STREAMLINING SOFTWARE RELEASE PROCESS AND RESOURCE MANAGEMENT FOR MICROSERVICE BASED ARCHITECTURE ON MULTI-CLOUD

Herath H. M. I. P.

IT20125516

B.Sc. (Hons) Degree in Information Technology

Specializing in Software Engineering

Department of Software Engineering

Sri Lanka Institute of Information Technology

Sri Lanka

September 2023

STREAMLINING SOFTWARE RELEASE PROCESS AND RESOURCE MANAGEMENT FOR MICROSERVICE BASED ARCHITECTURE ON MULTI-CLOUD

Herath H. M. I. P.

IT20125516

Dissertation submitted in partial fulfillment of the requirements for the Bachelor of Science specializing in Software Engineering

Department of Software Engineering

Sri Lanka Institute of Information Technology

Sri Lanka

September 2023

Declaration

I declare that this is our own work and this proposal does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other university or Institute of higher learning and to the best of our knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to Sri Lanka Institute of Information Technology, the nonexclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).”

Signature:

IT20125516	**Herath H. M. I. P.**	

The above candidate is carrying out research for the undergraduate Dissertation under my supervision.

Signature of the Supervisor: ………………………………...

Date: ………………………………...

Abstract

The popularity of microservice-based designs is rising as a result of its scalability, adaptability, and reliability. On multi-cloud platforms, maintaining and deploying microservices may be challenging and time-consuming, especially when it comes to providing new software upgrades. The software release process consists of several phases such as development, testing, staging, and deployment. Each step necessitates a unique set of resources and tools, and each stage introduces the possibility of human error. As a result, optimizing and automating the software release process can assist companies in lowering the risk of errors, improving release quality, and speeding up the deployment process. Fully automating the software release process entails using tools and technologies to create, test, and deploy software updates automatically. CI/CD pipelines, which automate the complete release process from code changes to deployment, can help with this. By completely automating the release process, organizations can remove the need for manual intervention, lowering the risk of errors and increasing deployment speed.

Identifying and eliminating inefficiencies in the software delivery process is part of optimizing it. Organizations can cut costs, improve resource management, and boost productivity by optimizing the release process. Several variables must be considered in order to streamline the software release process for microservice-based architectures on multi- cloud platforms. Containerization, orchestration, automation, tracking, and security are examples. Organizations can improve the reliability and scalability of their microservices, reduce downtime, and ensure application security by adopting best practices for each of these factors. The microservice-based applications are large and must be controlled effectively with service mesh management tools such as Istio. When deploying an application on the cloud, Kubernetes is the most common method to use in microservice-based architecture. As previously stated, Docker and Kubernetes are widely used for microservice-based architecture, and this research focuses on the software release process using those tools and technologies.

The overall research focuses on helping organizations by optimizing and automating their software release process for microservice-based architectures on multi-cloud platforms. By doing so, organizations can improve their resource management, reduce costs, and increase efficiency while ensuring the reliability and security of their applications.

Keywords: *Microservice, CI/CD, Containerization, Orchestration, Automation, Tracking, Security, Downtime, Kubernetes, Docker, Istio, Service Mesh, Multi-cloud*

Acknowledgement

I want to express my gratitude to my supervisor Dr. Nuwan Kodagoda and co-supervisor Mr. Udara Samarathu for their support and encouragement in helping me complete my research successfully. I also want to express my gratitude to the CDAP instructors and staff for giving me the chance to conduct this study, as well as the Department of Software Engineering at the Sri Lanka Institute of Information Technology.

Table of Content

List of Figures

List of Tables

List of Abbreviations

CI	Continuous Integration
CD	Continuous Delivery
CDT	Continuous Deployment
VM	Virtual Machine
MSA	Microservice System Architecture
TLS	Transport Layer Security
RBAC	Role Based Access Management
SRP	Software Release Process
PaaS	Platform as a Service
SaaS	Software as a Service
CPU	Central Processing Unit
IaaS	Infrastructure as a Service
SSO	Single Sign On
SCM	Source Code Management

1. INTRODUCTION

The software release process is critical in software development because it governs how software updates are planned, developed, tested, and distributed. A well-defined and optimized software release process can greatly improve software development efficiency, quality, and speed. Software release methods have changed dramatically in recent years, and Agile methodologies have emerged as a popular strategy to software development. Cloud computing, CI/CD and CDT automation [1] are becoming popular with the Agile Methodology, as most organizations began to use it and it became a fast way of creating software for both monolithic and microservice based architectures. With the rise of Agile and Microservice-based architecture, it became necessary to launch and maintain applications more quickly than previously. The primary goal of developing DevOps is to increase the automation of the software delivery process [2].

The software needed to be monitored and maintained after the software application was released during the software release process. There may be several phases of releasing a software program for creation, testing, user approval testing, and ultimately for use by end users called production or live. These stages may be different for each software company and the products depends on the need of the development process.

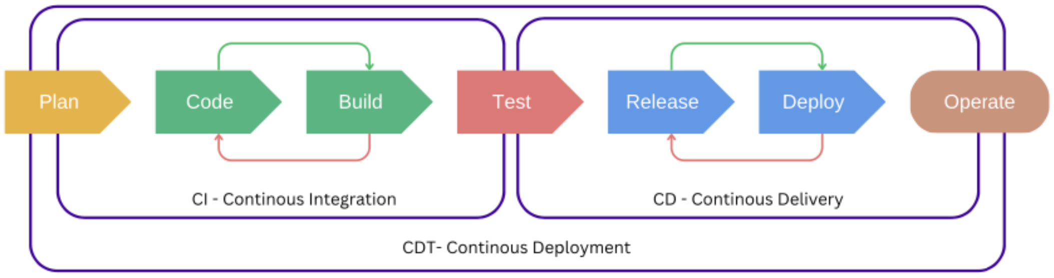


Figure 1: CI//CD/CDT Architecture

MSA [3] is based on set of micro-applications and communicate with each application as a single application to archive a business goal or goals. With the trend of using microservice based architecture, Docker is introduced and started to use that the micro-applications are deployed as docker container. Earlier, the monolithic architecture was popular in the software industry and used VMs as the deployment servers. VMs were huge barrier for the software release process with the growth of business needs and fast software releases by using microservice based architecture. Many of the lead companies such as Netflix, Coca-Cola, Spotify, Amazon, SoundCloud, eBay, Karma and Uber moved from monolithic to microservice based architecture due to the reason of scaling the application with their developed features. Docker [4] became the most popular containerization platform that can be used to deploy microservice based applications easily and reduce the resource usage.

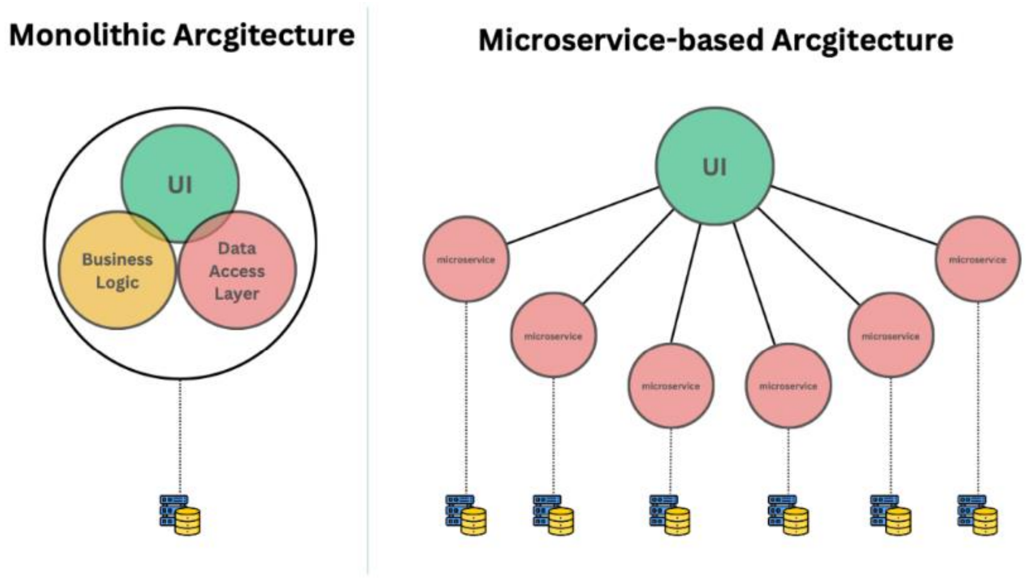


Figure 2: Monolithic vs Microservice Based Architecture

The application deployment was easy, scalable, and reliable with Docker, but with increasing number of containers, it was a hard and unmanageable to maintain the thousands of containers simultaneously. Kubernetes became the most popular and most usable platform to maintain containers as a container orchestration tool. Microservice based architecture was became a trend of software companies with the combination of Docker and Kubernetes platforms due to the easy and manageable of all the containers [5]. The micro-application communication mesh was the next challenge that needed to be resolved with giving a solution to manage and visible to manage easily. As a result, “Service Mesh Manipulation tools” were introduced such as Istio.

Kubernetes Service Mesh is a networking infrastructure component that manages and monitors interactions between Kubernetes cluster microservices. It includes several features that help with traffic control, security, and observability. Istio is one of the most common Kubernetes service networks. Istio is an open-source service mesh with a robust collection of tools for managing microservices [6]. It is based on Kubernetes and employs the Envoy proxy as a data plane to handle network communication between microservices. Istio includes functionality such as network routing, load sharing, circuit breaking, and fault injection. One of Istio's main benefits is its ability to modify the service mesh setup. It enables users to set network traffic management policies for their microservices, such as how much data is routed to each service and which services are permitted to interact with one another. Istio also includes a robust collection of security features such as mutual TLS, RBAC, and permission rules to help safeguard communication between microservices. Istio also includes a robust collection of observability tools. It offers measurements, tracing, and recording for network activity between microservices. Istio makes it simple to identify and fix problems in a Kubernetes cluster.

The security, testing and performance are most important aspects of the software release process. During the software release process, they needed to be measure and fixed if they needed any modification. The SRP is partially automated and using by software companies, but they are not fully automated and optimized. There are human interactions in between software development to software release and those stages may be roadblocks and frictions during the entire SRP. Some of the stages may be error prone due to the human interaction. Not only the automation is needed in the SRP, but it is also necessary to optimize the SRP according to architectural attributes such as performance, security, extendibility, reliability, and modifiability [7].

The optimized software release process is designed as streamlined architecture and developed with open-source solutions for all the aspects of the software release process as a fully automated system. The main reason of using open-source solutions is to reduce the cost for paid solutions. And the other reason of using open-source solution is, able to customize with using the source code and developing with necessary components for the defined streamlined SRP architecture.

This research proposal aims to address the need and feasibility of software companies by providing a solution to optimize the SRP based on architectural attributes and automate the software release process. The proposed system will be tested as a real-world solution to finalize the architectural solution, which will be fully automated and optimized. The ultimate goal is to streamline the software development and release process, improve the quality of software releases, and help software companies in Sri Lanka to stay competitive in the global market with having fast and reliable software release process.

1.1 Background and Literature

Before the "Agile" and "Microservice" based architecture is being popular in the software industry there were two main teams during the software development life cycle as "Developer Team" and the "Operation Team". The development team is responsible to release new features and hotfixes, but the operational team is responsible for software stability and reliability. With the different interests in both teams, the release process had many gaps and barriers during the quick release of a software version. Software versioning has become normalized after the microservice-based architecture and agile methodology comes to the industry, because of the rapid development of the software product. After the release of the initial product, there were hotfixes, minor updates, and major update releases on the software. With the rapid growth and release process, the old barriers were also a roadblock to releasing the updated software quickly to the end users. As a result, Development and Operation teams needed a solution and the solution was having DevOps architecture and DevOps is responsible to implement and manage the software development and software release process [8]. After the CI/CD and Containerization became the main components of the software release process, the wall of confusion between "development" and "operation" teams was destroyed.

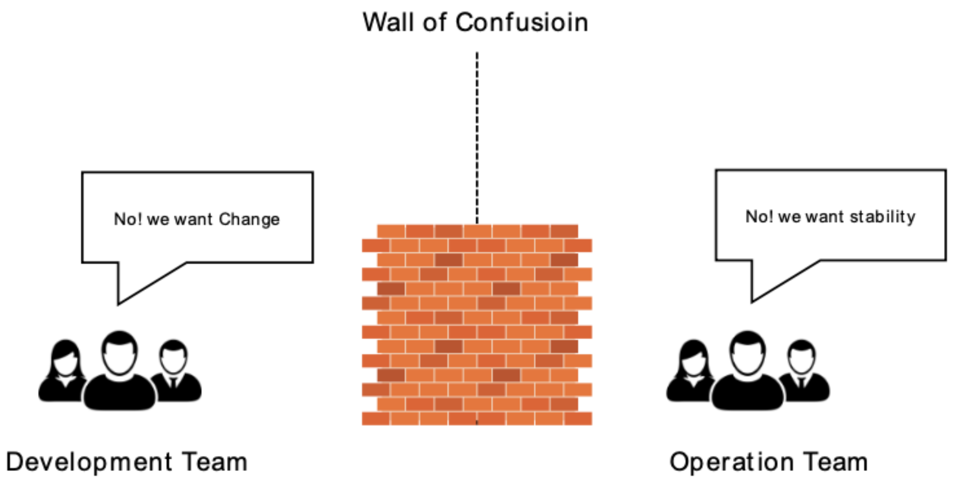


Figure 3: Wall of Confusion

The software release process is an essential part of software development, and optimization is critical for efficient, high-quality software releases. Agile methodologies, DevOps practices, containerization, container orchestration, security, testing, versioning, image manipulation, and service meshes are emerging trends in software development that are transforming the software release process. Istio is a popular open-source service mesh that provides a robust set of features for managing microservices in Kubernetes clusters. There are many numbers of solutions were

introduced for found problems and grow backs during the journey of DevOps into software release process. As the winner of the competition with different tools and technologies for containerization and container orchestrations, finally docker and Kubernetes survived their position with providing many numbers of features and solutions for raised problem during software release process and maintenance. Kubernetes architecture and Docker Architecture gives the clear and well-defined view for software release life cycle with providing the support to deploy and manage thousands of containerized applications [9]. During Docker manage the application images, Kubernetes manipulate the available containerized application.

