Streamlining DevOps: Automating Testing and deployment of Kubernetes Environments

MSc. Research Project
Cloud Computing

Mohammad Amaan Shaikh
Student ID: X23186925

School of Computing
National College of Ireland

Supervisor: Dr. Ahmed Makki

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Mohammad Amaan Shaikh
Student ID:	X23186925
Programme:	Cloud Computing
Year:	2024
Module:	MSc. Research Project
Supervisor:	Dr. Ahmed Makki
Submission Due Date:	12/12/2024
Project Title:	Streamlining DevOps: Automating Testing and deployment of Kubernetes Environments
Word Count:	7400
Page Count:	21

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	12th December 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Streamlining DevOps: Automating Testing and deployment of Kubernetes Environments

Mohammad Amaan Shaikh
X23186925

Abstract

In this study we automated testing and deployment of a Kubernetes application on AWS Elastic Kubernetes Service (EKS) cluster, utilizing Helm charts for streamlined deployment and management. KubeScore was employed for configuration validation of Helm charts. The report generated by KubeScore was analyzed and more than 70% of critical issues were resolved and a new revision of the application was deployed using updated Helm chart configurations. Additionally, SonarCloud was used as a tool for static analysis of code and to deploy bug-free quality code. In this study we also utilized the Open Container initiative (OCI) mechanism and stored the Helm charts securely in AWS S3. A fully automated Jenkins CI/CD pipeline was designed and developed to validate, analyze, generate reports and finally deploy the application. In this study, we aim to identify the critical gaps by previous contibutors and address those issues utilizing these tools. The key objectives include exploring these advanced tools, utilizing the efficiency of the OCI mechanism and automating the entire workflow to deliver a reliable and robust software delivery cycle. This research contributes to the evolving microservice and Kubernetes world, offering insights into best practices and tools. We conclude these study by evaluating metrics and analysis made by SonarCloud and KubeScore and thus opening doors to the future contributors.

Keywords: Kubernetes, EKS, SonarCloud, KubeScore, Jenkins, OCI, Helm, S3.

1 Introduction

In the rapidly evolving world of computing, the adoption of DevOps principles that facilitates cooperation between the development and operations teams has become an essential part. One of the important open source platforms that facilitated this transition is Kubernetes. Kubernetes is a highly effective container orchestration tool that helps in running and managing containerized applications(Smith and Doe; 2022). Kubernetes has become one of the most crucial tools for today's DevOps processes because of how well it serves dynamic and distributed environments, which are characteristic of cloud-native and microservice-based environments.

Kubernetes brings a completely new concept to application deployment by streamlining several tasks such as load balancing, resource allocation, and fault tolerance that typically require a lot of human intervention. Kubernetes makes use of declarative syntax, which allows for repeatable and robust deployment, while the self-healing capability ensures availability and reliability. However, with the advantages, Kubernetes also poses considerable challenges. A small misconfiguration can bring security, deployment failure, performance issues or can lead to an inefficient architecture. (Smith and Doe; 2022; Lee and Patel; 2023)

1.1 Background

Testing of Kubernetes environments is of significant importance in delivering a reliable as well as high-quality application. The distributed nature of the platform requires the use of advanced testing methods to ensure dynamic and efficient validation of both configuration and application logic. Testing is important for delivering high quality, scalable, and secure cloud-native applications. It is also important to avoid deployment failures in the production environment. To find the optimal configuration, to increase the security of an application and to implement clean, ethical static codes, it is important to include testing in your software delivery lifecycle (Johnson and Nguyen; 2024).

SonarCloud, a static code analysis tool, plays a significant role in ensuring code quality. Sonarcloud does this by identifying possible existing bugs in code, such as vulnerabilities and code smells etc. It also mandates that code being deployed follows a certain set of rules of coding, thus enhancing its maintainability and avoiding poor quality preventing it from being released to the production line, hence enabling efficient and secure product delivery. KubeScore, on the other hand, was implemented to deal with configuration related issues of Kubernetes environments and standardization of YAML files. It does this by identifying several risks including resource limits, identifying security related issues in configuration, improper setup of network related configuration etc. Hence, KubeScore strongly improves the reliability of Kubernetes configurations, as well as providing the configurations that enhance the security level. Hence, integrating Sonarcloud and Kubescore with the Jenkins CI/CD pipeline to automate the deployment alongside maintaining the code and configuration quality opens a door to implement secure code and configurations (Green and Carter; 2023).

1.2 Motivation

Looking at the importance of the adoption of DevOps practices and implementation of Kubernetes as an important tool for microservice based applications, there was a need to suggest a better and efficient architecture that implements testing of static code and configuration yaml alongside carrying out deployment using helm packages efficiently. In this research, Sonarcloud and Kubescore were integrated with Jenkins CI/CD pipeline to provide a robust solution. SonarCloud for enhancing code quality and implementing code standards, while KubeScore for implementing secure and optimized configurations. Another essential component of preserving scalable and repeatable deployments is effectively managing Helm charts and keeping them in an S3 store. This study offers a thorough approach to optimizing Kubernetes-based processes by tackling these issues. This paper presents a template of the CI/CD pipeline into which these tools and techniques have been placed to create a reusable reference to implement in organizations. By filling in these gaps, the research advances DevOps methods and ensures efficiency, security, and scalability in dynamic Kubernetes systems while meeting the demands of contemporary application deployment. [1]

[1]`https://spacelift.io/blog/kubernetes-devops`

1.3 Research Question

Testing is a significant part of DevOps, thus it is crucial to set up a baseline solution that fully incorporates testing of static code or Application code and testing of Helm configuration. The secure, bug-free static code combined with the best mode of configuration forms the basis of ethically and securely constructed applications needed by organizations.

Hence in this study we aim to address the following problems:

- Implementation of standard pipeline to automate testing as well as deployment of Application.
- To identify the tools and utilize them to resolve code smell, security Hotspots in application code and utilize tools to validate and analyze the configuration of Helm chart.
- Further, we aim to identifying a solution to store Helm charts efficiently.
- Lastly, to identify and implement best security practices and configuration to improve the reliability.

1.4 Objective

- The primary idea of this paper is to create a standard Jenkins CI/CD pipeline providing solutions to static code testing, enhancement of helm charts, secure storage of Helm charts, providing integration steps and security measures.
- Enhancing Code Quality: To utilize SonarCloud to run static code analysis of the Django application to identify bugs, vulnerabilities, and code smells. Sonarcloud provides a thorough report of bugs, vulnerabilities etc. It ensures that the code quality adheres to the standard quality gates set by the developers. Lastly, based on these suggestions, improvements were made to the application code to enhance code quality, thereby minimizing failure and improving reliability of the application.
- Optimizing Configuration Management: KubeScore ensures optimal and efficient resource utilization and adheres to Kubernetes standards and ethics. The objective was to utilize KubeScore and generate an analysis report identifying issues in Kubernetes configuration. Lastly, following the iterative approach to improve the Helm charts and deploy the new revision of application.
- Cost-Effective Resource Management: To provide the best and cost-effective method for storing helm charts. Based on this research, Amazon S3 was identified as an effective method. Helm charts were stored as OCI artifacts on Amazon S3. This approach eliminates the need for additional infrastructure while ensuring the availability and the ability to scale version-controlled deployment artifacts.
- Enhancing Deployment Efficiency: Helm charts maintain the reliability and consistency of an application in various stages of development. The objective was to utilize the Helm chart to deploy the application. Lastly, to implement an automated Jenkins pipeline that im[proves stability, reduces human errors, reduces the release time and makes the deployment process more efficient and accurate.

- Scalability and Adaptability: To utilize Amazon Linux EC2 instance to install Jenkins and other Jenkins related plugins and to develop a CI/CD pipeline that automates testing and deployment process to accommodate software delivery needs.

