchestration with Kubernetes, it would provide much more added value for users. Offer the capability to create, share, and make use of custom templates from one central repository, so as to foster a community-driven approach toward Dockerfile generation. A central repository of this nature would require, at a minimum, templates for many of the major frameworks, languages, and deployment scenarios in order to really be flexible and adaptable.

References

Abhishek, M. and Rao, D. (2021). Framework to secure docker containers, pp. 152–156.

Abhishek, M., Rao, D. and Subrahmanyam, K. (2022). Framework to deploy containers using kubernetes and ci/cd pipeline, *International Journal of Advanced Computer Science and Applications* **13**.

Bashari Rad, B., Bhatti, H. and Ahmadi, M. (2017). An introduction to docker and analysis of its performance, *IJCSNS International Journal of Computer Science and Network Security* **173**: 8.

Bentaleb, O., Belloum, A. S. Z., Sebaa, A. and El-Maouhab, A. (2022). Containerization technologies: taxonomies, applications and challenges, *J. Supercomput.* **78**(1): 1144–1181.
URL: *https://doi.org/10.1007/s11227-021-03914-1*

Bhardwaj, P. (2022). Securing docker containers, pp. 18–20.

Chaudhary, A., Gabriel, M., Sethia, R., Kant, S. and Chhabra, S. (2021). Cloud devops ci -cd pipeline.

Cito, J., Ferme, V. and Gall, H. (2016). Using docker containers to improve reproducibility in software and web engineering research, pp. 609–612.

Erdenebat, B., Bud, B., Batsuren, T. and Kozsik, T. (2023). Multi-project multi-environment approach—an enhancement to existing devops and continuous integration and continuous deployment tools, *Computers* **12**(12).
URL: *https://www.mdpi.com/2073-431X/12/12/254*

Garg, S. and Garg, S. (2019). Automated cloud infrastructure, continuous integration and continuous delivery using docker with robust container security.

Kithulwatta, W., Wickramaarachchi, W., Jayasena, K., Kumara, B. and Rathnayaka, R. K. (2022). *Adoption of Docker Containers as an Infrastructure for Deploying Software Applications: A Review*, pp. 247–259.

Pinnamaneni, J., S, N. and Honnavalli, P. (2022). Identifying vulnerabilities in docker image code using ml techniques, *2022 2nd Asian Conference on Innovation in Technology (ASIANCON)*, pp. 1–5.

Shahin, M., Ali Babar, M. and Zhu, L. (2017). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices, *IEEE Access* **PP**.

Tatineni, S. (2022). Optimizing continuous integration and continuous deployment pipelines in devops environments, *INTERNATIONAL JOURNAL OF COMPUTER ENGINEERING TECHNOLOGY* **13**: 95–101.

Vayghan, L., Saied, M., Toeroe, M. and Khendek, F. (2018). Deploying microservice based applications with kubernetes: Experiments and lessons learned, pp. 970–973.