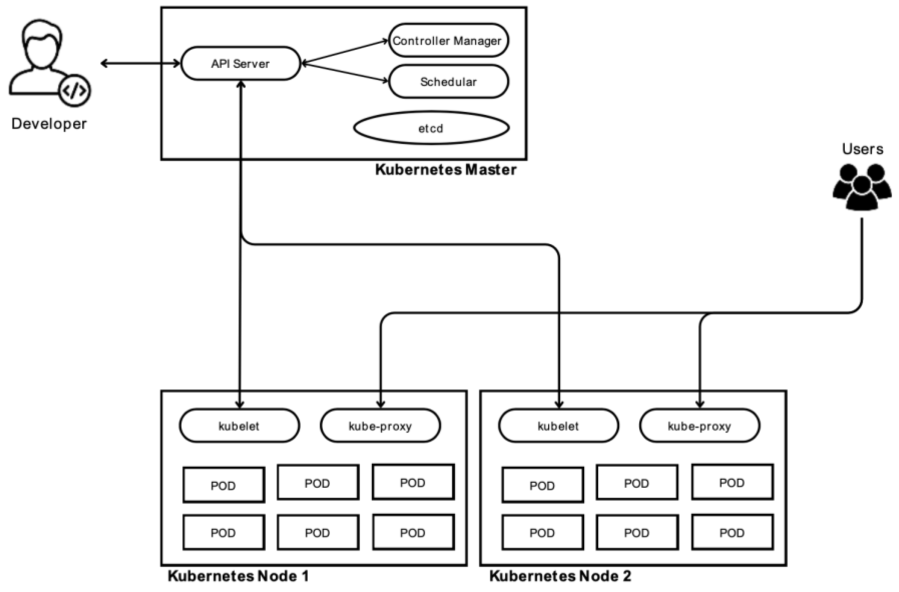


Figure 4: Kubernetes Architecture

Automated software release processes are invented to reduce the manual processes during software development life cycle. Currently, most of the software release process systems are automated but, they are not fully automated without the human interactions during approval processes and testing processes.

Because of the reason of not fully automated software release process systems, the need of fully automation become a common problem for the companies and businesses. As solutions CI/CD/CDT tools were invented. But there are only few of them are survived and became commonly used in the software industry. Not only those tools but also the tools and technologies for automated integration testing, automated security testing, unit testing and many other testing tools were invented and used in the industry.

Many reasons have influenced the need for automation and the implementation of DevOps techniques as software release processes have evolved gradually. One of the primary causes of this development has been the rising complexity of software systems, which has made manual release management increasingly challenging. As a result, a larger focus has been placed on automation and the use of tools and methods that can streamline the software release process.

Continuous Integration/Continuous Deployment (CI/CD) was a significant step forward in this development [10]. CI/CD is a software development methodology that stresses the significance of regular and automated code testing, integration, and deployment. This strategy has reduced the likelihood of mistakes and disputes during the software release process while also allowing teams to deliver software more swiftly and effectively.

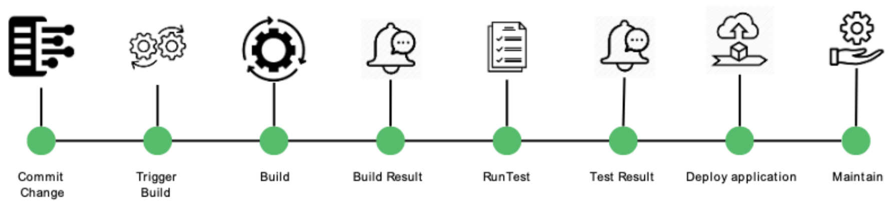


Figure 5: CI/CD Pipeline

The development of software release procedures has also been influenced by automation testing. Manual testing has become less efficient, time-consuming, and error prone as software systems have grown more complicated. Automation testing, on the other hand, allows teams to perform tests more swiftly, consistently, and frequently, lowering the risk of mistakes and conflicts during the software delivery process.

Another important element in the evolution of software release procedures has been the rise of DevOps techniques. DevOps is a concept that promotes cooperation, communication, and integration between development and operations teams. DevOps has helped to reduce the risk of errors and disputes during the software release process while also increasing the speed and efficiency of software delivery by breaking down silos between these teams and promoting a culture of cooperation.

Cloud computing has also changed the way software is released. Cloud platforms have allowed teams to scale their software systems rapidly and easily by giving on-demand access to a variety of computing resources, while also lowering the need for onpremises infrastructure [11]. Cloud platforms also offer a variety of tools and services that can assist teams in automating many parts of the software release process, from testing to distribution.

Even the software release processes are available with current automated tools, they have roadblocks due to some architectural attributes such as performance, security, extendibility, reliability, and modifiability [7]. And the current software release processes are not fully automated but only partially automated for different stages with the business requirement of different software companies. The current release process has mentioned architectural attributes that needed to be optimized because the systems are not easy to modify and easy to integrate with new tools and technologies to improve the quality of software development. Some systems take too much time to build and run tests, and as a result the software application get too much time to deploy in the necessary environments due to performance issues. The security is a main problem that needed to be covered during the software release life cycle and needed to be integrated and improved with streamlined software release process. Overall, the current software release processes needed to be well optimized under the mentioned attributes as a fully automated system.

1.3 Research Gap

The main research gap in this research to fulfill the gap in current software release processes with optimizing according to the architectural attributes. This research is focused on invent an architecture to automate and optimize to avoid the gap between current solutions available in the software industry for software release process. The security of the available software release processes is too low level because both external and internal developers are working in a same company and must share the resources with external teams. It became a challenge and security issue of software release process because the external teams get accesses to databases and storages. The performance issues are getting higher with the scaling up the number of users of the software release process systems. The system may perform slowly or get some down times due to the reason of unable to maintain with the increasing number of users. With the new technologies and tools, the current release processes can be optimized and have more benefits with extending with novelties for performance, security, and maintaining. But the current systems are tight with the company environment and have not extendibility. Same as the extending, the modifiability is also a one main attribute that needed to have in a software release process system to improve the reliability, accessibility, and maintainability.

According to the publication [12], the security of Kubernetes and Docker Swam can be archived with cloud storages. But as discussed previously, the security is unable to archive when working with external teams. The system can have backups for a disaster recovery; but it cannot be assumed as a security aspect. PaaS and SaaS services are available to secure data inside cloud storages from external vulnerabilities. Inside the company, it needed to be easy to have the higher security for the stored data from external teams who needed to access same data during the software release process. Not only during using by the external teams, also need to improve the security of deployed application and docker images during building and pushing to a container registry. As the publication [13] is mentioned, there are challenges during applying Security with DevOps as DevSecOps. Also need to consider the given solutions but, data security and containerization security has lack of resources and available solutions as open-source solutions.

According to the publication [14], the performance on Kubernetes deployment is measured and studies under VMs. But the necessary solutions are only on Kubernetes clusters with docker images. The performance may slow due to the lack of memory, lack of storage and lack of available CPU. The available solutions of publications are not meeting the necessary performance level to avoid the mentioned problems. The Docker image build time is also a waste of the time for the software release process with downloading necessary dependencies or packages for Maven or Node building technologies. Current available release process systems are downloading and pulling necessary data from internet all the time. Need to be optimized the performance with giving a solution for mentioned performance drawbacks.

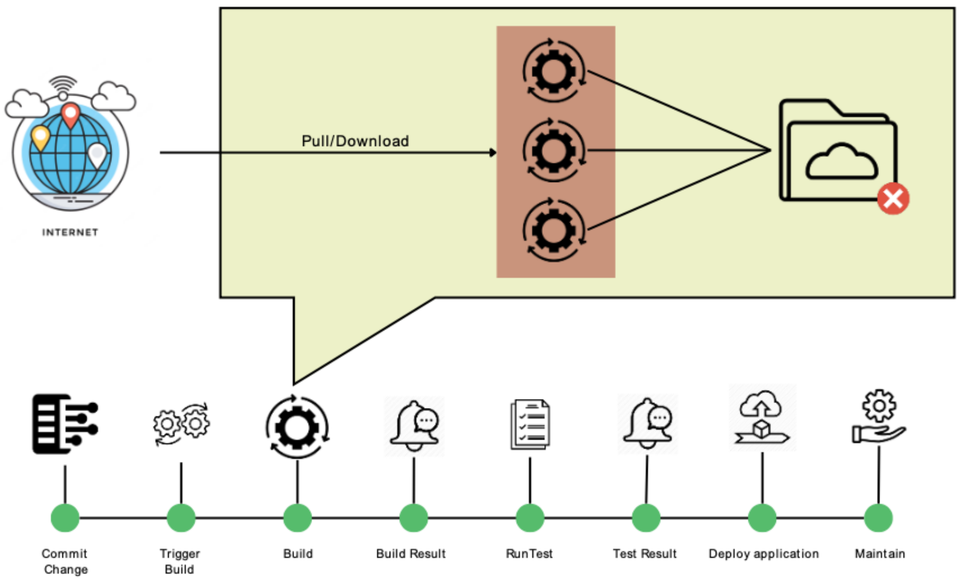


Figure 6: Download resources from the internet and finally clean the workspace.

Furthermore, as the publication [15], the extendibility is applied with automated configuration, initialization, and deployment with algebraic approach. But the proposed system is not optimized with Kubernetes and Docker based automated software release process system. The solution needed to be more effective and improved to available the extendibility and modifiability. According to the publication

[16], the unit testing is automated to avoid manual process of unit testing. But the solution is proposed with focusing Hydrologic Modeling. The current research is ongoing for microservice based architecture. Therefore, the solution needed to be improved to apply on microservice-based architecture. Moreover, the publication [17] I focused to automate the testing NodeJS and React projects. This needed to improve to available for any of the language or technology used in the software development. Once the testing is automated and could be applied to any programming technology, the extendibility can be optimized.

Software development reliability is needed to optimize from software building, testing, delivery, and deployment phases. As the publication [18], is focused to allocate resources during software testing phase. But the phases other than the testing is also needed to be optimized with the attribute of reliability. As the publication [19], the proposed solution is expensive and directly it effects to the cost of the software release process. The cost reducing is also a gap between available paid versions. Therefore, the proposed algorithm on cloud [19] is not the best solution when continue with open-source solutions.

There are some research and case studies are based on different areas of the architectural attributes that covered in this research projects. But they are not covered the architectural attributes cover in this research project during the optimization phase of the software release process. As mentioned in the publication [20], the attributes such as performance, security and reliability are considered during optimization, but they have not focused on extendibility, modifiability, and automation of the software release process. Also, in the publication [21], the research is focused to optimize performance and reliability, but they have not well optimized security, extendibility, modifiability, and automation.

As mentioned in the publication [22], the proposed automation is based on industrial architecture, and investigated with different microservice based architectures. But the solution is based on Eclipse and depends on the specific Eclipse based framework. If this solution is applied, the overall optimization cannot be archived due to the performance, extendible and modifiability architectural attributes cannot be archived.

2. RESEARCH PROBLEM

One of the most important problems in microservice-based designs is performance optimization, particularly in multi-cloud settings where service delay and network traffic can have a major effect on performance. Several factors contribute to performance optimization, including improving containerization and orchestration methods, making effective use of cloud resources, and finding performance obstacles through real-time tracking and analysis of system data.

When optimizing the software release process for microservice-based designs on multi-cloud platforms, security is another important factor to consider. Microservices are implemented in distributed settings, with each application posing a risk. To ensure security, possible security risks and weaknesses must be identified, safe communication routes between services must be established, and effective access control methods must be implemented. Security worries in microservice deployments are one of the most pressing study issues in this field. As more companies migrate to multi-cloud settings, guaranteeing the security of microservices becomes more difficult. This problem can be solved by conducting study on finding possible security risks and weaknesses in the software release process for microservice-based architectures on multi-cloud platforms. Solutions to minimize these risks and improve the security of microservices can then be created.

Another aspect that can affect the software delivery process for microservice-based designs on cloud platforms is extensibility. Extending microservices to satisfy shifting business needs frequently necessitates architectural changes, which can influence the release process. Adopting a modular architecture and applying design patterns such as the microkernel and plug-in patterns can aid in ensuring extendibility while reducing the effect on the release process. Microservice-based architectures must be reliable because they are frequently used to create mission-critical apps. Ensure reliability by finding possible sources of failure, adopting fault-tolerant mechanisms such as replication and redundancy, and having automatic recovery mechanisms.

Modifiability is another factor that must be considered when optimizing the software distribution process for microservice-based designs on Kubernetes platforms. Understanding the dependencies between services, finding the effect of changes on the system, and ensuring backward compatibility are all required when modifying microservices.

Collaboration and communication are essential for effective software delivery processes among coders, testers, and operations employees. Because of the distributed structure of the system, effective cooperation and communication can be especially difficult in multi-cloud microservices deployment.

With all the architectural characteristics we concentrated on here, the software release process must be completely automated, with no manual contact from software creation to software release.

3. OBJECTIVES

3.1 Main Objective

This research is aimed to invent an automated architecture and optimize the software release process system according to the architectural attributes such as performance, security, extendibility, reliability, modifiability.

3.2 Sub Objectives

The sub-objectives of the research are as follows.

- Identify the existing pain spots and inefficiencies in the software delivery process for cloud-based microservice designs. This could include performing questionnaires or conversations with release partners, evaluating existing release paperwork and data, or watching the current release process in action.
- Understand the architectural characteristics that are pertinent to the software release process, such as speed, security, extendibility, reliability, and modifiability. Conducting a literature study of best practices and research in

these areas, speaking with subject matter specialists, or evaluating case studies of effective microservice-based designs are all examples of this.

- Using suitable metrics and tools, assess the present software release process against the specified architectural characteristics. This will aid in identifying areas for development and may entail running trials or models to try various release strategies or setups.
- Create and execute an automated release process system that takes into consideration the architectural characteristics found. Selecting suitable tools and technologies, creating a pipeline or procedure that processes the various phases of the release process, and ensuring that the system is scalable, dependable, and secure are all examples of what this entails.
- Test and verify the automated release process system using suitable measurements and benchmarks and compare its efficiency to the prior manual release process. This will help to show the efficacy of the new system and may include user testing or other forms of proof.
- Over time, monitor and optimize the automated release process system, using input from stakeholders and continuing data to find areas for growth. This may entail updating the system to account for new technologies or shifting requirements, as well as continuous maintenance and assistance to ensure that the system continues to operate as intended.

4. METHODOLOGY

4.1 Requirement Gathering

Requirement gathering is an important part of the research process because it entails gathering and evaluating pertinent information to fulfill the research goals. Various methods are used to ensure a thorough grasp of the subject matter, including evaluating previous research performed in recent years and finding existing systems that offer alternative answers to the issues stated in previous parts. Furthermore, the research team collects information from several web sites, including official documentation from Jenkins, Kubernetes, OCI, Azure, Docker, and ArgoCD. These sites contain a wealth of information on the tools and technologies under consideration, such as their features, functionalities, constraints, and best practices for application.

A systematic strategy is used during the requirement gathering process to guarantee that all required information is gathered and analyzed. Identifying important stakeholders, interviewing subject matter specialists, running surveys, and reading pertinent literature are all part of this process. Using these methodologies, the research team can find gaps in current study and create research questions that can be answered through additional analysis.

4.1.1 Past Research Analysis

Many research papers have been published with the goal of automating the software release procedure. However, some study has been conducted to optimize software release procedures. Furthermore, no publishing has been made to provide the ability to create completely automated and optimize based on the architectural attributes concentrated on in this study.

The primary goal of past study analysis is to determine the tools and technologies required to construct the proposed system. Furthermore, it aids in identifying issues with previous publications as well as current difficulties.

4.1.2 Refer Official Documentation

Access to official documents is required to guarantee that we have current knowledge about the technologies that will be used in the development of the suggested system. While previous study papers can be useful tools, they may contain out-of-date material due to continuous technological updates. We have access to the most precise and up-to-date knowledge about the technologies we use because we use formal documentation from trustworthy sources. As a result, we will be able to create a more effective and efficient system that serves the requirements of our clients.