1.5 Research implication and Contribution

This research demonstrating the integration of Jenkins, Helm, SonarCloud and Kubescore contributes to a lot of scientific studies and is beneficial for organizations focusing on efficient DevOps solutions. Utilization of testing tools and automating deployment and testing reduces manual errors and ensures operation consistency across the environment. The practical implementation of this research can be done by those organizations that focus on adopting cloud native technologies. By providing detailed methods and implementation steps for automating the pipeline and integration steps, organizations can achieve higher quality of software deployment and efficiency in the software delivery life cycle. This research makes an important contribution to DevOps and its practices and provides its adaptability across different cloud platforms. With some tailoring of this standard pipeline, organizations can directly use this study to meet their specific needs. This research addresses a gap discussed in the Literature review section and plays an important role in contributing to the evolving DevOps domain.

1.6 Paper Structure

This research report is put into the following section to provide a detailed overview of the study. The report starts with an Abstract providing a brief summary of the research topic, highlighting the importance of research, method used and key approaches. Section 2 of the report focuses on the pre-work that was done as part of this study. It provides a detailed review of the papers that were referred for this research to highlight the key findings and identifies the gaps which are implemented in this research. In section 3 of the report, the introduction of the study is mentioned, emphasizing the importance of DevOps and Kubernetes in the software delivery life cycle. This section also highlights the challenges that come up with Kubernetes related to security and complexity of managing the cluster. This section also introduces the tools and technology used, alongside mentioning key objectives of research. Section 4 of the report describes the methodology used for developing the CI/CD template and to implement testing of Helm configuration and static code testing. This section provides details of the implementation of the study. Furthermore, section 5 of this report provides the Design perspective to reach the final goal of this study. It describes challenges encountered during implementation and also describes the functionality that was employed to encounter it. Additionally, section 6 discusses the key evaluation technique, improvements made to achieve code quality, and changes made to the Helm configuration based on the report by Kubescore. It also evaluates the cost, security, and code optimization with this study. Lastly, the last section concludes with the end note of the study providing the scope for future researchers and discussing the impact.

2 Related Work

Upon reviewing a lot of research articles, it is clear that Kubernetes excels as a first choice in container orchestration. But, each article mentioned its complexity related to managing configuration, implementing secure configurations, fault free deployment, and optimized memory usage. Many research studies discuss how default configurations of Kubernetes were inadequate even for smaller deployments. As discussed by Giangiulio and Malmberg (2022), issues and faults occurring due to misconfiguration leads to data breaches in Kubernetes-based deployments.Further, a study by Abhishek et al. (2022) Abhishek et al. (2022) shows that while Kubernetes has a solid architecture, there is a huge gap of frameworks for automating the containers management without violating the set security requirements. A study by the Cloud Native Computing Foundation (CNCF) clearly shows that a large majority, 78%, of companies that adopt Kubernetes do this by running their apps in multi-cloud and hybrid cloud environments due to its elasticity. [2] However, problems such as fragmented resources and complex cluster structure still remain. Closely, 42% of the users identified managing of microservice dependencies and securing interservice communication as critical and the need for innovative testing methods and a standard template for deployment.

2.1 Automation and Testing of CI/CD pipeline

A lot of studies discuss how automation using CI/CD pipelines plays a vital role in increasing the efficiency of the software delivery life cycle, producing high quality applications. As mentioned by Golis et al. (2023), the use of tools like SonarCloud is quite important to analyze and identify the vulnerabilities at an early stage of deployment. The author (Golis et al.; 2023) also mentions that early stage identification of vulnerabilities will ensure reliable and secure application deployment. Furthermore, the learning tool as discussed by Golis et al. (2023) lacks reduces the complexity of deployment but does not well-address the helm chart configuration validation technique, which is quite important for Kubernetes operations. The research by Zerouali et al. (2023) lacks a simplified integration of Helm with Jenkins simplifying the deployment process, but lacks in providing a method to validate configuration and static codes.Tools like KubeScore were not investigated by the author Zerouali et al. (2023). Hence, according to the above studies and the issue addressed, current studies either focus on static code analysis or easier deployment process or configuration validation, rarely addressing all of them and increasing the efficiency of applications. This gap justifies the need to integrate automation, testing and configuration validation to improve CI/CD pipeline suitably.

2.2 Enhancing Deployment: Role of Helm and Security Challenges

The ability of Helm to package resources, deploy rollbacks, and ease updates, due to which Helm charts, plays an important role in Kubernetes deployment. Gokhale et al. (2021) discuss an example of how Helm decreases the time for deployment by six times compared to the manual approach. Author (Gokhale et al.; 2021) also claims that no other tool could be as efficient as Helm, making it an indispensable tool. However, as discussed by the author, Zerouali et al. (2023) there are some issues like security flaws in configuration, duplications in configuration in publicly available Helm charts. Both the studies highlight the importance of using helm charts, but they lack providing a systematic method of security practices like vulnerability scanning, validation of configuration, secure storage

[2]https://www.cncf.io/wp-content/uploads/2020/09/Q4-2019-CNCF-State-of-Cloud-Native-Development-Ex
pdf

of helm charts using OCI standards is unexplored. Hence,in this study we propose a method that bridges this existing gap and enhances the quality and deployment of the application.

2.3 Storage and Management of Kubernetes Deployment Artifacts

Zerouali et al. (2023) in his study mentions securely managing and storing helm charts is important. Author (Zerouali et al.; 2023) also provides the idea of storing the Helm charts in Artifact Hub, but also says storing in Artifact hubs raises concerns about outdated and vulnerable images. Author, mention's 88.1% of the publicly available images contain vulnerabilities. Even if Artifact Hub provides a centralized access, storing Helm charts in Amazon S3 using OCI standards is more scalable, secure and cheaper (Golis et al.; 2023). However, these storage methods are still not fully implemented into automated pipelines.Hence, there was a need for a research addressing these limitations by integrating CI/CD with Helm chart stored in Amazon S3. Hence, by implementation of this approach, this study presents a secure management of Helm charts while ensuring accessibility, version control and compliance.

2.4 Challenges in Securing Kubernetes Workflow

Securing Kubernetes is an important and crucial task as vulnerabilities and misconfiguration in deployment artifacts can occur. The study presented by Giangiulio and Malmberg (2022), highlights the threats in multi-tenant Kubernetes clusters and also mentions risks related to data beaches. Similarly, Abhishek et al. (2022) proposed an automated Helm-based framework but didn't mention any security requirements in the CI/CD pipeline, leaving an important gap to address. Additionally, study by Spillner (2019) highlights the importance of Helm chart quality with HelmQA but didn't provide a clear integration method with CI/CD pipelines and lack security practices. The research also failed to consider security validation tools, including KubeScore and Sonarcloud. Hence, as discussed in the above research, all the above discussed studies lack secure configurations. Therefore, to address these challenges, the proposed study integrates static code analysis tools, configuration validation tools, and secure practices of Jenkins to address and provide secure workflow solutions to the challenges of the above studies.

2.5 Research gap and a Summary

The above discussed papers emphasize either on enhancement of deployment or improvement in Helm Charts or static analysis of code. However, none of the studies discussed above provided a standardized CI/CD template, secure configurations, method to analyze static code, storage of Helm chart in Amazon S3 into a unified study. Hence, this brings a need to approach these gaps, particularly related to security practices and integration of tools. Therefore, the proposed research integrates these missing tools, aiming to improve the reliability of Kubernetes-based deployment and increase in efficiency. This study proposes a standard Jenkins pipeline incorporating SonarCloud for static analysis of code, KubeScore for quality analysis of configuration, storage of Helm charts in Amazon S3 using OCI mechanism. This implementation not only bridges the gap but also improves the efficiency and quality of deployment by incorporating security measures and enhanced deployment strategies.

2.6 Overview of Reviewed Literature

Author	**Main focus**	**Technological solution**	**Limitations**
Golis et al. (2023)	Creation of framework for Kubernetes automation	Helm, CI/CD, Kubernetes	Less focus on security of configuration, didn't provide an idea or steps to integrate CI/CD with Kubescore
Spillner (2019)	Provided a method to test the quality of the Helm chart.	Helm charts	Lacks development of automated pipeline and integration of Helm with automation tools. Didn't address the challenge of storage of Helm artifacts.
Zerouali et al. (2022)	Analysis of publicly available Helm artifacts stored in Artifact Hub.	Artifact Hub, Helm charts	Very constrained discussion of automation, lacks focus on integration of OCI and use with Jenkins or other CI/CD tools.
Giangiulio & Malmberg (2022)	Analyzed security risks of Kubernetes based environment multi-tenant setups.	AKS, Kubernetes security	Focussed on benefits of namespace isolation but lack in providing an overview or discussion for configuration validation tool or improvement of configuration.
Abhishek et al. (2022)	Proposed a Method of using Helm and ansible-based container to carry out deployments.	CI/CD, Helm charts, Kubernetes	Lack in providing artifact storage solution and validation of security in CI/CD pipeline
Wadhams et al. (2024)	Explored the SonarCloud integration with Gitlab for analysis of static code	Sonarcloud, static code Analysis	Lacks the explanation of Kubernetes specific workflows.