4.1.3 Identify Existing Systems

There are several current methods for automating the software release procedure, as described in publications [20, 21]. However, the suggested methods are not optimized, completely automated, or usable in real-world general software businesses. However, the systems are not completely optimized because they have many disadvantages due to fully automating and refining the system to provide the best performance, dependability, and availability.

4.2 Feasibility Study

4.2.1 Technical Feasibility

4.2.1.1. Knowledge of CI/CD

The knowledge of CI/CD is very helpful during architect the fully automated software release process. Because the main concept behind automation is grab with CI/CD pipelines. Understanding the usability and features are more important during inventing the fully automated software release process system.

4.2.1.2. Knowledge of Microservices

The system is focused to developed for microservice based architecture. Therefore, the core concept behind the inventing system is needed to be well-known before give the solution. The knowledge of microservices and architecture with related technologies will give the help to optimize and automate the software release process.

4.2.1.3. Knowledge of Jenkins

Once the CI/CD knowledge is achieved, the Jenkins is needed to know as the solution for CI. The Jenkins can be used to do the automaton building, delivery during the software release process.

4.2.1.4. Knowledge of ArgoCD

Knowledge of ArgoCD is the other main objective to have before moving forward with architecting the fully optimized and automated software release process. CD phase can be archived with ArgoCD in the CI/CD automations.

4.2.1.5. Knowledge of Docker

Before moving forward with microservice based architecture, it is necessary to have the knowledge of Docker. Docker images can be used to deploy the application as a set of micro-services. Docker enables us to maintain the necessary versions of images easily with container registries.

4.2.1.6. Knowledge of Containerization

As mentioned in the Docker section; the containerization is the core concept behind the Docker architecture. Having the knowledge of different between VMs and Containerization is mandatory before design the architecture for the release process.

4.2.1.7. Knowledge of Kubernetes

To manipulate and maintain the Docker containers, it is mandatory to have a container orchestrations tool. As a result, the Kubernetes became the best solution and the common platform for container orchestration. Therefore, the knowledge of Kubernetes is necessary to optimize and automate the software development life cycle.

4.2.1.8. Knowledge of Go Language

For the configurations and developments of the project components, it is a mandatory to have the knowledge of Go programming language. Because most of the development tools are developed using Go programming language that we are going to apply in the research project.

4.2.1.9. Knowledge of Agile Methodology and Project Management

Having knowledge of Agile methodology and project management for completing the research project is high. The Agile methodology is well-suited for iterative development, which allows for continuous improvement and adaptation to changing requirements. Project management skills are also essential for organizing and coordinating the various aspects of the project, including team members, timelines, and resources. Moreover, expertise in these areas can help identify and mitigate potential technical challenges and risks associated with the project, ensuring a successful outcome.

4.2.2 Schedule Feasibility

The suggested solution must be implemented within the time frame of the research. At the end of the study, the suggested automated and optimized software release process system can be combined with other members' solutions such as resource allocation, tracking, and multi-cloud deployment.

4.2.3 Economic Feasibility

The suggested system cost should be as reasonable as feasible to be preferred over current automatic systems and tools. Most companies make their systems and solutions for Kubernetes and Docker available as open source. All automation and efficiency solutions must also be linked with open-source tools and technologies.

4.3 System Analysis

4.3.1 Fully Automated and Optimized Software Release Process

The following is an overview of the suggested software solution. The following are the major components of the approach.

- Continuous Integration Server
- Continuous Delivery Server
- Optimization of Automated CI/CD

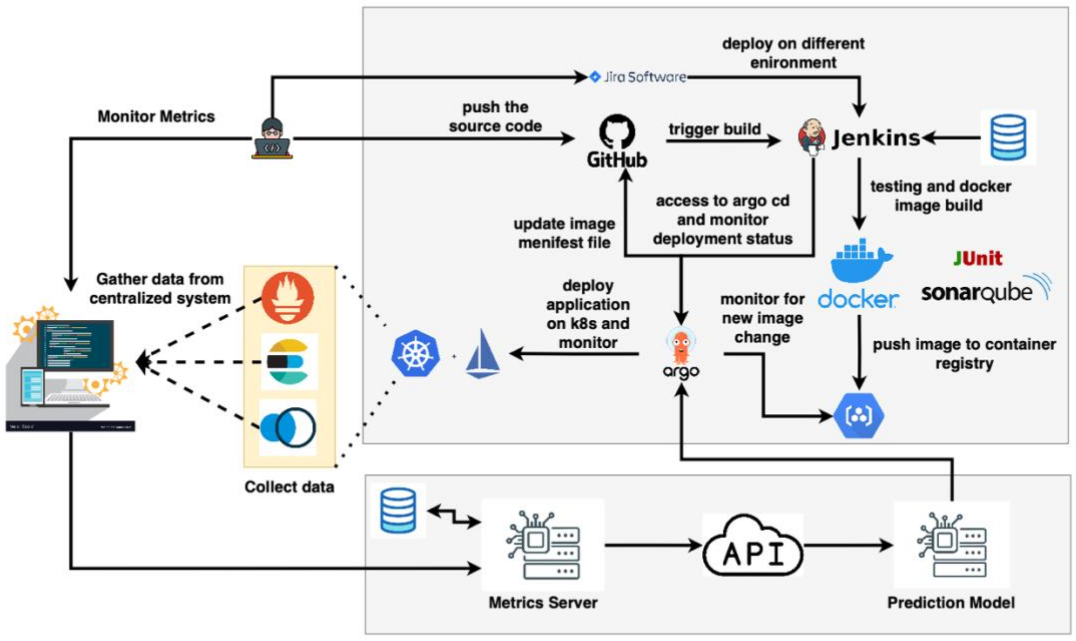


Figure 7: System Overview Diagram

4.4 System Development and Implementation

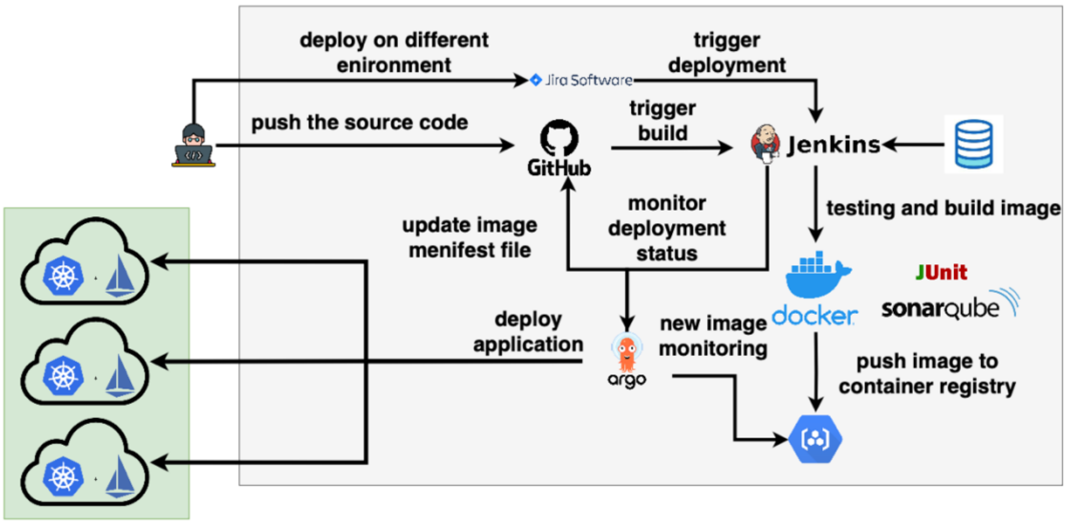


Figure 8: Automated Software Release Process Architecture Diagram

Followings are the main steps that needs to be covered during the development and implementation of the fully automated and optimized software release process.

1. Choose open-source tools and technologies that can help optimize the release process for the defined attributes, such as Jenkins for continuous integration, SonarQube for code quality analysis, and Docker for containerization.
2. Re-architect the software release process with the open-source solutions for fully automated software release process.
3. Optimize the architected software release process according to the architectural attributes focused on this research.
4. Define the stages and steps of the release pipeline, such as code integration, testing, deployment, and monitoring.
5. Implement a continuous integration process to automatically build, test, and integrate code changes, using tools such as Jenkins, Git, and Maven.
6. Implement a continuous delivery process to automate the deployment of code changes to staging and production environments, using tools such as Ansible, Chef, or Puppet.
7. Implement automated security testing to detect vulnerabilities in the code, using tools such as OWASP ZAP or SonarQube.
8. Implement automated performance testing to measure the performance of the system under different loads, using tools such as JMeter or Gatling.
9. Implement monitoring and alerting tools to detect and respond to issues in real-time, using tools such as Nagios or Prometheus.
10. Define and measure success metrics for the release process, such as the frequency and quality of releases, reduction in time to market, and improvement in customer satisfaction.
11. Train and educate stakeholders on the new release process and tools, to ensure smooth adoption and usage.
12. Continuously iterate and improve the release process and tools, based on feedback, results, and emerging best practices.

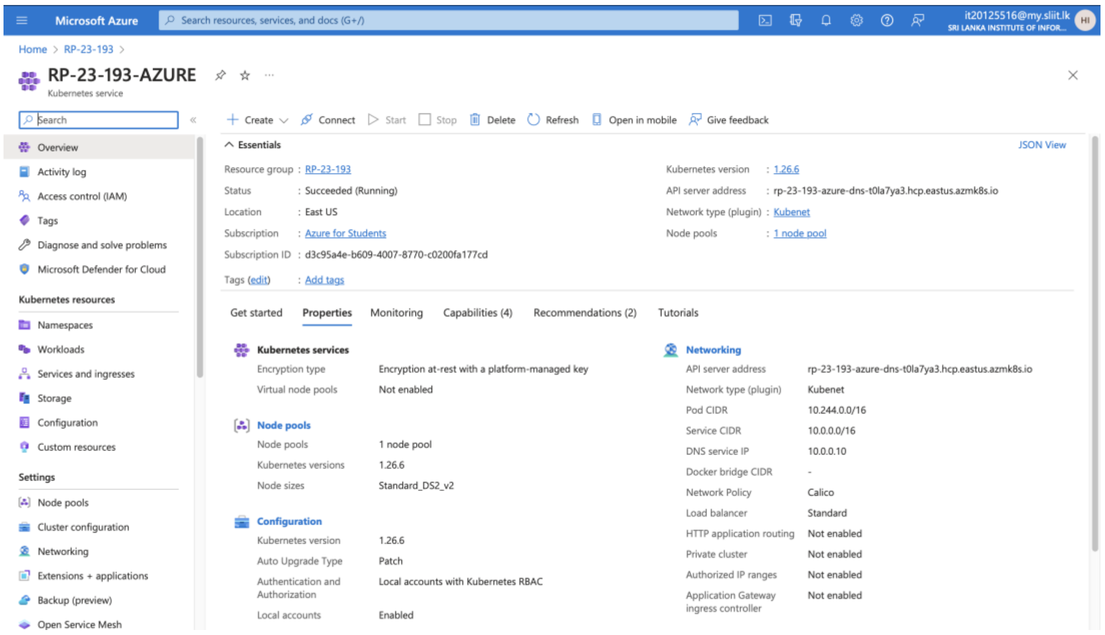
The tools and technologies are as mentioned as follows to development and implement the proposed architecture.

Programming Languages	● Go ● Java ● Groovy
Configuration Languages	● YAML ● Bash
Tools & Technologies	● Docker ● Kubernetes ● Git ● GitHub ● Azure ● Jenkins ● Argo CD ● Kustomize

Table 1: Languages, tools, and Technologies

4.4.1 Provisioning the Kubernetes Clusters and Install the Istio

The Kubernetes is the targeted deployment platform in this research. There are two separate Kubernetes Clusters are provisioned in Azure Kubernetes Service (AKS) and Google Kubernetes Engine (GKE). The Istio is used as the service mesh manipulation tool and install in both clusters after they are provisioned.



RP-23-193-AZURE
Kubernetes service

Essentials

Resource group	: RP-23-193
Status	: Succeeded (Running)
Location	: East US
Subscription	: Azure for Students
Subscription ID	: d3c95a4e-b609-4007-8770-c0200fa177cd
Tags (edit)	: Add tags
Kubernetes version	: 1.26.6
API server address	: rp-23-193-azure-dns-t0la7ya3.hcp.eastus.azmk8s.io
Network type (plugin)	: Kubenet
Node pools	: 1 node pool

Kubernetes services

Encryption type	Encryption at-rest with a platform-managed key
Virtual node pools	Not enabled

Node pools

Node pools	1 node pool
Kubernetes versions	1.26.6
Node sizes	Standard_DS2_v2

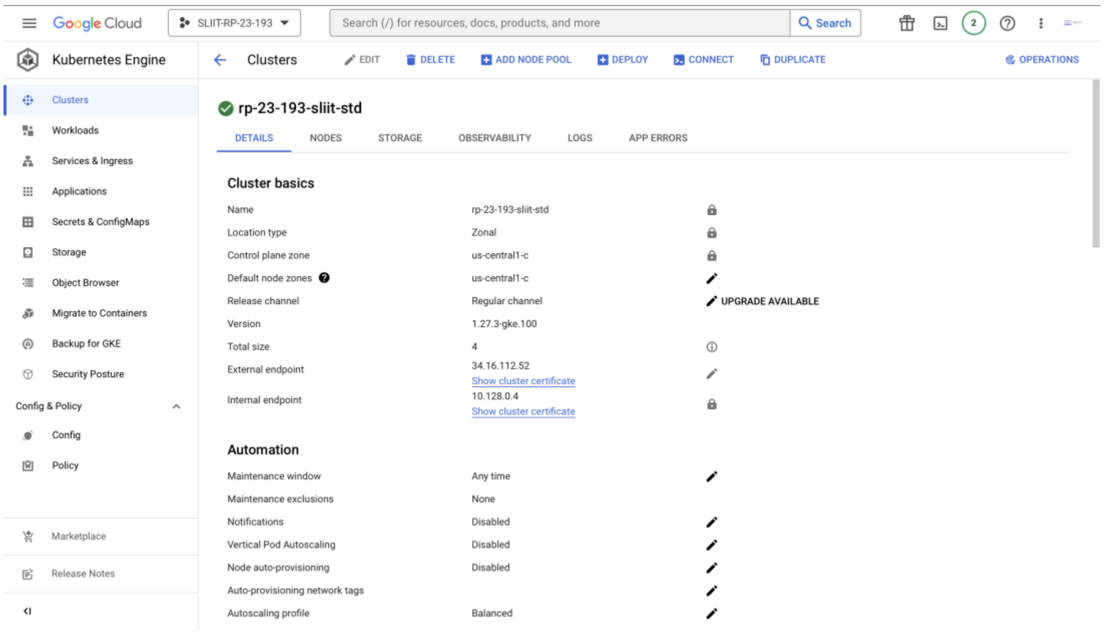
Configuration

Kubernetes version	1.26.6
Auto Upgrade Type	Patch
Authentication and Authorization	Local accounts with Kubernetes RBAC
Local accounts	Enabled

Networking

API server address	rp-23-193-azure-dns-t0la7ya3.hcp.eastus.azmk8s.io
Network type (plugin)	Kubenet
Pod CIDR	10.244.0.0/16
Service CIDR	10.0.0.0/16
DNS service IP	10.0.0.10
Docker bridge CIDR	-
Network Policy	Calico
Load balancer	Standard
HTTP application routing	Not enabled
Private cluster	Not enabled
Authorized IP ranges	Not enabled
Application Gateway ingress controller	Not enabled

Figure 9: Kubernetes Cluster in AKS



rp-23-193-sliit-std

Cluster basics

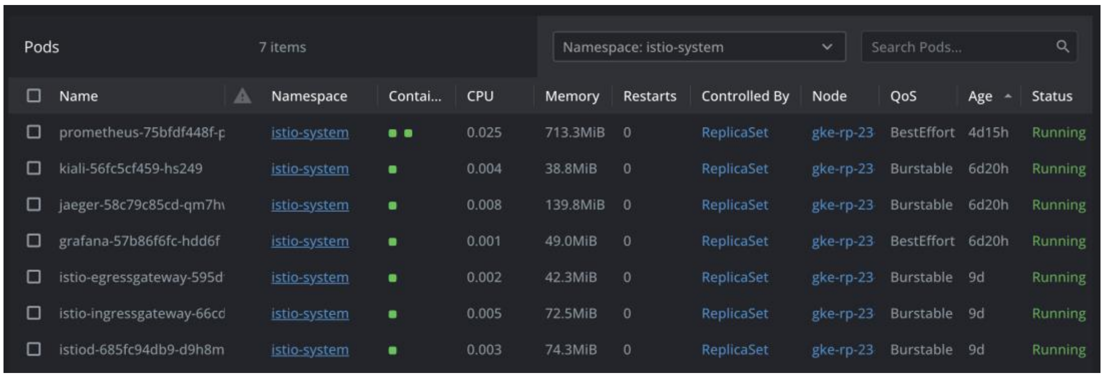
Name	rp-23-193-sliit-std	
Location type	Zonal	
Control plane zone	us-central1-c	
Default node zones	us-central1-c	
Release channel	Regular channel	UPGRADE AVAILABLE
Version	1.27.3-gke.100	
Total size	4	
External endpoint	34.16.112.52 Show cluster certificate	
Internal endpoint	10.128.0.4 Show cluster certificate	

Automation

Maintenance window	Any time
Maintenance exclusions	None
Notifications	Disabled
Vertical Pod Autoscaling	Disabled
Node auto-provisioning	Disabled
Auto-provisioning network tags	
Autoscaling profile	Balanced

Figure 10: Kubernetes Cluster in GKE

The Istio is deployed in the cluster including the monitoring tools that can be bind with Istio.



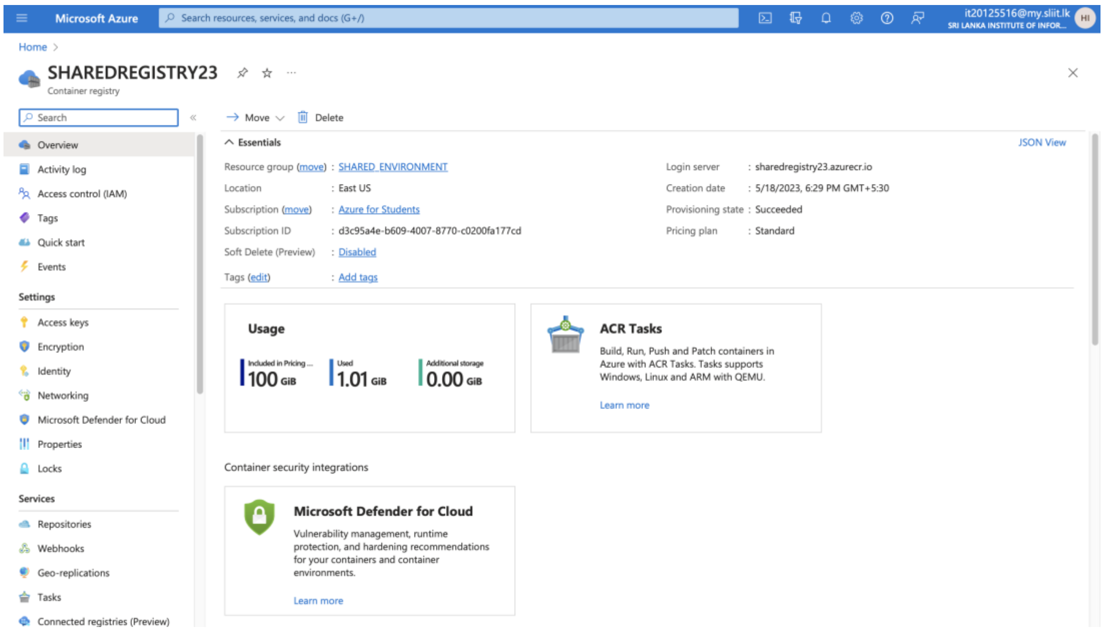
Name	Namespace	Contai...	CPU	Memory	Restarts	Controlled By	Node	QoS	Age	Status
prometheus-75bfdf448f-p	istio-system	🟩 🟩	0.025	713.3MiB	0	ReplicaSet	gke-rp-23	BestEffort	4d15h	Running
kiali-56fc5cf459-hs249	istio-system	🟩	0.004	38.8MiB	0	ReplicaSet	gke-rp-23	Burstable	6d20h	Running
jaeger-58c79c85cd-qm7hv	istio-system	🟩	0.008	139.8MiB	0	ReplicaSet	gke-rp-23	Burstable	6d20h	Running
grafana-57b86f6fc-hdd6f	istio-system	🟩	0.001	49.0MiB	0	ReplicaSet	gke-rp-23	BestEffort	6d20h	Running
istio-egressgateway-595d	istio-system	🟩	0.002	42.3MiB	0	ReplicaSet	gke-rp-23	Burstable	9d	Running
istio-ingressgateway-66cd	istio-system	🟩	0.005	72.5MiB	0	ReplicaSet	gke-rp-23	Burstable	9d	Running
istiod-685fc94db9-d9h8m	istio-system	🟩	0.003	74.3MiB	0	ReplicaSet	gke-rp-23	Burstable	9d	Running

Figure 11: Istio Installed in the Cluster

4.4.2 Provisioning the Private Container Registry

The Azure Container Registry is used as the shared container registry platform in the research implementation. The cluster is provisioned as a private cluster and only can be accessed with the authentication keys.

The Kubernetes cluster is contained with the keys as secrets for each namespace that necessary to pull the pushed container images including in the Jenkins Server.



SHAREDREGISTRY23
Container registry

Resource group (move)	SHARED_ENVIRONMENT
Location	East US
Subscription (move)	Azure for Students
Subscription ID	d3c95a4e-b609-4007-8770-c0200fa177cd
Soft Delete (Preview)	Disabled
Tags (edit)	Add tags
Login server	sharedregistry23.azurecr.io
Creation date	5/18/2023, 6:29 PM GMT+5:30
Provisioning state	Succeeded
Pricing plan	Standard

Usage

Included in Pricing ...	Used	Additional storage
100 GiB	1.01 GiB	0.00 GiB

ACR Tasks

Build, Run, Push and Patch containers in Azure with ACR Tasks. Tasks supports Windows, Linux and ARM with QEMU.

Container security integrations

Microsoft Defender for Cloud

Vulnerability management, runtime protection, and hardening recommendations for your containers and container environments.

Figure 12: Shared Container Registry in ACR

4.4.3 Develop the Fully Automated and Optimized Continuous Integration Process

The responsible of this component is automatically triggering the docker image building pipeline of Jenkins server, docker image build and push the docker image to the container registry. The Azure Container Registry (ACR) is used for the implementation as the shared container registry for the all the deployment environment. Once the source code is updated and pushed to the GitHub, the GitHub webhook will trigger the related Jenkins pipeline and the pipeline starts to build the Docker image for the source code. As an optimization option, the Jenkins is using Kubernetes pods as slave nodes for every pipeline when they started. There is a pod is provisioned as a slave node of Jenkins related to the started pipeline and the slave node is separated from the Jenkins server. The slave pod will be automatically terminated once the Jenkins pipeline ends the job from the server side with all the junk files related to the pipeline job.

The Jenkins server is using a separate persistence volume as the main storage and the storage is basically used to store the data related to the Jenkins server. There are separate persistence volumes are used for all the build pipelines. There can be any number of volumes based on the programming languages the pipelines try to build such as Node JS, Java, Python, C#. The purpose of using separate volumes separately for different pipelines to save the old packages and necessary data related to each programming languages. This solution is used as an optimization solution to reduce the downloads from the internet for every builds. If this solution is not available, every build will download the necessary package resource and dependencies from the internet all the time. But in this solution, the downloaded packages and dependencies are stored in the related persistence volume and first check whether the package or the dependency available in the volume. If the necessary resource is available in the volume, the pipeline job uses the resource already available in the volume, or if the volume does not contain the resource, job will download it form the internet and store the newly downloaded package or dependance in the related persistence volume based on the programming language.

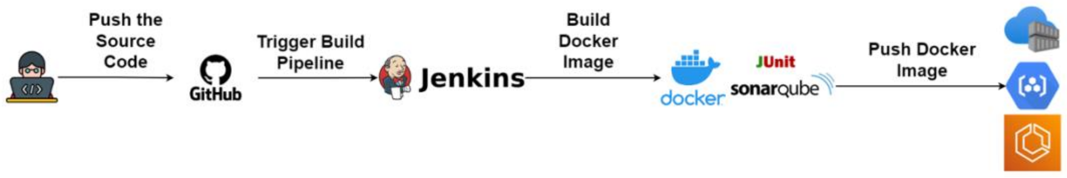


Figure 13: CI Architecture

In the docker image build stage, the pipeline job uses Kaniko as the docker build solution as an optimized solution. The Kaniko can be used to build docker images using the Dockerfile without the root permission. The docker daemon needs the root level permission to build the docker image. Also, it is allocated more resources than used by Kaniko and not needed to write separate lines to build the docker image and push the built docker image to the container registry. There is only one line to build and push the image to the container registry and we can build the docker image as a test without pushing it to the container registry. The docker image is pushed to the ACR once the build is successfully completed.

The CI Architecture is optimized to archive the architectural attributes that focused on this research such as performance, security, extendibility, reliability, and modifiability.

GitHub SSO is used for the security of the Jenkins Server to Authenticate and Authorized based on RBAC. And shared volumes are used to reduce the unnecessary downloads.

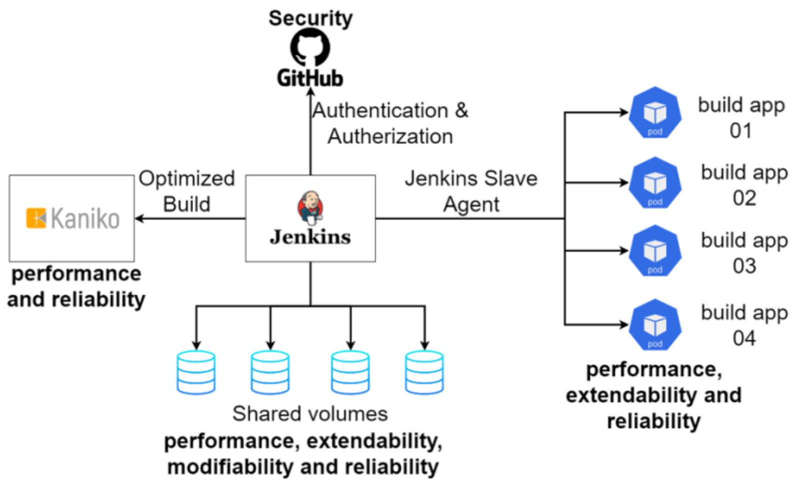
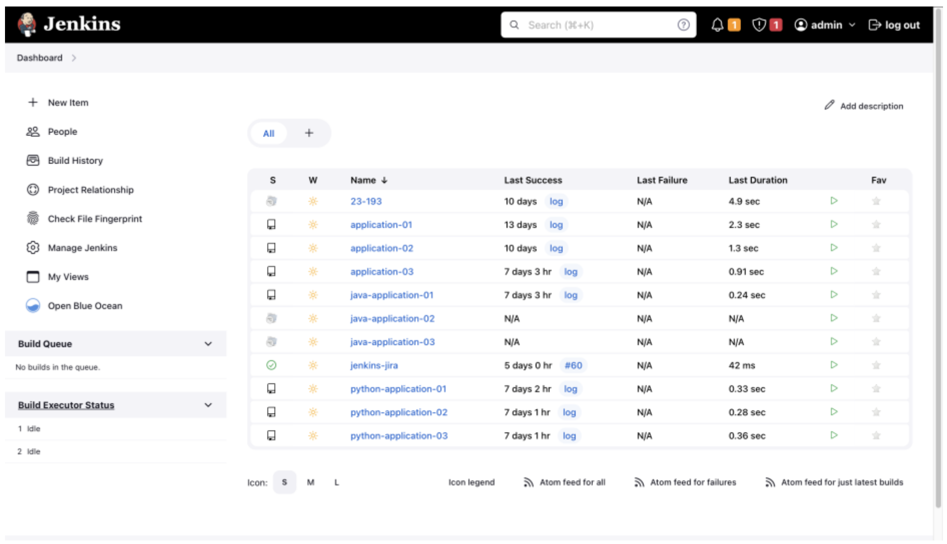

Security
GitHub
Authentication & Autherization
Optimized Build
Kaniko
performance and reliability
Jenkins
Jenkins Slave Agent
build app 01
build app 02
build app 03
build app 04
pod
performance, extendability and reliability
Shared volumes
performance, extendability, modifiability and reliability

Figure 14: Optimized CI Architecture

4.4.3.1. Configure the Jenkins Server

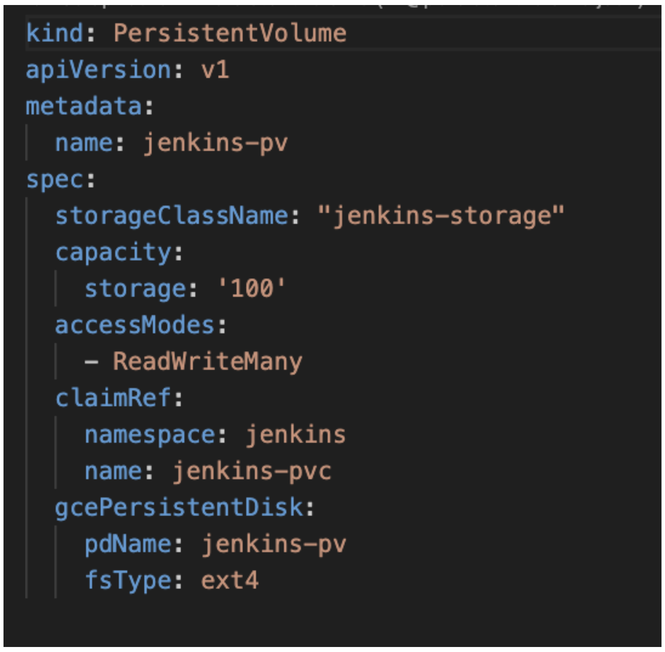
The main purpose of the Jenkins server is to automate the CI process of the software release process life cycle. The Jenkins server is deployed in the Kubernetes server. This is using as a shared server for all deployments including for the multi-cloud deployments.