Table 1: comparison of related work

3 Methodology

The objective of this study is to create and apply CI/CD pipeline that incorporates static code analysis, configuration testing, and automated deployment of kubernetes applications. The pipeline uses open-source tools like Jenkins, SonarCloud, Helm, KubeScore, and Amazon S3 repositories for storage of Helm charts and storage of reports of KubeScore.The detailed tool used, configuration setup, and high level overview of the implementation and workflow is discussed below

3.1 Research Procedure and Workflow Design

- Requirement Analysis: Defining the issues of current Kubernetes deployment process and related workflow, specifically identifying gaps of integration of static code analysis, configuration validation, and artifacts storage etc with Jenkins CI/CD.
- Pipeline Design: Designing a CI/CD pipeline to align the stages of developing, testing, validation as well as deploying applications. Pipeline is triggered when the new commit is identified, followed by Code Quality Check by sonar cloud, configuration assessment by KubeScore, thereby generating a report at Jenkins server and storage of that report in Amazon S3 and finally the deployment using helm charts located in Amazon S3 bucket.
- Tool Integration: This step involves integration of each tool within the Jenkins environment where different stages are combined making sure that data and interactions pass from one to another without interruption. This includes, defining webhooks, setting up of API,installation and configuring the plugins and scripting of pipeline stages using Jenkins pipeline-script.
- Iterative Development: Every element was tried and tested to check for results on the table to make improvements wherever necessary. Based on the results of KubeScore and SonarCloud, changes in configuration and code were made. Hence, a pipeline was made following the iterative approach of development, validating each stage and, at the end, making a constructive pipeline.
- Evaluation and Analysis: And finally, as a last stage of the development process, data at each stage was collected, like configuration scores of KubeScore, metrics or reports from SonarCloud and deployment success rate were documented and analyzed. This stage marks the future scope of this study and provides improvements that can be made.

The above procedure was followed and designed to align well with the standard software development lifecycle and adhere to research objectives. The above workflow provides a standard pipeline that integrates different tools for the solutions.

3.2 Technologies used, Tools and System configurations

The implementation of this study involved the use of different tools and technologies which were integrated, providing a template to simulate a secure and standard reliable pipeline for deployment scenarios.

- Jenkins: An open-source automation tool used to develop CI/CD pipelines. Jenkins was set up on an Amazon EC2 running Amazon Linux 2 OS. Different plugins along with default plugins like Git, SonarCloud etc. were installed.
- SonarCloud: A cloud-based solution for Code Quality and Security analysis for Static Code. It was integrated with Jenkins to analyze code as soon as commits were made on GitHub, providing reports on code smells, bugs, and vulnerabilities.
- Helm: An advanced package manager for Kubernetes which utilizes Helm charts to define, install and upgrade complex Kubernetes applications. For deployment, Helm was set up in Jenkins in order to carry out the deployment process..

- KubeScore: A tool for testing Kubernetes object definitions for best practices and potential issues. Kubescore was integrated with the Jenkins pipeline to check Helm chart configurations before running them.
- Amazon S3: It was utilized for storing Helm charts using the OCI standards. It offers a safe, scalable storage solution that is accessible during deployments.
- Kubernetes (EKS):Amazon Elastic Kubernetes service was used as a deployment engine. The EKS cluster was configured with 2 nodes and a production-like environment was created.

System Setup:

- Hardware: To host the Jenkins server, a t2.medium Amazon EC2 instance with 2 vCPUs and 4 GB of RAM was used. Additionally, a two node EKS cluster, of instance type t3.medium, was created to handle changing traffic at EKS.
- Software: Helm version 3.16.2, Docker 25.0.5, Jenkins-2.479.1-1.1.noarch and required plugins were installed on the Jenkins server. The Jenkins server was given network access to the EKS cluster and Amazon S3 buckets. There were two problems with Jenkins: it had network access to EKS clusters and Amazon S3 buckets.
- Security Configurations: IAM roles and policies were created so that Jenkins can communicate securely with EKS and S3. The Sonar Token of SonarCloud and GitHub personal access token were generated and stored in the credential manager at Jenkins. This token also ensures secure communication between open-source tools.

3.3 Techniques of Data Collection, Validation and analysis

Data was collected from various stages of the pipeline to assess the effectiveness of integrated tools and to determine the quality of the deployment process.

- Static Code Analysis Data: The metrics concerning code quality, such as code smells, bugs, vulnerabilities, and code coverage were reported in detail by SonarCloud based on the quality gate defined. The data was generated once the static code analysis stage in the pipeline was triggered.
- Configuration Validation Data: KubeScore generated a report containing crucial, significant, and other potential problems within the Kubernetes configurations, specified in Helm charts. The reports also contained some suggestions for enhancement.
- Deployment Metrics: Detailed information of rollout time, deployment success rate and other metrics were logged and monitored during deployment using Cloud Watch.
- Artifact Storage Logs: Since Helm charts artifacts were stored in Amazon S3, Amazon S3 access logs were enabled to track the correct retrieval and storage of Helm charts.

- Iterative Refinement: According to the gathered data, both Helm charts and application code were gradually improved. For instance, if the KubeScore identified a lack of resource limits, then the Helm chart was modified to meet the established requirements.
- Peer Review: Configurations file and code changes were discussed by a mentor to catch and improve the code and configuration not identified by automated tool.
- Compliance Checks: It ensured that configurations met ethical and governance policies and ensured best practices were implemented for Kubernetes deployment.

Analysis Method:

- Code Quality and Configuration Quality: It was measured by the number of reductions in code smells, bugs and vulnerabilities. Furthermore, configuration improvements were tracked by checking the number of priority, critical and severe issues resolved.
- Resource Utilization: Using cloud watch Continuously Monitoring was done to study the consumption of the CPU and memory of Kubernetes cluster and Jenkins server.
- Failure Analysis: When there were deployment failures or other issues at the moment of pipeline running, a detailed root cause analysis studying the Jenkins logs and build logs, were done to identify the issue and resolve it.

4 Design Specification

This section provides a brief overview of techniques, architecture of the study, framework used and pipeline structure that was used to implement the study.

4.1 Proposed architecture of the study

Figure 1 is the architecture diagram of a study that was applied to implement the CI/CD pipeline setup. All the necessary tools were installed and configured on the Jenkins server where they acted as a main automation tool. The Jenkins server had Helm, Kubectl, AWS-CLI installed. AWS-CLI facilitated communication with Kubernetes (specifically EKS) and Amazon S3. Furthermore, KubeScore was installed on the Jenkins server to validate the Kubernetes configuration from Helm charts. Using AWS CLI, the server was configured to communicate and authenticate with the Amazon EKS cluster to facilitate full-fledged deployments. The authentication method includes secure communication using AWS credentials and IAM roles for secure access using AWS CLI.