S	W	Name ↓	Last Success	Last Failure	Last Duration		Fav
		23-193	10 days log	N/A	4.9 sec	▷	☆
		application-01	13 days log	N/A	2.3 sec	▷	☆
		application-02	10 days log	N/A	1.3 sec	▷	☆
		application-03	7 days 3 hr log	N/A	0.91 sec	▷	☆
		java-application-01	7 days 3 hr log	N/A	0.24 sec	▷	☆
		java-application-02	N/A	N/A	N/A	▷	☆
		java-application-03	N/A	N/A	N/A	▷	☆
✓		jenkins-jira	5 days 0 hr #60	N/A	42 ms	▷	☆
		python-application-01	7 days 2 hr log	N/A	0.33 sec	▷	☆
		python-application-02	7 days 1 hr log	N/A	0.28 sec	▷	☆
		python-application-03	7 days 1 hr log	N/A	0.36 sec	▷	☆

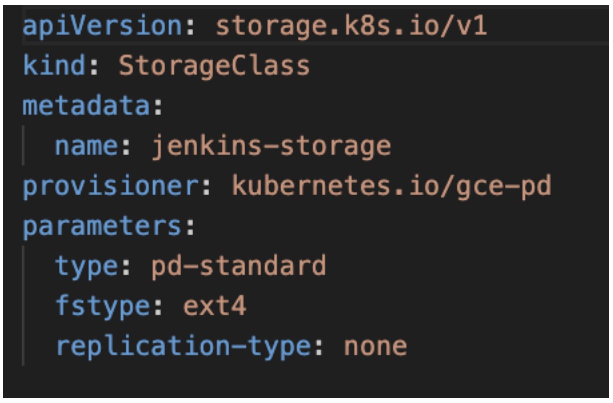
Figure 15: Deployed Jenkins Server

There is a persistence volume is used to store the data of the Jenkins and backup data in a restart of the Jenkins Server Pod.



```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: jenkins-pv
spec:
  storageClassName: "jenkins-storage"
  capacity:
    storage: '100'
  accessModes:
    - ReadWriteMany
  claimRef:
    namespace: jenkins
    name: jenkins-pvc
  gcePersistentDisk:
    pdName: jenkins-pv
    fsType: ext4
```

Figure 16: Persistence Volume for Jenkins Server

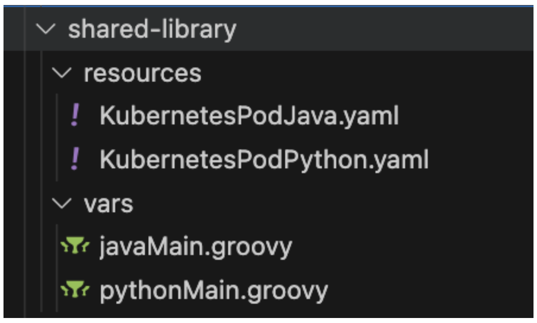


```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: jenkins-storage
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  fstype: ext4
  replication-type: none
```

Figure 17: Storage Class Created for Persistence Volumes

4.4.3.2. Configure the Jenkins Server to use Kubernetes Slave Nodes

The clouds can be integrated into the Jenkins server. In this scenario, the Kubernetes Cluster is used as a cloud and integrated Kubernetes Cluster with Jenkins to allow Jenkins jobs to provision pods as Jenkins slave nodes. A shared folder is used to run Jenkins jobs commonly for all the pipelines and the common files are used to run the continuous integration using supportive Kubernetes slave nodes. One slave pod contains several containers such as JNLP container, maven version 8, maven version 11, Python 3 for specifically that needed for each slave pod. The shared folder is called shared-library, and this will be extended with the programming languages or the pipeline type that needed to be run commonly.



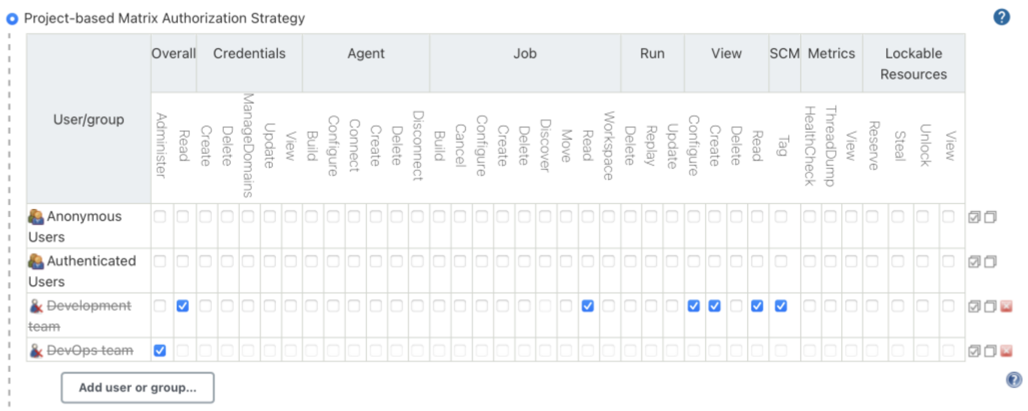
```
∨ shared-library
  ∨ resources
     ! KubernetesPodJava.yaml
     ! KubernetesPodPython.yaml
  ∨ vars
     javaMain.groovy
     pythonMain.groovy
```

Figure 18: Shared Library Resources

The slave pods are provisioned in a specifically configured namespace in the Kubernetes Cluster. The name can be any with the preference of the DevOps Team.

4.4.3.3. Configure the Jenkins Server with GitHub SSO

The GitHub SSO based on SAML is allowed to integrate with Jenkins to enhance the security with RBAC. Once the integration is successfully configured with Jenkins, the Jenkins login will be automatically redirected to the GitHub login. First, the GitHub authentication needed to be verified and then it checks the role based on the GitHub Teams. With the given permissions, the user may authorize to access the given options in the Jenkins Server.



● Project-based Matrix Authorization Strategy

User/group	Overall		Credentials					Agent						Job									Run			View				SCM	Metrics			Lockable Resources			
	Administer	Read	Create	Delete	ManageDomains	Update	View	Build	Configure	Connect	Create	Delete	Disconnect	Build	Cancel	Configure	Create	Delete	Discover	Move	Read	Workspace	Delete	Replay	Update	Configure	Create	Delete	Read	Tag	HealthCheck	ThreadDump	View	Reserve	Steal	Unlock	View
Anonymous Users																																					
Authenticated Users																																					
~~Development team~~		✓																			✓					✓	✓		✓	✓							
~~DevOps team~~	✓																																				

Add user or group...

Figure 19: Role Based Authentication with GitHub Teams

4.4.4 Develop the Fully Automated and Optimized Continuous Deployment Process

The main responsibility of this component is automating the application deployment process with well optimized based on the focused architectural attributes.

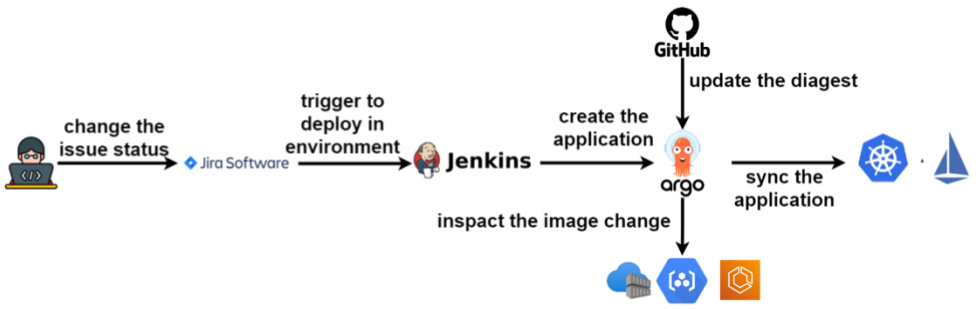
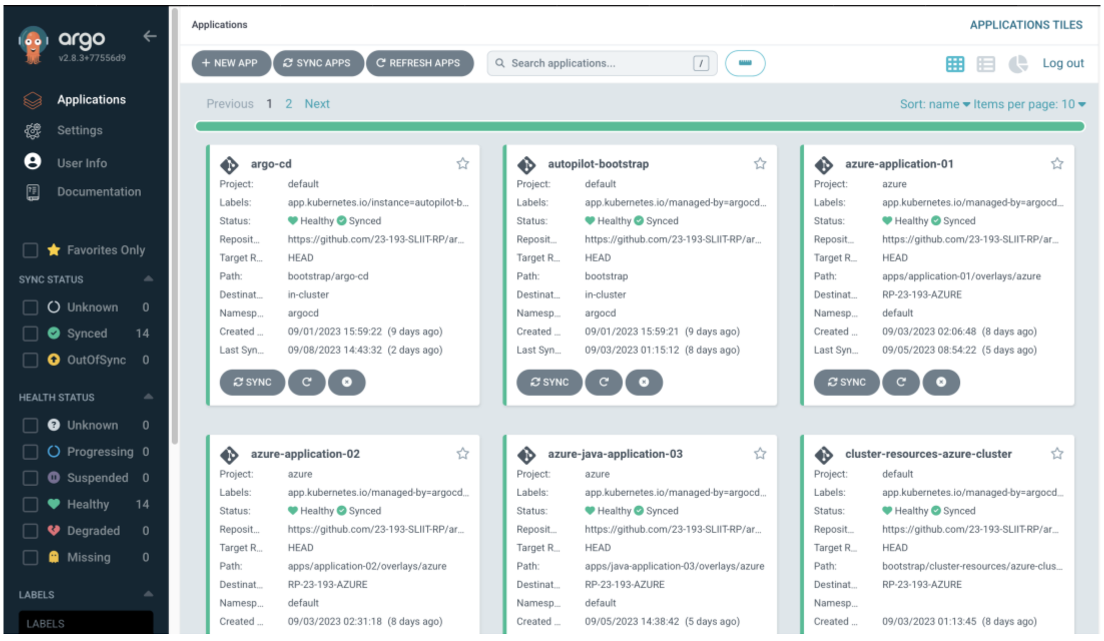


Figure 20: Fully Automated CD Architecture

The Argo CD is used as the centralized server to maintain the deployment process and ensure the service availability of the Kubernetes Cluster. The Argo CD server can automatically deploy the application over multi-cloud. The proposed system was developed to deploy in multi-cloud and selected two cloud providers as Azure and Google to provision the clusters.



argo-cd
Project: default
Labels: app.kubernetes.io/instance=autopilot-b...
Status: Healthy Synced
Reposit... https://github.com/23-193-SLIIT-RP/ar...
Target R... HEAD
Path: bootstrap/argo-cd
Destinat... in-cluster
Namesp... argocd
Created ... 09/01/2023 15:59:22 (9 days ago)
Last Syn... 09/08/2023 14:43:32 (2 days ago)

autopilot-bootstrap
Project: default
Labels: app.kubernetes.io/managed-by=argocd...
Status: Healthy Synced
Reposit... https://github.com/23-193-SLIIT-RP/ar...
Target R... HEAD
Path: bootstrap
Destinat... in-cluster
Namesp... argocd
Created ... 09/01/2023 15:59:21 (9 days ago)
Last Syn... 09/03/2023 01:15:12 (8 days ago)

azure-application-01
Project: azure
Labels: app.kubernetes.io/managed-by=argocd...
Status: Healthy Synced
Reposit... https://github.com/23-193-SLIIT-RP/ar...
Target R... HEAD
Path: apps/application-01/overlays/azure
Destinat... RP-23-193-AZURE
Namesp... default
Created ... 09/03/2023 02:06:48 (8 days ago)
Last Syn... 09/05/2023 08:54:22 (5 days ago)

azure-application-02
Project: azure
Labels: app.kubernetes.io/managed-by=argocd...
Status: Healthy Synced
Reposit... https://github.com/23-193-SLIIT-RP/ar...
Target R... HEAD
Path: apps/application-02/overlays/azure
Destinat... RP-23-193-AZURE
Namesp... default
Created ... 09/03/2023 02:31:18 (8 days ago)

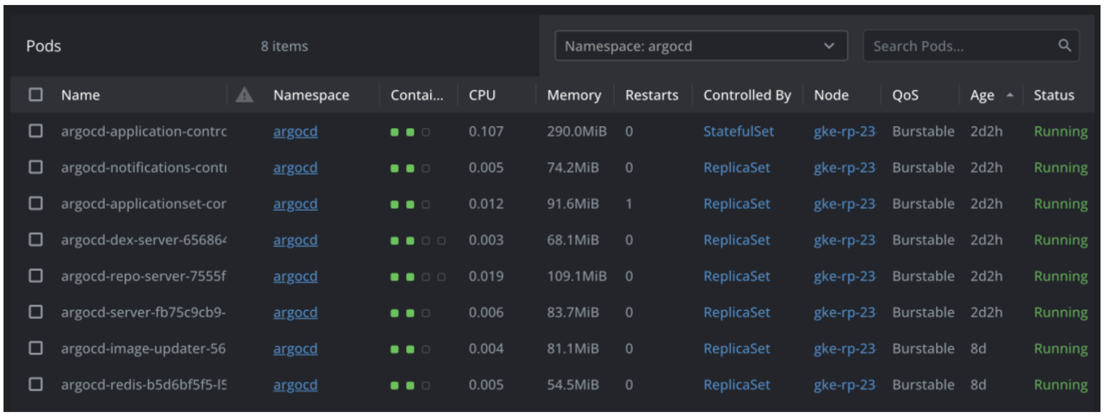
azure-java-application-03
Project: azure
Labels: app.kubernetes.io/managed-by=argocd...
Status: Healthy Synced
Reposit... https://github.com/23-193-SLIIT-RP/ar...
Target R... HEAD
Path: apps/java-application-03/overlays/azure
Destinat... RP-23-193-AZURE
Namesp... default
Created ... 09/05/2023 14:38:42 (5 days ago)

cluster-resources-azure-cluster
Project: default
Labels: app.kubernetes.io/managed-by=argocd...
Status: Healthy Synced
Reposit... https://github.com/23-193-SLIIT-RP/ar...
Target R... HEAD
Path: bootstrap/cluster-resources/azure-clus...
Destinat... RP-23-193-AZURE
Namesp...
Created ... 09/03/2023 01:13:45 (8 days ago)

Figure 21: Deployed Argo CD Server

The users can define the environment or the cloud that they needed to deploy the application with Jira automatically. The Jira includes a flow that can be simply add a task and change the environment that needed to deploy the application. Same process we can deploy in multi-cloud.

The Argo CD image updater is a separate component that can be used to deploy the applications to the mentioned environment automatically once a new container image is pushed to the private container registry. The Image Updater is monitoring the registered container registry for any of a new container image and update the manifest file in the GitHub related to the Argo CD deployment that used by the Argo CD server to pick the new container image ID. Once Argo CD pick the container image ID, it compares the currently deployed version and deploy the new pod related to the application in the Kubernetes cluster with rolling update feature to ensure the service availability.


Pods 8 items Namespace: argocd Search Pods...

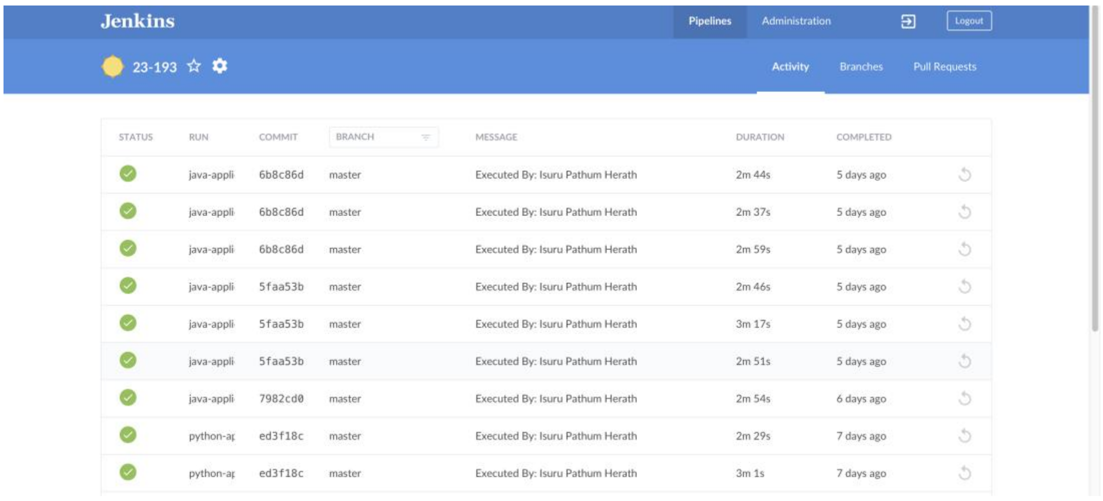
Name	Namespace	Contai...	CPU	Memory	Restarts	Controlled By	Node	QoS	Age	Status
argocd-application-contrc	argocd	■■□	0.107	290.0MiB	0	StatefulSet	gke-rp-23	Burstable	2d2h	Running
argocd-notifications-conti	argocd	■■□	0.005	74.2MiB	0	ReplicaSet	gke-rp-23	Burstable	2d2h	Running
argocd-applicationset-cor	argocd	■■□	0.012	91.6MiB	1	ReplicaSet	gke-rp-23	Burstable	2d2h	Running
argocd-dex-server-65686	argocd	■■□□	0.003	68.1MiB	0	ReplicaSet	gke-rp-23	Burstable	2d2h	Running
argocd-repo-server-7555f	argocd	■■□□	0.019	109.1MiB	0	ReplicaSet	gke-rp-23	Burstable	2d2h	Running
argocd-server-fb75c9cb9-	argocd	■■□	0.006	83.7MiB	0	ReplicaSet	gke-rp-23	Burstable	2d2h	Running
argocd-image-updater-56	argocd	■■□	0.004	81.1MiB	0	ReplicaSet	gke-rp-23	Burstable	8d	Running
argocd-redis-b5d6bf5f5-l5	argocd	■■□	0.005	54.5MiB	0	ReplicaSet	gke-rp-23	Burstable	8d	Running

Figure 22: Deployed Argo CD Server Components

If the new image is failed to deploy due to an error in the container image, the available service will be served without any service unavailability to the end user until the error is fixed and the new image is pushed it to the container registry.

4.4.4.1. Configure automated deployment Jenkins Jobs

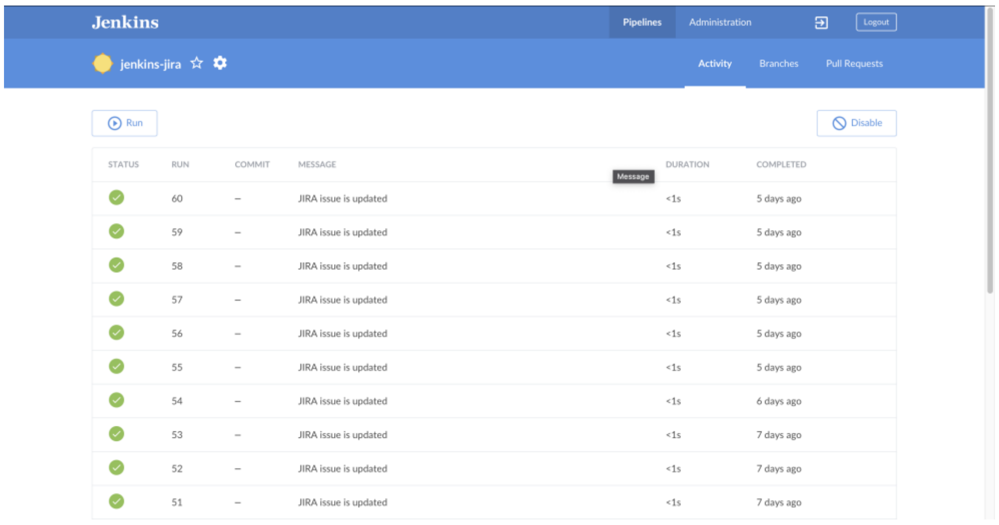
There is a centralized Jenkins pipeline job to deploy the necessary application in the selected environments and multi-cloud. The pipeline creates an Argo CD application by checking whether the application is available in the Argo CD Server. If the application is already available in the server, it deploys the new version of the container image or if the application is not available, it will automatically create by the Jenkins job.



STATUS	RUN	COMMIT	BRANCH	MESSAGE	DURATION	COMPLETED
✓	java-appli	6b8c86d	master	Executed By: Isuru Pathum Herath	2m 44s	5 days ago
✓	java-appli	6b8c86d	master	Executed By: Isuru Pathum Herath	2m 37s	5 days ago
✓	java-appli	6b8c86d	master	Executed By: Isuru Pathum Herath	2m 59s	5 days ago
✓	java-appli	5faa53b	master	Executed By: Isuru Pathum Herath	2m 46s	5 days ago
✓	java-appli	5faa53b	master	Executed By: Isuru Pathum Herath	3m 17s	5 days ago
✓	java-appli	5faa53b	master	Executed By: Isuru Pathum Herath	2m 51s	5 days ago
✓	java-appli	7982cd0	master	Executed By: Isuru Pathum Herath	2m 54s	6 days ago
✓	python-ap	ed3f18c	master	Executed By: Isuru Pathum Herath	2m 29s	7 days ago
✓	python-ap	ed3f18c	master	Executed By: Isuru Pathum Herath	3m 1s	7 days ago

Figure 23: Centralized Jenkins Job to Deploy Applications

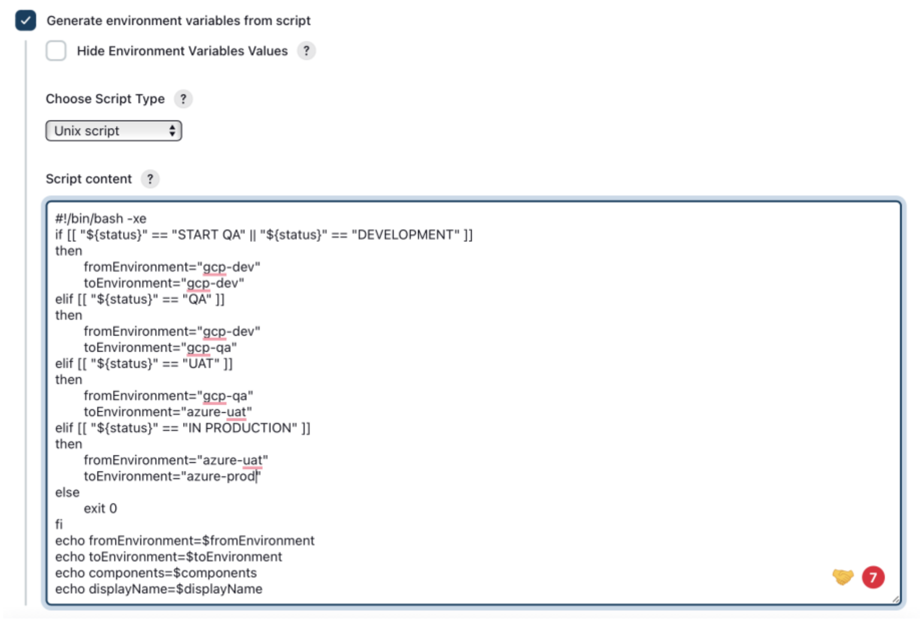
The centralized job is triggered by another job called “Jenkins-jira” that get the deployment data from the Jira updates. The triggering pipeline is getting necessary details from the Jira Server such as cloud details, environment details, application name and the name of the user who needs to deploy the application. Once the webhook triggers with the necessary data, the triggering pipeline triggers the centralized pipeline job by passing the necessary data to deploy in the relevant cloud and the environment.



STATUS	RUN	COMMIT	MESSAGE	DURATION	COMPLETED
✓	60	–	JIRA issue is updated	<1s	5 days ago
✓	59	–	JIRA issue is updated	<1s	5 days ago
✓	58	–	JIRA issue is updated	<1s	5 days ago
✓	57	–	JIRA issue is updated	<1s	5 days ago
✓	56	–	JIRA issue is updated	<1s	5 days ago
✓	55	–	JIRA issue is updated	<1s	5 days ago
✓	54	–	JIRA issue is updated	<1s	6 days ago
✓	53	–	JIRA issue is updated	<1s	7 days ago
✓	52	–	JIRA issue is updated	<1s	7 days ago
✓	51	–	JIRA issue is updated	<1s	7 days ago

Figure 24: Triggering Pipeline called "jenkins-jira"

The pipeline picks the data and generate environment variables from a script.


Generate environment variables from script
Hide Environment Variables Values
Choose Script Type
Unix script
Script content

```
#!/bin/bash -xe
if [[ "${status}" == "START QA" || "${status}" == "DEVELOPMENT" ]]
then
        fromEnvironment="gcp-dev"
        toEnvironment="gcp-dev"
elif [[ "${status}" == "QA" ]]
then
        fromEnvironment="gcp-dev"
        toEnvironment="gcp-qa"
elif [[ "${status}" == "UAT" ]]
then
        fromEnvironment="gcp-qa"
        toEnvironment="azure-uat"
elif [[ "${status}" == "IN PRODUCTION" ]]
then
        fromEnvironment="azure-uat"
        toEnvironment="azure-prod"
else
        exit 0
fi
echo fromEnvironment=$fromEnvironment
echo toEnvironment=$toEnvironment
echo components=$components
echo displayName=$displayName
```

Figure 25: Generate Environment Variable by Script

4.4.4.2. Configure Jira to Deployment Process

The Jira Cloud is used as the deployment triggering service. Developers can create a task related to the assigned developments and it can be follows until it goes to production. The Jira project is created and there is a customized workflow is used to identify the cloud environments and the deployment environments.

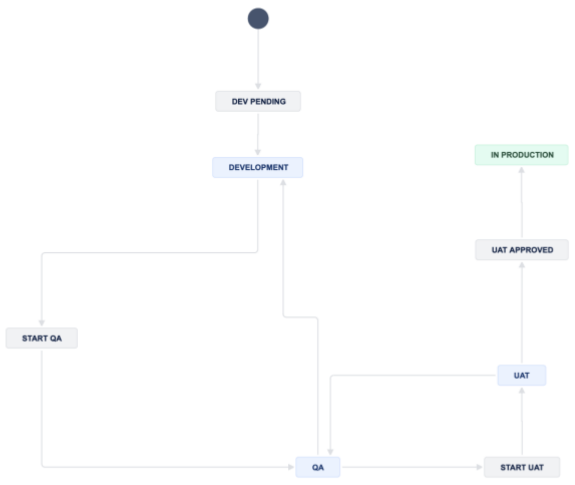


Figure 26: Jira Workflow to deployment process.

4.5 Project Requirements

4.5.1 Functional Requirements

Following are the functional requirements of the proposed system.

- Automatically create the container image when source code is updated in the SCM system.
- Automatically push the container image to the private container registry
- Automatically trigger deployment pipelines
- Automatically deploy the application on multi-cloud
- Optimized based on architectural attributes such as performance, security, extendibility, reliability, modifiability.
- Ensure the application availability.
- Minimize the deployment time and resource usage for deployments.
- Fast deployment by fully automated sync process with GitOps

4.5.2 Non-Functional Requirements

The following are the non-functional requirements focused on during the proposed system's development.

- Availability
- Usability
- Speed
- Visibility

4.6 Commercialization

The proposed fully automated and optimized software release process, which utilizes open-source solutions, can be a cost-effective solution for software companies and the industry. By utilizing open-source solutions, organizations can reduce the cost of licensing proprietary software and access a wide range of tools and resources for development and deployment. The use of open-source solutions also promotes collaboration and knowledge-sharing within the industry.

The automated and optimized release process can help organizations save costs by reducing the manual effort required for releases, minimizing the risk of errors, and reducing the need for additional resources. The optimized architectural attributes of the software release process can help ensure that the releases are of high quality and meet the desired performance, security, extendibility, reliability, and modifiability requirements.

The benefits of this solution can be commercialized in the following ways:

1. The automated release process can help organizations save costs by reducing the manual effort required for releases, minimizing the risk of errors, and reducing the need for additional resources.
2. By automating the release process, organizations can reduce the time required for each release, which can lead to faster time-to-market for their products.
3. The optimized architectural attributes of the software release process can help ensure that the releases are of high quality and meet the desired performance, security, extendibility, reliability, and modifiability requirements.
4. By adopting an automated and optimized release process, organizations can differentiate themselves from their competitors by offering faster and higher quality software releases.
5. As more organizations adopt the automated and optimized release process, it can become an industry standard, which can further enhance the commercialization of this solution.

By adopting an automated and optimized release process that utilizes open-source solutions, organizations can gain a competitive advantage in the industry by offering faster and higher quality software releases at a lower cost. This can lead to increased efficiency, improved quality, and overall cost savings for software companies and the industry.

5. RESULTS AND DISCUSSION

5.1 Test Results

There are several applications are deployed via the developed software release process and tested based on the programming languages. Initially tested Java and Python programming languages in the test environment to ensure the CI and CD optimizations based on the time. The architecture solution is deployed in the enterprise level company as an enhanced solution for their software release process. The test results from the company are as follows based on a comparison of previous system and proposed system. Based on the programming language, the result outcome is showing the performance improvement of the CI Process and the CD Process.

Programming Language	Average Time to build in previous system	Average Time to build with proposed system	Average builds per day	Performance Improvement
Java (Spring Boot)	10 - 15 min	5 - 10 min	30 +	33.33 %
JavaScript (React)	15 - 25 min	5 - 15 min	10 +	40 %

Table 2: Performance Improvement of CI Process

Programming Language	Average Time to deploy in previous system	Average Time to deploy with proposed system	Average deployments per day	Performance Improvement
Java (Spring Boot)	10 - 15 min	4 - 5 min	50 +	66.66 %
JavaScript (React)	15 - 25 min	4 - 5 min	30 +	80 %

Table 3: Performance Improvement of CI Process

5.2 Research Findings

The available systems are partially automated or manually triggered according to the research findings. The companies' software release processes are investigated to identify the necessary components of the software release process and commonly architect an architecture to cover whole CI/CD process. Different companies use different SCM flows such as GitHub Flow, Git Flow, GitLab Flow, Git Bucket Flow. There was not any solution that can be fully automate the software release process for the mentioned SCM flows commonly.