The Jenkins pipeline was created as a multi-stage CI/CD to automate deployment and validation. The pipeline consists of various stages. Once the pipeline is triggered, updated code from GitHub was fetched on the Jenkins server. The fetched repository consisted of application code and a Dockerfile. Further, in a second stage of the Jenkins pipeline, the build process was triggered, where a Docker image was created and securely stored in ECR (Amazon Elastic container repository). Furthermore, a dedicated stage in which SonarCloud was defined for the analysis of code. SonarCloud analyzed the

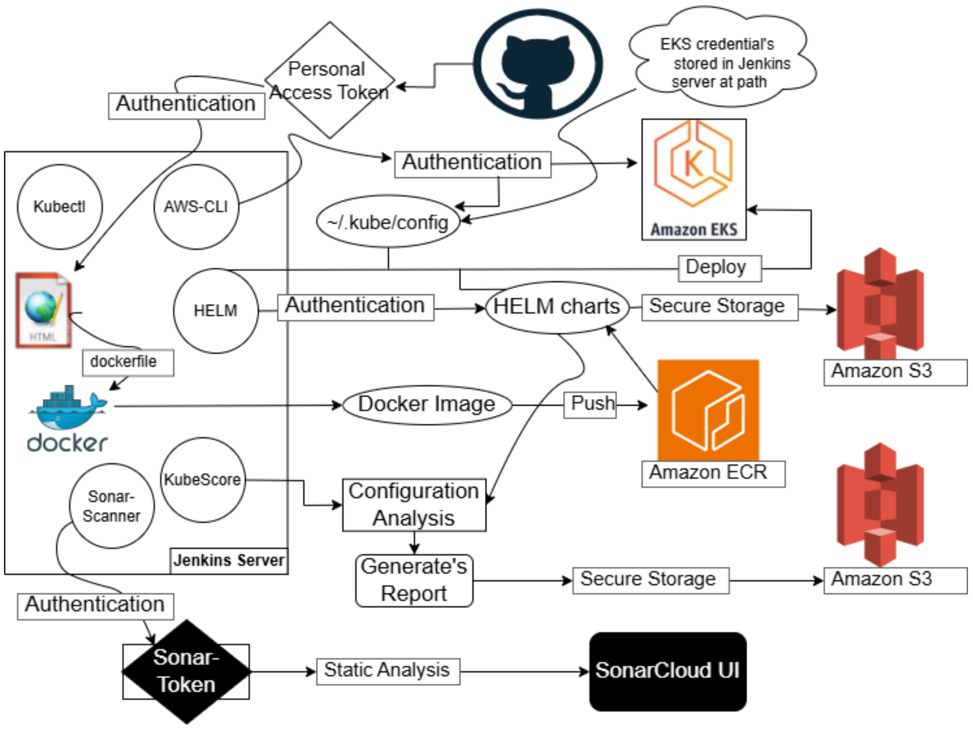


Figure 1: This is a Architecture Diagram

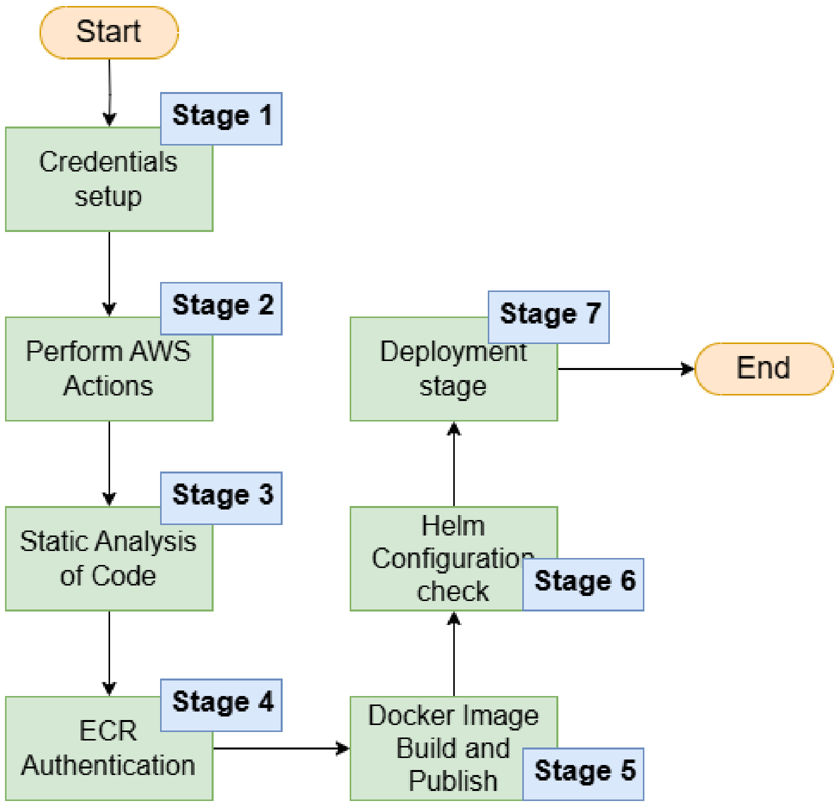


Figure 2: Flow Diagram of Jenkins Pipeline

bugs, vulnerabilities, and code smells in the application, immediately from the source code and provided the result in the form of insights. Following this, Helm charts were generated in the build stage. Utilizing Helm's OCI (Open Container Initiative) storage mechanism, these charts were securely stored in a repository in Amazon S3. This offered a safe solution which was horizontally scalable, version controlled, providing a suitable and efficient artifact management system for Helm charts.

Lastly, the KubeScore validation stage analyzed Helm charts configurations to check for Kubernetes best practices and configuration validation. The pipeline concluded with the deployment stage, where Helm was used to deploy the updated and validated application to the Amazon EKS cluster. This architecture discussed ensures a seamless, efficient and secure deployment process.

5 Implementation

This section describes the implementation strategies that were implemented to achieve the final goal. In software development, various tools and strategies are incorporated and integrated to achieve a final result. In our current study as discussed earlier, the core part of our CI/CD template is the Jenkins server configured on Amazon linux instance. The server, being the core component, had Jenkins, Docker, Helm, AWS CLI, Kubectl, Sonar-scanner installed on it and integrated with it. This section provides a detailed understanding of how each tool was integrated and utilized in this study. The detailed implementation and creation of Dockerfile, port exposed, Implementation of deployment configurations, Helm charts and output produced are emphasized in this section.

5.1 Jenkins Server Setup and Tools Integration

The Jenkins server acts as a central backbone of our CI/CD template, running on an Amazon Linux instance. It was specifically configured to integrate seamlessly with several critical development and deployment tools:

- GitHub Integration: GitHub webhooks were used by GitHub to trigger the pipeline automatically. For each new commit to GitHub, the Jenkins pipeline is triggered and each stage of pipeline was initiated sequentially. Jenkins had the Personal Access Token stored securely in Jenkins credentials manager, which facilitated the authentication, allowing for securely fetching the latest committed code into the Jenkins workspace on the Jenkins server.

- SonarCloud Integration: SonarCloud was used as a tool for static analysis of code. It was incorporated in the continuous integration stage of the Jenkins pipeline. Sonar-scanner was installed on the Jenkins pipeline, which utilizes the sonar-project.properties file to get the metadata, access configuration, project key, and inclusions specifying the target directory for analysis. As a part of the Jenkins pipeline stage, static analysis of code was done. SonarCloud had the quality gate defined, which contained the conditions and validation, based on which the code is supposed to be analyzed. These gates help to check for the quality of the code in various areas, such as security and other coding quality standards, before the deployment.

- Tool Installation: The Jenkins server setup also involved installation of other required plugins, including command-line tools like AWS-cli to manage the AWS services, kubectl to interact with EKS clusters, helm for creating and managing helm charts and sonar-scanner for execution of static code analysis.

5.2 Docker setup and Amazon ECR Integration

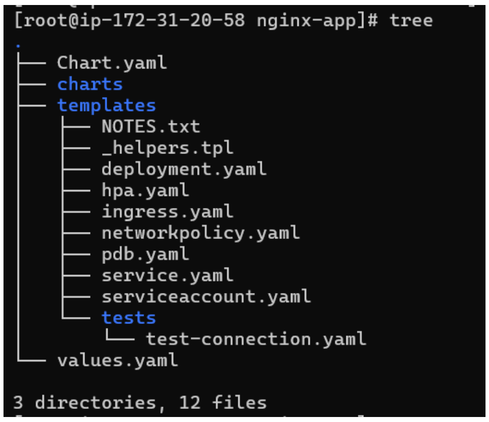
- Docker setup: The Docker engine was installed on the Jenkins server to initiate the creation of the docker image of our Django application deployed on EKS. The dockerfile was kept along with the application code. The Docker file used in the code, uses a lightweight, secure container image of python, and involves installation of required packages from a requirements.txt file, ensuring the complete code is copied to the container and the django server is run.