Optimization the cost and the architectural attributes such as performance, security, extendibility, reliability, and modifiability are very difficult with the available system. The available services or the systems are cost effective and needed to replace with open-source and free tools with integrating the enhancement of the security. Argo CD and Jenkins comes up to use as the tools to architect the fully automated software release process with considering the necessary automations and optimization approaches.

6. CONCLUSION

The proposed research project aims to optimize and automate the software release process for microservice-based architectures on multi-cloud platforms. Microservice architectures are becoming increasingly popular due to their scalability, flexibility, and dependability. However, deploying and managing microservices on multi-cloud platforms can be complex and time-consuming, particularly when it comes to releasing new software updates. The software release process consists of several phases, including development, testing, staging, and deployment. Each step requires a unique set of resources and tools, and each stage introduces the potential for human error. By optimizing and automating the software release process, organizations can reduce the risk of errors, enhance release quality, and speed up the deployment process.

To fully automate the release process, organizations need to use tools and technologies to automatically create, test, and deploy software updates. CI/CD pipelines, which automate the entire release process from code changes to deployment, can help with this. By eliminating the need for manual intervention, organizations can reduce the risk of errors and increase deployment speed. Inefficiencies in the software delivery process must be identified and eliminated to optimize it. This can be achieved through identifying bottlenecks, automating repetitive tasks, and improving resource utilization. By optimizing the release process, organizations can reduce costs, improve resource management, and increase productivity.

The research is focused to optimize the software release process according to five architectural attributes such as performance, security, extendibility, reliability, and modifiability. The solution is proposed using open-source solutions available for DevOps and cloud platforms. Several factors must be considered to streamline the software release process for microservice-based architectures on cloud platforms, including containerization, orchestration, automation, monitoring, and security. By adopting best practices for each of these factors, organizations can improve the reliability and scalability of their microservices, reduce downtime, multi-cloud deployment solution and ensure application security.

REFERENCES

[1] "Exploring the Benefits of Combining DevOps and Agile," Fernando Almeida, Jorge Simoes, Sergio Lopes, 19 February 2022.

[2] "Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects," Ionut-Catalin Donca, Ovidiu Petru Stan, Marius Misaros, Dan Gota, Liviu Miclea, 20 June 2022.

[3] "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation," Grzegorz Blinowski, Anna Ojdowska, Adam Przybyłek, 18 February 2022

[4] "Microservice Architecture Practices and Experience: a Focused Look on Docker Configuration Files," Luciano Baresi, Giovanni Quattrocchi, Damian Andrew Tamburri, 6 Dec 2022.

[5] "The key differences between Kubernetes and Docker and how they fit into containerization", Josh Campbell, [Online]. Available: https://www.atlassian.com/microservices/microservices-architecture/kubernetes-vs-docker.

[6] "Analyzing and Monitoring Kubernetes Microservices based on Distributed Tracing and Service Mesh," Yu-Te Wang, Shang-Pin Ma, Yue-Jun Lai, Yan-Cih Liang, December 2022. [Accessed 04 March 2023].

[7] "The Challenges and Mitigation Strategies of Using DevOps during Software Development," Dhaya Sindhu Battina, 1 January 2021.

[8] "Performance Assessment of Traditional Software Development Methodologies and DevOps Automation Culture," P. Narang, P. Mittal, December 2022.

[9] "Kubernetes - evolution of virtualization," Marek Moravcik, Martin Kontsek, Pavel Segec, David Cymbalak, 15 December 2022.

[10] "Effect of Using Continuous Integration (CI) and Continuous Delivery (CD) Deployment in DevOps to reduce the Gap between Developer and Operation," Abrar Mohammad Mowad, Hamed Fawareh, Mohammad A. Hassan, 04 January 2023.

[11] “Containerized Microservices Orchestration and Provisioning in CloudComputing: A Conceptual Framework and Future Perspectives,” Abdul Saboor, Mohd Fadzil Hassan, Rehan Akbar, Syed Nasir Mehmood Shah, Farrukh Hassan, Saeed Ahmed Magsi, Muhammad Aadil Siddiqui, 07 June 2022.

[12] “CLOUD DATA SECURITY METHODS: KUBERNETES VS DOCKER SWARM,” Avinash Ganne, December-2022.

[13] “Challenges and solutions when adopting DevSecOps,” Roshan N. Rajapakse, Mansooreh Zahedi, M. Ali Babar, Haifeng Shen, January 2022.

[14] “Performance Study of Kubernetes Cluster Deployed on Openstack,VMs and BareMetal,” Yeddula Sai Dhanush Reddy, Padumati Saikiran Reddy, Nithya Ganesan, B. Thangaraju, 30 August 2022.

[15] “Automation of Configuration, Initialization and Deployment of Applications Based on an Algebraic Approach,” PavelShapkin, 26 November 2022.

[16] “Automated Unit Testing of Hydrologic Modeling Software with CI/CD and Jenkins,” Levi T. Connelly, Melody L. Hammel, Benjamin T. Eger, Lan Lin, 2022.

[17] “AUTOMATED TESTING IN A CI/CD PIPELINE,” Ville Santala, 2022.

[18] “Testing resource allocation for software with multiple versions,” Adarsh Anand, Subhrata Das, Ompal Singh, Vijay Kumar, February 19, 2022.

[19] “Reliability modelling and optimization for microservice-based cloud application using multi-agent system”, Zheng Liu, Huiqun Yu, Guisheng Fan, Liqiong Chen, 13 March 2022.

[20] Z. Zhu, M. A. Naeem, and L. Li, "Automated Continuous Deployment at Facebook," in Proceedings of the 17th International Conference on Mining Software Repositories, IEEE Press, 2020, pp. 460-470.

[21] [21] H. Guo, J. Chen, T. Li, Z. Li, L. Zhang, and J. Li, "An optimization model for software release management based on continuous delivery," Information and Software Technology, vol. 107, pp. 101-115, 2019H. Guo, J. Chen, T. Li, Z. Li, L. Zhang, and J. Li, "An optimization model for software release management based on continuous delivery," Information and Software Technology, vol. 107, pp. 101-115, 2019

[22] “AUTOMA TING DEPLOYMENTS OF THE LA TEST APPLICA TION VERSION USING CI-CD WORKFLOW”, 2022, Spoorthi Jayaprakash Malgund, Dr Sowmyarani C N. .

[23] “Commit, Release, Package: Automation in the development process for the ReFEx GNC System,” Sommer, Jan, 9 February 2022. .

[24] Grzegorz Blinowski, Anna Ojdowska, Adam Przybyłek, “Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation”, 18 February 2022.

[25] Luciano Baresi, Giovanni Quattrocchi, Damian Andrew Tamburri, “Microservice Architecture Practices and Experience: a Focused Look on Docker Configuration Files”, 6 Dec 2022.

[26] Josh Campbell, “The key differences between Kubernetes and Docker and how they fit into containerization”, [Online]. Available: https://www.atlassian.com/microservices/microservices-architecture/kubernetes-vs-docker, [Accessed 04 March 2023].

[27] Ionut-Catalin Donca, Ovidiu Petru Stan, Marius Misaros, Dan Gota, Liviu Miclea, “Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects”, 2022.

[28] Fernando Almeida, Jorge Simoes, Sergio Lopes, “Exploring the Benefits of Combining DevOps and Agile”, 2022.

[29] Yeddula Sai Dhanush Reddy, Padumati Saikiran Reddy, Nithya Ganesan, B. Thangaraju, “Performance Study of Kubernetes Cluster Deployed on OpenStack, VMs and BareMetal”, 30 August 2022.

[30] PavelShapkin, “Automation of Configuration, Initialization and Deployment of Applications Based on an Algebraic Approach”, 2022.

[31] D. C Kalubowila, S. M Athukorala, B. A. S Tharaka, H. W. Y. R Samarasekara, Udara Srimath S. Samaratunge Arachchilage, Dharshana Kasthurirathna, “Optimization of Microservices Security,” December 2021.

[32] Adarsh Anand, Subrata Das, Ompal Singh, Vijay Kumar, “Testing resource allocation for software with multiple versions”, February 19, 2022.

[33] Zheng Liu, Huiqun Yu, Guisheng Fan, Liqiong Chen, “Reliability modelling and optimization for microservice-based cloud application using multi-agent system”, 13 March 2022.

[34] Z. Zhu, M. A. Naeem, and L. Li, "Automated Continuous Deployment at Facebook," in Proceedings of the 17th International Conference on Mining Software Repositories, IEEE Press, 2020, pp. 460-470.

[35] H. Guo, J. Chen, T. Li, Z. Li, L. Zhang, and J. Li, "An optimization model for software release management based on continuous delivery," Information and Software Technology, vol. 107, pp. 101-115, 2019.

[36] Spoorthi Jayaprakash Malgund, Dr Sowmyarani C N, “Automating Deployments Of The Latest Application Version Using Ci-Cd Workflow”, 2022.

[37] Yutsuki Miyashita, Yuki Yamada, Hiroaki Hashiura, Atsuo Hazeyama, “Design of the Inspection Process Using the GitHub Flow in Project Based Learning for Software Engineering and Its Practice", 6 Feb 2022.

[38] Meixia Yang, Ming Hung, “A Microservice-Based OpenStack Monitoring Tool”,19 March 2020,doi: 10.1109/ICSESS47205.2019.9040740.

[39] Lei Chen, Ming Xian, JianLiu, “Monitoring System of OpenStack Cloud Platform Based on Prometheus”,10-12 July 2020, doi:10.1109/CVIDL51233.2020.0-100.

[40] Abhishek Pratap Singh, “A Data Visualization Tool- Grafana”, Jan 26 2023.

[41] Andrea Janes, Xiaozhu Li, Valentina Lenarduzzi, “Open tracing tools: Overview and critical comparison”, Vol. 204, October 2023, 111793.

[42] Mahantesh Birje, Chetan Bulla, “Commerical and Open Source Cloud Monitoring tools: A Review”, January 2020, doi: 10.1007/978-3-030-24322-7_59

[43] Shubo Zhang1, Tianyang Wu1, Maolin Pan1, Chaomeng Zhang2 and Yang Yu, “A-SARSA: A Predictive Container Auto-Scaling Algorithm Based on Reinforcement Learning”, 2020 IEEE International Conference on Web Services (ICWS).

[44] Nhat-Minh Dang-Quang and Myungsik Yoo, “Deep Learning-Based Autoscaling Using Bidirectional Long Short-Term Memory for Kubernetes”, Applied Science – 11

[45] weave works, “Ensure High Availability and Uptime With Kubernetes HPA (Horizontal Pod Autoscaler) and Prometheus”, https://www.weave.works/blog/kubernetes-horizontal-pod-autoscaler-andprometheus

[46] Wei Fang; ZhiHui Lu; Jie Wu; ZhenYin Cao, “RPPS: A Novel Resource Prediction and Provisioning Scheme in Cloud Data Center”,2012 IEEE Ninth International Conference on Services Computing

[47] Mahmoud Imdoukh, Imtiaz Ahmad & Mohammad Gh. Alfailakawi,"Machine learning-based auto-scaling for containerized applications", https://link.springer.com/article/10.1007/s00521-019-04507-z

[48] Xuehai Tang, Qiuyang Liu , Yangchen Dong, Jizhong Han, Zhiyuan Zhang ,“Fisher: An Efficient Container Load Prediction Model with Deep Neural Network in Clouds”, 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communication.

APPENDIX

Appendix A: Jenkins Server Deployment YAML

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jenkins
  namespace: jenkins
spec:
  replicas: 1
  selector:
    matchLabels:
      name: jenkins
  template:
    metadata:
      labels:
        name: jenkins
    spec:
      volumes:
        - name: jenkins-nfs-pvc-new
          persistentVolumeClaim:
            claimName: jenkins-pvc
        - name: config
          secret:
            secretName: config
      containers:
        - name: kaniko
          image: gcr.io/kaniko-project/executor:debug
          command:
            - /busybox/cat
          resources: {}
          volumeMounts:
            - name: jenkins-nfs-pvc-new
              mountPath: /var/jenkins_home
            - name: config
              mountPath: /var/jenkins_home/config
          terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    imagePullPolicy: IfNotPresent
    tty: true
  - name: jenkins
    image: jenkins/jenkins:2.404-jdk11
    ports:
      - name: http
        containerPort: 8080
        protocol: TCP
      - name: executor
        containerPort: 50000
        protocol: TCP
    env:
      - name: "JENKINS_OPTS"
        value: "--prefix=/jenkins"
      - name: "JAVA_OPTS"
        value: "-Xmx9800m"
    resources:
      limits:
        cpu: '1'
        memory: 1Gi
      requests:
        cpu: 500m
        memory: 500Mi
    volumeMounts:
      - name: jenkins-nfs-pvc-new
        mountPath: /var/jenkins_home
      - name: config
        mountPath: /var/jenkins_home/config
    livenessProbe:
      httpGet:
        path: /jenkins/login
        port: 8080
        scheme: HTTP
      initialDelaySeconds: 120
      timeoutSeconds: 3
      periodSeconds: 10
```

```
          successThreshold: 1
          failureThreshold: 3
        readinessProbe:
          httpGet:
            path: /jenkins/login
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 120
          timeoutSeconds: 3
          periodSeconds: 10
          successThreshold: 1
          failureThreshold: 3
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        imagePullPolicy: IfNotPresent
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
      serviceAccountName: jenkins
      securityContext:
          fsGroup: 1000
          runAsUser: 1000
      schedulerName: default-scheduler
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
      maxSurge: 25%

  revisionHistoryLimit: 10
  progressDeadlineSeconds: 600
```

Appendix B: Jenkins Server Service YAML

```
apiVersion: v1
kind: Service
metadata:
  name: jenkins
  namespace: jenkins
```

```yaml
spec:
  ports:
    - name: http
      protocol: TCP
      port: 8080
      targetPort: 8080
    - name: agent
      protocol: TCP
      port: 50000
      targetPort: 50000
  selector:
    name: jenkins
  type: ClusterIP
```

Appendix C: Jenkins Server Virtual Service YAML

```yaml
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  namespace: jenkins
  name: jenkins
spec:
  gateways:
    - default/argocd-gateway
  hosts:
    - releasex.tech
  http:
  - match:
    - uri:
        prefix: "/jenkins"
    route:
    - destination:
        port:
          number: 8080
        host: jenkins.jenkins.svc.cluster.local
```

Appendix D: Jenkins Server Storage Class YAML

```yaml
apiVersion: storage.k8s.io/v1
```

```
kind: StorageClass
metadata:
  name: jenkins-storage
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  fstype: ext4
  replication-type: none
```

Appendix E: Jenkins Server Persistence Volume Claim YAML

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: jenkins-pvc
  namespace: jenkins
spec:
  storageClassName: "jenkins-storage"
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Gi
```

Appendix F: Kubernetes & Jenkins integration RBAC Role Service Account

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: jenkins-admin
  namespace: builder-jenkins
---

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: jenkins
  namespace: builder-jenkins
  labels:
```

```
  "app.kubernetes.io/name": 'jenkins'
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: [""]
  resources: ["pods/log"]
  verbs: ["get","list","watch"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
---

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: jenkins-role-binding
  namespace: builder-jenkins
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: jenkins
subjects:
- kind: ServiceAccount
  name: jenkins-admin
  namespace: builder-jenkins
```

Appendix E: Shared Volume Storage Class

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: jenkins-bilder-storage
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  fstype: ext4
  replication-type: none
```