- Amazon ECR integration: Once the image is built, the docker image is tagged and pushed to ECR (Amazon Elastic container Registry). Furthermore, this stage of pipeline involves authentication with ECR using aws ecr get-login-password piped with docker login. This method handles the ECR credentials securely and avoids exposing them during the runtime of the pipeline.

5.3 Helm and Kubernetes Deployment Management

- Helm Chart Configuration: All the resources needed for deploying the Django application to Amazon EKS are defined by Helm charts that manage the deployment process. Helm setup in Jenkins involves creation of Helm chart, Helm chart update, managing Helm releases and configuration of resources like deployment.yaml, service.yaml and other configuration yamls. Helm was installed on the Jenkins server using an installer script that automatically gets the latest version of Helm and installs the helm after execution of the script.

- Helm Chart Implementation: To implement the helm charts, a helm chart was first created. Once the Helm chart is created, it provides us with a directory structure as shown in Figure 3.



```
[root@ip-172-31-20-58 nginx-app]# tree
.
├── Chart.yaml
├── charts
├── templates
│   ├── NOTES.txt
│   ├── _helpers.tpl
│   ├── deployment.yaml
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── networkpolicy.yaml
│   ├── pdb.yaml
│   ├── service.yaml
│   ├── serviceaccount.yaml
│   └── tests
│       └── test-connection.yaml
└── values.yaml

3 directories, 12 files
```

Figure 3: Helm Directory structure

The core components of helm charts are values.yaml, deployment.yaml and service.yaml. These are the files providing us a central understanding of how the application was deployed.

- Service Configuration: In service.yaml, the specification for the Kubernetes Service which routes traffic to the created pods is outlined. It is set to operate in a variable type defined by values.yaml, providing flexibility to change the traffic flow between ClusterIP, Loadbalancer and NodePort. In our study, NodePort was used to expose the application to the external world, mapping traffic from node port to pod and finally reaching the container.

- Deployment Configuration: The deployment.yaml is used to define strategies on which the application is deployed. It describes how pods are labeled and selected, specifies configuration related to security, and also handles resource limits to ensure efficient utilization of resources. Additionally, readiness probes were configured in deployment.yaml to check the readiness of the instance and application's health at the specified path and port. Lastly, configuration related to autoscaling, dynamically handling the number of replicas based on CPU usage were configured. Hence, ensuring easy handling of replicasets, handling utilization, checking the liveliness of applications and managing deployments.

- Values Configuration: The values.yaml file acts as the backbone of the Helm charts and contains essential properties such as the image repository, security contexts, and auto-scaling metrics. These configurations not only improve the flexibility and security of deployment but also guarantee that resources are optimized for improved cost and performance. Parameters mentioned in values.yaml are utilized by service.yaml and deployment.yaml without altering the main configuration file (deployment.yaml, service.yaml). The separation of configuration information from actual template ensures reusability, helps in maintaining clean code and reliable configurations. It provides a robust and efficient way to handle configuration or settings related to applications that contribute to providing how application is handled for scaling, security and how applications is deployed.

- Secret Management: To access the ECR credentials and pull Docker images, an Image Pull secret mechanism was implemented. Kubernetes uses thesis secrets to handle credentials safely in Kubernetes. Secret with the name "ecr-secret" was first created using Kubectl and then that secret was referenced as a parameter in values.yaml, When the deployment is created, Kubernetes will look at the imagePullSecrets, it will use the ecr-secret to authenticate with the ECR registry and deploy the pods. Managing these secrets efficiently is crucial to maintain the reliability of the environment and to utilize the correct private image of the application.

- OCI support:OCI support was enabled in the project to allow for managing Helm charts as container images. Enabling OCI support facilitated versioning and artifact sharing. By setting 'HELM_EXPERIMENTAL_OCI=1', we enabled a Helm's ability to communicate with OCI-compliant registries. Furthermore, enabling OCI support allowed Helm charts to be pushed and pulled securely from Amazon S3 configured as an OCI registry.index.yaml file was created and maintained in the Amazon S3 repository. It acted as a searchable database, and had metadata about each chart version, including its name, version, URL to Helm chart and a description. By updating the index.yaml file for each new version of Helm chart, this study ensures users have access to the latest updates of the Helm chart and an efficient configuration was implemented.

5.4 Security, Compliance and Continuous Improvement

- Security practices: GitHub token, AWS credentials and sonar token were stored securely in Jenkins credentials manager. Access to these credentials was controlled and restricted to the pipeline environment.

- KubeScore Integration: After the Kubernetes resources were defined and bundled as a helm chart, KubeScore was used to analyze the configuration for best practices and security related issues in the configuration. After the installation of KubeScore, a common manifest file containing all the configurations was created and the KubeScore score command was used to generate the analysis of the report. This analysis was further redirected to a text file and stored securely in AWS S3. And based on the iterative based improvement practice, the report generated by KubeScore was reviewed, and necessary configuration related improvements were made and a new revision of the Helm chart was released. Furthermore, to utilize the new revision and apply the efficient configuration, Jenkins pipeline was triggered and the latest version of the Helm chart was utilized, and the application was deployed on EKS.

- Pipeline creation and Flow: The Jenkins pipeline was created as a scripted pipeline utilizing groovy. Different stages of the pipeline include running sonar-scanner for static analysis, creation of Docker image, pushing the image to ECR, analysis of configuration using KubeScore and generation of reports and finally, deployment of the Django application on EKS (Elastic Kubernetes service).

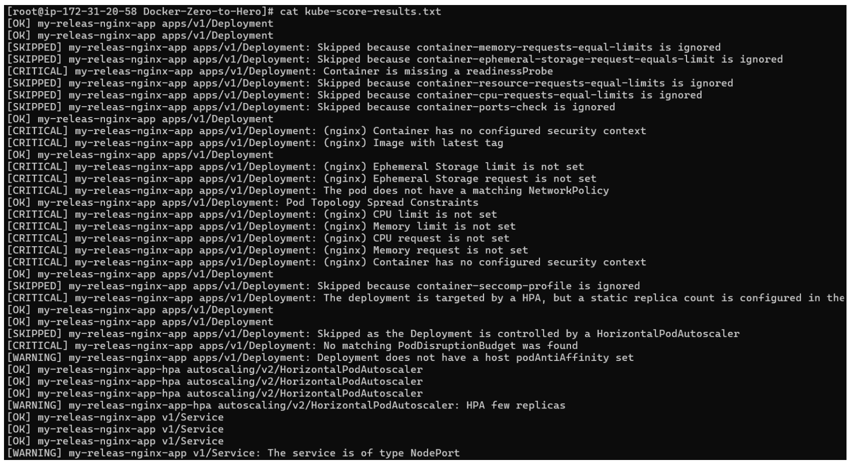
Hence, implementation of the above discussed CI/CD template utilizing Jenkins, Helm, Docker and EKS presents how modern open source tools can be integrated together to achieve an efficient solution. Through the strategic use of Jenkins for automation, utilization of Docker for image creation and containerization, using helm for configuration bundling and deployment on EKS, we achieved a significantly streamlined deployment process. Furthermore, the use of the OCI mechanism for managing Helm charts and storage of Helm charts in Amazon S3, enhanced our version control of Helm charts, thereby significantly improving deployment reliability and efficiency. The above implementation provided us with key evaluations and insights, which are discussed in the next section of the report.

6 Evaluation

This section aims at providing a critical analysis to identify the performance and security gain achieved through the integration of Helm, CloudWatch, KubeScore and SonarCloud with our CI/CD pipeline. Utilizing these tools, we were able to analyze and produce metrics concerning CPU utilization, network traffic, efficiency of resource allocation etc. SonarCloud was successful in providing insights into the code, detecting vulnerabilities, security concerns etc. Quality gates were configured to ensure the code meets standards for coverage, duplication, reliability, security etc. Hence, this evaluation section provides key metrics and outputs that increase the reliability, enhance the security of the system, maintain the high quality code of application being deployed and finally provide us with an efficient software delivery cycle. This section also provides further enhancement practices and techniques that can be utilized to further improve the environment.

6.1 Kubescore evaluation

The evaluation of the CI/CD pipeline's effectiveness significantly focuses on using KubeScore, an important tool to analyze Kubernetes configurations. When the KubeScore stage in the pipeline was triggered, it identified several issues related to security, missing configuration of readiness probes, threshold for maximum request handling and resource limits etc. These insights mentioned in the report were important to address and prompted revision of the Kubernetes deployment configuration within the Helm chart template.



```
[root@ip-172-31-20-58 Docker-Zero-to-Hero]# cat kube-score-results.txt
[OK] my-releas-nginx-app apps/v1/Deployment
[OK] my-releas-nginx-app apps/v1/Deployment
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped because container-memory-requests-equal-limits is ignored
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped because container-ephemeral-storage-request-equals-limit is ignored
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: Container is missing a readinessProbe
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped because container-resource-requests-equal-limits is ignored
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped because container-cpu-requests-equal-limits is ignored
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped because container-ports-check is ignored
[OK] my-releas-nginx-app apps/v1/Deployment
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) Container has no configured security context
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) Image with latest tag
[OK] my-releas-nginx-app apps/v1/Deployment
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) Ephemeral Storage limit is not set
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) Ephemeral Storage request is not set
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: The pod does not have a matching NetworkPolicy
[OK] my-releas-nginx-app apps/v1/Deployment: Pod Topology Spread Constraints
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) CPU limit is not set
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) Memory limit is not set
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) CPU request is not set
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) Memory request is not set
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) Container has no configured security context
[OK] my-releas-nginx-app apps/v1/Deployment
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped because container-seccomp-profile is ignored
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: The deployment is targeted by a HPA, but a static replica count is configured in the
[OK] my-releas-nginx-app apps/v1/Deployment
[OK] my-releas-nginx-app apps/v1/Deployment
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped as the Deployment is controlled by a HorizontalPodAutoscaler
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: No matching PodDisruptionBudget was found
[WARNING] my-releas-nginx-app apps/v1/Deployment: Deployment does not have a host podAntiAffinity set
[OK] my-releas-nginx-app-hpa autoscaling/v2/HorizontalPodAutoscaler
[OK] my-releas-nginx-app-hpa autoscaling/v2/HorizontalPodAutoscaler
[OK] my-releas-nginx-app-hpa autoscaling/v2/HorizontalPodAutoscaler
[WARNING] my-releas-nginx-app-hpa autoscaling/v2/HorizontalPodAutoscaler: HPA few replicas
[OK] my-releas-nginx-app v1/Service
[OK] my-releas-nginx-app v1/Service
[OK] my-releas-nginx-app v1/Service
[WARNING] my-releas-nginx-app v1/Service: The service is of type NodePort
```

Figure 4: First Analysis of KubeScore

The feedback mentioned in Figure-4 from KubeScore report led to improvement in Kubernetes manifests. And, several Helm chart files were added to address the specific issues. The files that were added to address the particular issues are hpa.yaml, ingress.yaml, networkpolicy.yaml, pdb.yaml, and serviceaccount.yaml, each playing an important role in optimizing the application.

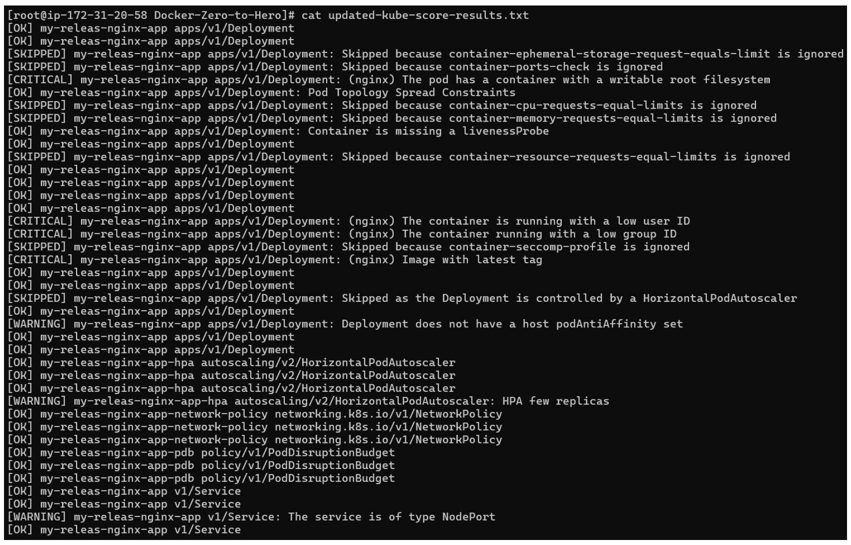
- Hpa.yaml (Horizontal Pod Autoscaler): This file plays an important role in handling the automation of scaling pod replicas based on CPU utilization and other metrics. This file ensures varying loads are handled efficiently, autoscaling during performance requirements or spikes are detected in CPU utilization etc. After inclusion of this file, CRITICAL issues related to CPU and memory were resolved in an updated report provided by KubeScore.

- Ingress.yaml: It helps in managing external access to the application services present in the Kubernetes cluster. It does this by defining rules of routing like HTTP(S) traffic etc. to the appropriate service. Furthermore, it simplifies exposing a single IP address and is crucial for SSL termination and load balancing. The issues like secure routing of traffic, SSL/TLS termination, centralized configuration management for routing were addressed by this file.

- PDB.yaml: This file was crucial to implement as this ensures that any kind of disruptions, upgrades or maintenance, the minimum number of replicas of the application should be running. It prevents your application from becoming unavailable and maintaining service level agreements (SLAs).

- Networkpolicy.yaml: By implementing network policy, Kubernetes setup was enhanced with security and traffic flow concerning pod-to-pod communication within the cluster.It also resolved issues related to uncontrolled access to data and flow by defining ingress and egress rules. This file ensures only authorized users and traffic has access, implementing strict regulatory standards.

Including the above files and enhancing the configuration as suggested by Kubescore was a strategic decision to enhance the security, scalability and deployment management. Each file was implemented considering the Kubernetes best practices and after the redeployment of the Helm chart and at the next iteration of analysis more than 70% of the CRITICAL issues recognized in the previous report were resolved, ensuring enhancement in DevOps practices and providing a strong foundation for deployment in EKS.The updated report can be seen in Figure 5.