Appendix F: Shared Volume Java, Node & Python Persistence Volume Claims

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 name: jenkins-python-bilder-storage-pvc
 namespace: builder-jenkins
spec:
 storageClassName: "jenkins-bilder-storage"
 accessModes:
  - ReadWriteOnce
 resources:
  requests:
   storage: 10Gi
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 name: jenkins-node-bilder-storage-pvc
 namespace: builder-jenkins
spec:
 storageClassName: "jenkins-bilder-storage"
 accessModes:
  - ReadWriteOnce
 resources:
  requests:
   storage: 10Gi
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
 name: jenkins-python-bilder-storage-pvc
 namespace: builder-jenkins
```

```
spec:
 storageClassName: "jenkins-bilder-storage"
 accessModes:
  - ReadWriteOnce
 resources:
  requests:
   storage: 10Gi
```

Appendix G: Centralized Deployment Pipeline Jenkins File

```
def componentsArray
def componentNameList=''

pipeline {
    agent {
        kubernetes {
            defaultContainer 'argocd-cli'
            yamlFile 'KubernetesPod.yaml'
        }
    }

    parameters {
        string(name: 'componentsArray', defaultValue: 'name=loan-management,')
        string(name: 'fromEnvironment', defaultValue: 'dev')
        string(name: 'toEnvironment', defaultValue: 'qa')
        string(name: 'displayName', defaultValue: 'isurupathumherath')
    }
    environment {
        DOCKER_REGISTRY_CREDS = credentials('AZURE_CONTAINER_REGISTRY')
        ARGOCD_SERVER = 'argocd-server.argocd.svc.cluster.local:80'
        DOCKER_REGISTRY_URL = 'sharedregistry23.azurecr.io'

        registry="sharedregistry23.azurecr.io"
        scope="repository:application-01:pull,push"

acr_credential="U0hBUkVEUkVHSVNUUlkyMzpONDdkYzN5WUN1M2xIZDIza2JTbGFTREY0YnpqVn
BOcTVwTkZrYVZXZlErQUNSQ3ZTNFN5"
```

```
    }
    stages {
        stage("Preparing") {
            steps {
                container('curl') {
                    checkout scm
                    script {
                        currentBuild.description = "Executed By: ${displayName}"
                        componentsArray = params.componentsArray.split(', BasicComponent')
                        for (int i = 0; i < componentsArray.size(); i++) {
                            appName = (componentsArray[i] =~ /name=(.+),/)[ 0 ][ 1 ]
                            echo "appName: ${appName}"
                            componentNameList=(componentNameList.length()>0) ?
componentNameList+','+appName : componentNameList+appName

                            stage("${appName} (${toEnvironment})") {
                                sh '''
                                set +x
                                REPOSITORY=${appName}
                                TAG_OLD=${fromEnvironment}
                                TAG_NEW=${toEnvironment}
                                CONTENT_TYPE="application/vnd.docker.distribution.manifest.v2+json"
                                TOKEN=$(curl -v -H "Authorization: Basic $acr_credential"
"https://$registry/oauth2/token?service=$registry&scope=$scope" | jq -r .access_token)
                                echo "$TOKEN"
                                MANIFEST=$(curl -H "Accept: ${CONTENT_TYPE}" -H "Authorization: Bearer
${TOKEN}" "https://${registry}/v2/${REPOSITORY}/manifests/${TAG_OLD}")
                                echo "$MANIFEST"
                                curl -X PUT -H "Content-Type: ${CONTENT_TYPE}" -H "Authorization: Bearer
${TOKEN}" -d "${MANIFEST}" "https://${registry}/v2/${REPOSITORY}/manifests/${TAG_NEW}"
                                set -x
                                '''
                            }

                        }
                        currentBuild.displayName = "${componentNameList}"
                    }
                }
            }
        }
        stage("Deploy") {
            steps {
                container('argocd-cli') {
                    withCredentials([string(credentialsId: 'github_token', variable: 'github_token')]) {
                        script {
                            sh '''
                                git config --global user.email "it20125516@my.sliit.lk"
                                git config --global user.name "isurupathumherath"
                            '''
                            for (int i = 0; i < componentsArray.size(); i++) {
                                appName = (componentsArray[i] =~ /name=(.+),/)[ 0 ][ 1 ]
                                if (appName.contains("-")){
                                  countryCode=appName.split('-')[1];
                                }
                                filePathList = sh ( script: "find ./common-resources -type d -name ${appName}" ,
returnStdout: true).trim().split('\n')
                                echo "${filePathList}"

                                namespace = sh ( script: "echo '${filePathList[0]}' | cut -d/ -f 3-3" , returnStdout:
true).trim()
                                def argocdAppName =sh ( script: """echo \$(argocd-autopilot application list
${toEnvironment} --git-token ${github_token} --repo https://github.com/23-193-SLIIT-RP/argocd-
gitops.git | grep ${appName} ) | grep -E "(^| )\""""+appName+""""( |\$)" && echo 'matched' || true """,
returnStdout: true).trim()
                                if (!argocdAppName){
                                    echo "APPLICATION IS NOT AVAILABLE. CREATING NOW ... "
                                    sh "argocd-autopilot app create ${appName} --app github.com/23-193-SLIIT-
RP/argocd-gitops/common-resources/${namespace}/${appName}/ -p ${toEnvironment} --git-token
${github_token} --repo https://github.com/23-193-SLIIT-RP/argocd-gitops.git"
                                } else {
                                    echo "APPLICATION IS AVAILABLE"
                                }
                    }
                  }
                }
              }
            }
          }

          stage("Wait") {
            steps {
              container('argocd-cli') {
                withCredentials([string(credentialsId: 'argocd_password', variable: 'ARGOCD_PASS')]) {
                  sh 'sleep 60'
                  sh "echo y | argocd login ${ARGOCD_SERVER} --insecure --username jenkins-ci --
password ${ARGOCD_PASS} --grpc-web-root-path /argo-cd"
                  script {
                    for (int i = 0; i < componentsArray.size(); i++) {
                      appName = (componentsArray[i] =~ /name=(.+),/)[ 0 ][ 1 ]
                      sh "argocd app wait ${toEnvironment}-${appName} --timeout 600"
                    }
                  }

                }
              }
            }
          }
        }
      }
```

Appendix H: Centralized Deployment Pipeline Slave Node

```
apiVersion: v1
kind: Pod
metadata:
  name: deploy-on-argocd
  namespace: builder-jenkins
spec:
  imagePullSecrets:
```

```
 - name: registry-secret
 containers:
 - name: jnlp
   image: jenkins/inbound-agent:latest
   command: ["/bin/sh","-c"]
   args: ["sleep 30; /usr/local/bin/jenkins-agent"]
 - name: curl
   image: sharedregistry23.azurecr.io/fusionx-curl:1.0.0
   command:
   - cat
   tty: true
 - name: argocd-cli
   image: sharedregistry23.azurecr.io/argocd-cli:2.1.1
   command:
   - cat
   tty: true
```

Appendix I: Shared Library Java Build Pipeline Jenkins File

```
apiVersion: v1
kind: Pod
metadata:
  name: jenkins-slave-spring-boot
  namespace: builder-jenkins
  labels:
    name: jenkins-slave-spring-boot
spec:
  containers:
  - name: jnlp
    image: jenkins/inbound-agent:latest
    command: ["/bin/sh","-c"]
    args: ["sleep 30; /usr/local/bin/jenkins-agent"]
  - name: kaniko
    image: gcr.io/kaniko-project/executor:debug
    command:
    - /busybox/cat
    tty: true
    volumeMounts:
```

```
    - name: kaniko-secret
      mountPath: /kaniko/.docker/
  - name: maven-11
    image: maven:3.6.0-jdk-11
    command:
    - cat
    tty: true
  - name: maven-8
    image: maven:3.6.0-jdk-8
    securityContext:
      privileged: true
    command:
    - cat
    tty: true
  volumes:
    - name: kaniko-secret
      secret:
        secretName: kaniko-secret
```

Appendix J: Shared Library Python Build Pipeline Jenkins File

```
apiVersion: v1
kind: Pod
metadata:
  name: jenkins-slave-spring-boot
  namespace: builder-jenkins
  labels:
    name: jenkins-slave-spring-boot
spec:
  containers:
  - name: jnlp
    image: jenkins/inbound-agent:latest
    command: ["/bin/sh","-c"]
    args: ["sleep 30; /usr/local/bin/jenkins-agent"]
  - name: kaniko
    image: gcr.io/kaniko-project/executor:debug
    command:
    - /busybox/cat
    tty: true
    volumeMounts:
      - name: kaniko-secret
        mountPath: /kaniko/.docker/
  - name: maven-11
    image: maven:3.6.0-jdk-11
    command:
    - cat
    tty: true
  - name: python-3
    image: python:3-alpine
    securityContext:
      privileged: true
    command:
    - cat
    tty: true
  volumes:
    - name: kaniko-secret
      secret:
        secretName: kaniko-secret
```

Appendix K: Shared Library Java Build Pipeline Kubernetes Slave Node

```
def call(body) {
  evaluate the body block, and collect configuration into the object
  def pipelineParams = [: ]
  body.resolveStrategy = Closure.DELEGATE_FIRST
  body.delegate = pipelineParams
  body()

  def jobnameparts = JOB_NAME.tokenize('/') as String[]
  def app_name = jobnameparts[0]
  def gitCommit = 'UNKNOWN'
  def applicationName = 'UNKNOWN'

pipeline {
  agent {
    kubernetes {
        defaultContainer 'maven-11'
        yaml libraryResource('KubernetesPodJava.yaml')
      }
    }

    stages {
        stage('package') {
            steps {
                sh "mvn package"
            }
        }

        stage("Unit Tests") {
          steps {
            container('maven-11') {
              sh 'mvn verify'
            }
          }
        }

        stage("Push to Registry") {
          steps {
            container('kaniko') {
              echo "${env.GIT_COMMIT}"
              script {
                gitCommit = "${env.GIT_COMMIT}"
              }
              echo "${gitCommit}"
              script {
                    def repositoryName = sh(returnStdout: true, script: "basename -s .git
${env.GIT_URL}").trim()
                    echo "Repository Name: ${repositoryName}"
                    sh "/kaniko/executor --dockerfile `pwd`/Dockerfile --context `pwd` --build-arg
GIT_COMMIT=$gitCommit --build-arg ARTIFACT=target/spring-boot-hello-world-lolc.jar --label
org.opencontainers.image.revision=$gitCommit --
destination=sharedregistry23.azurecr.io/$repositoryName:gcp"
              }
            }
          }
        }
      }
}


}
```

Appendix L: Shared Library Python Build Pipeline Jenkins File

```
def call(body) {
 evaluate the body block, and collect configuration into the object
 def pipelineParams = [: ]
 body.resolveStrategy = Closure.DELEGATE_FIRST
 body.delegate = pipelineParams
 body()

 def jobnameparts = JOB_NAME.tokenize('/') as String[]
 def app_name = jobnameparts[0]
 def gitCommit = 'UNKNOWN'
 def applicationName = 'UNKNOWN'

pipeline {
    agent {
      kubernetes {
        defaultContainer 'python-3'
        yaml libraryResource('KubernetesPodPython.yaml')
      }
    }

    stages {

        stage("Check Application Package") {
          steps {
            container('python-3') {
              sh 'pip install -r requirements.txt'
            }
          }
        }

        stage("Push to Registry") {
          steps {
            container('kaniko') {
              echo "${env.GIT_COMMIT}"
              script {
                gitCommit = "${env.GIT_COMMIT}"
              }
              echo "${gitCommit}"
              script {
                    def repositoryName = sh(returnStdout: true, script: "basename -s .git
${env.GIT_URL}").trim()
                    echo "Repository Name: ${repositoryName}"
                    sh "/kaniko/executor --dockerfile `pwd`/Dockerfile --context `pwd` --build-arg
GIT_COMMIT=$gitCommit --label org.opencontainers.image.revision=$gitCommit --
destination=sharedregistry23.azurecr.io/$repositoryName:gcp"
              }
            }
        }
      }
    }
}


}
```

Appendix M: Argo CD Config Management

```
apiVersion: v1
data:
 repository.credentials: |
   - passwordSecret:
       key: git_token
       name: autopilot-secret
     url: http://gitlab.sliit.lk/
     usernameSecret:
       key: git_username
       name: autopilot-secret
```

```
  timeout.reconciliation: 15s
  accounts.jenkins-ci: apiKey, login
kind: ConfigMap
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"repository.credentials":"- passwordSecret:\n    key: git_token\n    name:
autopilot-secret\n  url: http://gitlab.sliit.lk/\n  usernameSecret:\n    key: git_username\n    name:
autopilot-
secret\n","timeout.reconciliation":"15s"},"kind":"ConfigMap","metadata":{"annotations":{},"labels":{"app.k
ubernetes.io/instance":"argo-cd","app.kubernetes.io/name":"argocd-cm","app.kubernetes.io/part-
of":"argocd"},"name":"argocd-cm","namespace":"argocd"}}
  creationTimestamp: "2023-08-30T17:55:37Z"
  labels:
    app.kubernetes.io/instance: argo-cd
    app.kubernetes.io/name: argocd-cm
    app.kubernetes.io/part-of: argocd
  name: argocd-cm
  namespace: argocd
  resourceVersion: "3708764"
  uid: 5c6535c0-c730-4621-84a0-9df82485514b
```

Appendix N: Argo CD RBAC Management

```
apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    app.kubernetes.io/instance: argo-cd
    app.kubernetes.io/name: argocd-rbac-cm
    app.kubernetes.io/part-of: argocd
  name: argocd-rbac-cm
  namespace: argocd
  resourceVersion: "3169958"
  uid: 608a7b09-d38b-49b9-b6c5-2c2a89f7a7cb
data:
  policy.csv: |
    p, role:org-admin, clusters, get, *, allow
    p, role:org-admin, repositories, get, *, allow
    p, role:org-admin, repositories, create, *, allow
    p, role:org-admin, repositories, update, *, allow
    p, role:org-admin, repositories, delete, *, allow

    p, role:image-updater, applications, get, */*, allow
    p, role:image-updater, applications, update, */*, allow

    g, image-updater, role:image-updater
    g, jenkins-ci, role:org-admin

  policy.default: role:readonly
```

Appendix O: Argo CD Patch

```
- op: add
  path: "/spec/template/spec/containers/0/command/1"
  value: "--insecure"
- op: add
  path: "/spec/template/spec/containers/0/command/2"
  value: "--rootpath"
- op: add
  path: "/spec/template/spec/containers/0/command/3"
  value: "/argo-cd/"
```

Appendix P: Argo CD Virtual Service

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  namespace: argocd
  name: argocd-server
spec:
  hosts:
  - "*"
  gateways:
  - default/argocd-gateway
```

```yaml
 http:
  - match:
     - uri:
         prefix: /
    route:
     - destination:
         host: argocd-server.argocd.svc.cluster.local
         port:
           number: 80
```

Appendix Q: Main Ingress Gateway

```yaml
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  namespace: default
  name: main-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - hosts:
        - "*"
      port:
        name: http
        number: 80
        protocol: HTTP
      tls:
        httpsRedirect: false
```