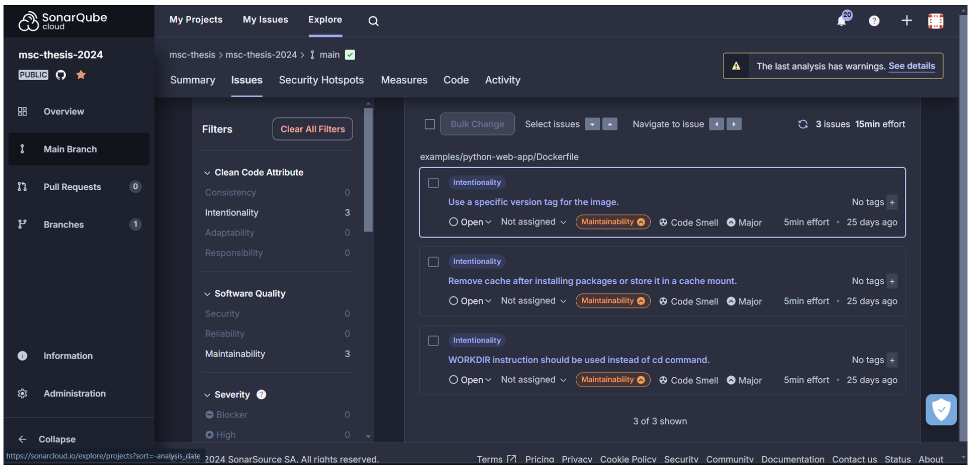
```
[root@ip-172-31-20-58 Docker-Zero-to-Hero]# cat updated-kube-score-results.txt
[OK] my-releas-nginx-app apps/v1/Deployment
[OK] my-releas-nginx-app apps/v1/Deployment
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped because container-ephemeral-storage-request-equals-limit is ignored
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped because container-ports-check is ignored
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) The pod has a container with a writable root filesystem
[OK] my-releas-nginx-app apps/v1/Deployment: Pod Topology Spread Constraints
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped because container-cpu-requests-equal-limits is ignored
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped because container-memory-requests-equal-limits is ignored
[OK] my-releas-nginx-app apps/v1/Deployment: Container is missing a livenessProbe
[OK] my-releas-nginx-app apps/v1/Deployment
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped because container-resource-requests-equal-limits is ignored
[OK] my-releas-nginx-app apps/v1/Deployment
[OK] my-releas-nginx-app apps/v1/Deployment
[OK] my-releas-nginx-app apps/v1/Deployment
[OK] my-releas-nginx-app apps/v1/Deployment
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) The container is running with a low user ID
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) The container running with a low group ID
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped because container-seccomp-profile is ignored
[CRITICAL] my-releas-nginx-app apps/v1/Deployment: (nginx) Image with latest tag
[OK] my-releas-nginx-app apps/v1/Deployment
[OK] my-releas-nginx-app apps/v1/Deployment
[SKIPPED] my-releas-nginx-app apps/v1/Deployment: Skipped as the Deployment is controlled by a HorizontalPodAutoscaler
[OK] my-releas-nginx-app apps/v1/Deployment
[WARNING] my-releas-nginx-app apps/v1/Deployment: Deployment does not have a host podAntiAffinity set
[OK] my-releas-nginx-app apps/v1/Deployment
[OK] my-releas-nginx-app apps/v1/Deployment
[OK] my-releas-nginx-app-hpa autoscaling/v2/HorizontalPodAutoscaler
[OK] my-releas-nginx-app-hpa autoscaling/v2/HorizontalPodAutoscaler
[OK] my-releas-nginx-app-hpa autoscaling/v2/HorizontalPodAutoscaler
[WARNING] my-releas-nginx-app-hpa autoscaling/v2/HorizontalPodAutoscaler: HPA few replicas
[OK] my-releas-nginx-app-network-policy networking.k8s.io/v1/NetworkPolicy
[OK] my-releas-nginx-app-network-policy networking.k8s.io/v1/NetworkPolicy
[OK] my-releas-nginx-app-network-policy networking.k8s.io/v1/NetworkPolicy
[OK] my-releas-nginx-app-pdb policy/v1/PodDisruptionBudget
[OK] my-releas-nginx-app-pdb policy/v1/PodDisruptionBudget
[OK] my-releas-nginx-app-pdb policy/v1/PodDisruptionBudget
[OK] my-releas-nginx-app v1/Service
[OK] my-releas-nginx-app v1/Service
[WARNING] my-releas-nginx-app v1/Service: The service is of type NodePort
[OK] my-releas-nginx-app v1/Service
```

Figure 5: Latest Analysis of KubeScore

6.2 Static code analysis

The static analysis segment of the study presents an important aspect of implementing effective DevOps practices in the CI/CD pipeline for analysis of vulnerabilities and resolving them. The primary goal was to implement an efficient pipeline that is capable of detecting vulnerabilities, security hotspots particularly focusing on those threats that pose significant threats such as insecure handling of secrets or credentials, Cross-Site Request Forgery (CSRF) and insecure configurations. This study presents a template to detect the above vulnerabilities and an example of how these vulnerabilities can be resolved with respect to quality gates and suggestions provided by SonarCloud.

This study evaluated and provided a resolution to specific high-priority vulnerabilities, which provided a significant improvement in the security of the application. For instance, the CSRF vulnerability was detected by SonarCloud, exposing the threats of unauthorized commands or unintentional data being transmitted to the web application without the user's knowledge. The CSRF threat detected in fig b was resolved by adding a decorator @require_http_methods(["GET", "POST"]), specifying only GET and POST requests



The last analysis has warnings. See details

Consistency	0
Intentionality	3
Adaptability	0
Responsibility	0
Security	0
Reliability	0
Maintainability	3
Blocker	0
High	0

3 issues 15min effort

examples/python-web-app/Dockerfile

Intentionality
Use a specific version tag for the image. No tags
Open · Not assigned · Maintainability · Code Smell · Major · 5min effort · 25 days ago

Intentionality
Remove cache after installing packages or store it in a cache mount. No tags
Open · Not assigned · Maintainability · Code Smell · Major · 5min effort · 25 days ago

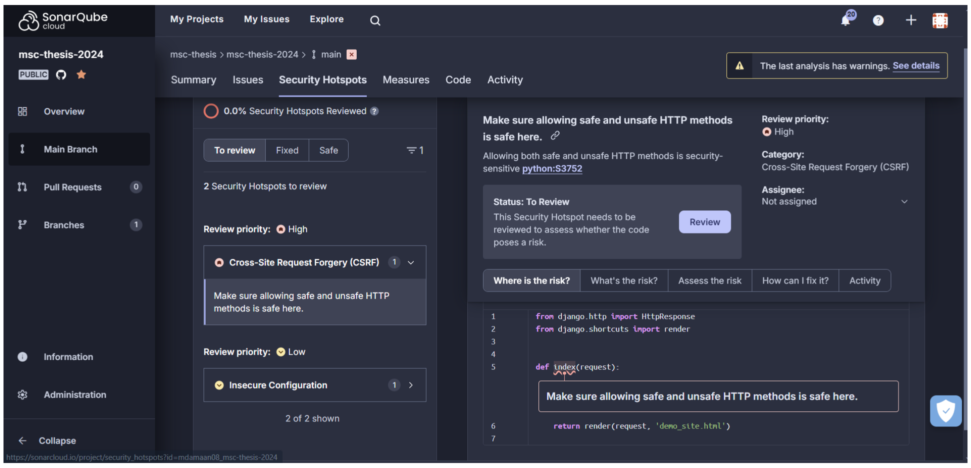
Intentionality
WORKDIR instruction should be used instead of cd command. No tags
Open · Not assigned · Maintainability · Code Smell · Major · 5min effort · 25 days ago

3 of 3 shown

https://sonarcloud.io/explore/projects?sort=-analysis_date

2024 SonarSource SA. All rights reserved.

Figure 6: Sonar Scanner Results



The last analysis has warnings. See details

0.0% Security Hotspots Reviewed

2 Security Hotspots to review

Review priority: High

Cross-Site Request Forgery (CSRF) 1

Make sure allowing safe and unsafe HTTP methods is safe here.

Review priority: Low

Insecure Configuration 1

2 of 2 shown

Make sure allowing safe and unsafe HTTP methods is safe here.

Allowing both safe and unsafe HTTP methods is security-sensitive python:S3752

Review priority: High

Category: Cross-Site Request Forgery (CSRF)

Assignee: Not assigned

Status: To Review

This Security Hotspot needs to be reviewed to assess whether the code poses a risk.

```
from django.http import HttpResponse
from django.shortcuts import render


def index(request):
```

Make sure allowing safe and unsafe HTTP methods is safe here.

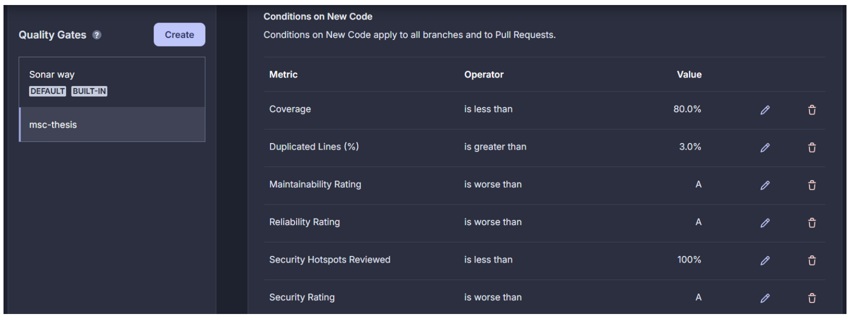
```
    return render(request, 'demo_site.html')

```

https://sonarcloud.io/project/security_hotspots?id=mdamaan08_msc-thesis-2024

Figure 7: Sonar Scanner Analysis Results

are allowed to index function. However, restricting HTTP methods doesn't completely mitigate the CSRF risks. Hence, one can also look for implementing a CSRF token into the form that checks this token during form submissions etc. By addressing such issues and with the iterative analysis of revised codes, a standard code was implemented ensuring it adheres to the quality gate defined in sonarcloud. The quality gates that were implemented are:



Conditions on New Code

Conditions on New Code apply to all branches and to Pull Requests.

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	A
Security Hotspots Reviewed	is less than	100%
Security Rating	is worse than	A

Figure 8: Quality Gates implemented

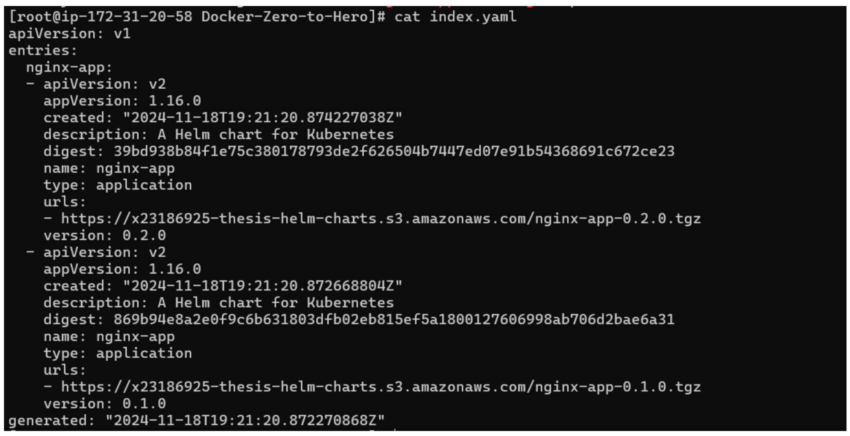
Hence, by focusing on high priority issues, research not only enhanced the security of the environment of the application but also demonstrated the effectiveness of implementing SonarCloud with your DevOps pipeline. Implementing SonarCloud and the above evaluation metrics addressed the research objective to create an efficient DevOps implementation that focuses on static analysis of code too.

6.3 Versioning of index.yaml file

Implementation of versioning in index.yaml file plays an important role in managing Helm charts within a Kubernetes environment. By creating versions, the system allows for control over deployment, ensuring specific versions can be rolled back. Additionally, the use of Open Container Initiative (OCI) enhances the security and creates a standard format for storing them and distributing them. The file index.yaml, when analyzed, had different versions created, confirming that the implementation was done accurately. The application was further deployed using different versions and for each deployment, changes corresponding to each version, were reflected in the application. This integration improved the overall reliability and security of the deployment practice and provided a modern cloud-native solution that can be implemented in DevOps practices. Figure 9, shows the different version created in index.yaml file.

6.4 Discussion

The above experiment discussed from the implemented CI/CD pipeline and integrating KubeScore, Helm, SonarCloud has provided us with the key evaluation and insights. The integration of these tools has enhanced the deployment efficiency, providing metrics like CPU utilization, resource allocation, network traffic etc. Use of SonarCloud played a crucial role in identifying vulnerabilities and handling code quality based on quality gates. Despite the improvements related to security and reliability, the evaluation after the iteration revealed issues like insecure configuration, version tags in Dockerfile remained



```
[root@ip-172-31-20-58 Docker-Zero-to-Hero]# cat index.yaml
apiVersion: v1
entries:
  nginx-app:
  - apiVersion: v2
    appVersion: 1.16.0
    created: "2024-11-18T19:21:20.874227038Z"
    description: A Helm chart for Kubernetes
    digest: 39bd938b84f1e75c380178793de2f626504b7447ed07e91b54368691c672ce23
    name: nginx-app
    type: application
    urls:
    - https://x23186925-thesis-helm-charts.s3.amazonaws.com/nginx-app-0.2.0.tgz
    version: 0.2.0
  - apiVersion: v2
    appVersion: 1.16.0
    created: "2024-11-18T19:21:20.872668804Z"
    description: A Helm chart for Kubernetes
    digest: 869b94e8a2e0f9c6b631803dfb02eb815ef5a1800127606998ab706d2bae6a31
    name: nginx-app
    type: application
    urls:
    - https://x23186925-thesis-helm-charts.s3.amazonaws.com/nginx-app-0.1.0.tgz
    version: 0.1.0
generated: "2024-11-18T19:21:20.872270868Z"
```

Figure 9: Versioning of Helm charts

unresolved. This suggests a need for more iterative focus and revision of code to make more bug-free and secure code.

The findings from KubeScore also provided us with suggestions related to Kubernetes configurations. Inclusion of files like hpa.yaml, networkpoicy.yaml, pdb.yaml, ingress.yaml resolved critical issues identified in the Kubernetes environment by deploying a secure, scalable and manageable application.

Furthermore, this study focuses on creation of efficient pipelines having provision to analyze static code, configurations, secure creation of image and deployment rather than solving all the static code issues identified. This study provides how security can be analyzed at an early stage of deployment and maintaining an interactive and continuous improvement solution.

Overall, this section discusses the limitations and potential improvement made by implementation of the above-discussed CI/CD pipeline. It contributes to ongoing research and implementation practices to achieve an efficient solution and a secure deployment process.

7 Conclusion and Future Work

The evaluation and methodologies discussed in this study provide us with a resolution of critical issues, particularly related to Cross-site Request Forgery (CSRF), as analyzed by SonarCloud. Almost more than 70% of CRITICAL issues identified by Kubescore were resolved in latest revision of Helm chart. Although, issues like version tag implementation in Dockerfile, use of WORKDIR, storage of cache in cache mount, these issues remained unsolved. Contributors can work on resolving these issues to deliver a more robust and secure environment. Furthermore, the use of COI mechanism for managing the Helm versions proved effective. However, tools like Spinnaker and Terraform can be used and explored to further enhance the maintenance and configuration of the deployment environment. Additionally, contributors can look for implementing a monitoring system that provides detailed visual insights and provide ease in troubleshooting.Implementation of monitoring will enhance the observability, provides robust metrics and significantly enhancing the system reliability.Further iterations can be made and evaluated based on different tool to achieve a fully automated, secure, and efficient solution.Such iteration would refine the deployment strategies and enhances the security posture of software delivery cycle.

References

Abhishek, M. K., Rao, D. R. and Subrahmanyam, K. (2022). Framework to deploy containers using kubernetes and ci/cd pipeline, *International Journal of Advanced Computer Science and Applications* **13**(4): 522–530.

Giangiulio, F. and Malmberg, S. (2022). Testing the security of a kubernetes cluster in a production environment. Degree project in technology, first cycle, 15 credits.

Gokhale, S., Gunjawate, S., Gupta, S., Poosarla, R., Hajare, A., Karve, K., Tikar, S. and Deshpande, S. (2021). Creating helm charts to ease deployment of enterprise application and its related services in kubernetes, *2021 International Conference on Computing, Communication and Green Engineering (CCGE)*, IEEE, pp. 1–7.

Golis, T., Dakić, P. and Vranić, V. (2023). Automatic deployment to kubernetes cluster by applying a new learning tool and learning processes, *SQAMIA 2023 - Tenth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications*, Bratislava, Slovakia, pp. 174–180.

Green, L. and Carter, S. (2023). Automated configuration management in kubernetes with kubescore: A case study, *Advances in Cloud Computing* **18**(2): 198–215.

Johnson, M. and Nguyen, H. (2024). Enhancing security and configuration management in devops: The role of static analysis and ci/cd automation, *Journal of DevOps Practices* **9**(3): 345–362.

Lee, C. and Patel, B. (2023). Cloud native approaches to software delivery: Tools and strategies for effective devops, *Advances in Software Engineering* **15**(2): 150–165.

Smith, J. and Doe, A. (2022). Enhancing devops through automated ci/cd pipelines and security tools integration, *Journal of Cloud Computing Advances, Challenges and Applications* **13**(1): 234–249.

Spillner, J. (2019). Quality assessment and improvement of helm charts for kubernetes-based cloud applications, *arXiv preprint arXiv:1901.00644* **abs/1901.00644**: 1–19. This research was partially funded by Innosuisse - Swiss Innovation Agency in project MOSAIC/19333.1.
URL: *https://arxiv.org/abs/1901.00644*

Zerouali, A., Opdebeeck, R. and Roover, C. D. (2023). Helm charts for kubernetes applications: Evolution, outdatedness and security risks, *Conference on Software Engineering and Applications*, Brussels, Belgium, pp. 1–9.