Master of Science in Computer Science
September 2022

Zero Downtime Deployment approaches in Cloud Architecture

Efficiency of deployment process

Emmanuel Digvijay Roa Kastala

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full-time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s): Emmanuel Digvijay Rao Kastala
E-mail: emka17@student.bth.se

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

ABSTRACT

Background. Our dependence on applications running cloud or server-based applications has increased exponentially over the last decade through easy access to computers. Keeping web service running through updates and bug fixes is a challenge faced by the software development community. Hence, avoiding service interrupts and automating deployments have become more important now. This paper studies the efficiency and efficacy of three such deployment strategies for a web application. The deployment strategies are the Blue-Green deployment (can also be called A/B or Red Black) strategy, canary deployment, and rolling deployment.

Objectives. The objective of the thesis is to examine whether all three produce zero downtime during deployment and examine which deployment deploys a new service with the least amount of time. This study is done by deploying a new version of a web application on a cloud infrastructure during the application's service is online. To check whether the service interrupte during the deployment of a new version.

Methods. The study is carried out with two research methods: a Literature Review and experimentation. The main goal of the Literature Review is to gain insights into the existing approaches and techniques and the second method aims to Experiment with these approaches in a scenario, which in turn will produce the result required to understand and answer the research questions raised in the thesis. The three deployment approaches are selected. Selected three approaches will be evaluated using the same resources, application, and environment, and results will be presented.

Results. The performance of the above-mentioned models has been compared with each other. None of the development strategies produce downtime of the service during the deployment process. The experiments result show that the Blue Green is 395 sec faster than the Canary Deployment. Rolling deployment is 257 sec faster than Canary deployment.

Conclusions. The research revolves around the deployment strategy producing service zero downtime during deployment. Results show that one is faster than others and non produces non-zero downtime of services. This research will guide a developer to understand and pick the best deployment process for their software deployment process.

Keywords. Cloud-Native, Distributed System, Services Automation, Blue-Green Deployment, Canary Deployment, Rolling Deployment.

ACRONYMS

API : Application Programming Interface
IaaS : Infrastructure As A Service
PaaS : Platform As A Service
SaaS : Software As A Service
CI : Continues Integration
CD : Continuous Deployment
CPU : Central Processing Unit
RAM : Random Access Memory
OS : Operating Systems
VM : Virtual Machine
I/O : Input/Output
REST : Representational State Transfer
QoS : Quality of Services
QoP : Quality of Processes
IoT : Internet of Things
DNS : Domain Name System
AWS : Amazon Web Services
ISO : International Organization for Standardization
ROM : Read Only Memory
GUI : Graphical User Interface
YAML : Yet Another Markup Language
RT : Response Time
OD : Overlap Duration
ST : Switch Time
SD : Standard Deviation
SDT : Service Downtime
SLR : Systematic Literature Review

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my professor Emilia Mendes for understating and supporting me throughout my tough times. I would also like to thank director Gurudutt Velpula for his guidance throughout my MS in BTH.

I would like to thank my dear and best friends Sidhartha Srinadhuni and Neeraj Reddy Avutu for their presence. I cannot think of a better person for the roles they played in my life. I am forever thankful.

Finally, I would like to thank my family for demonstrating their unwavering support throughout my studies. Everything was through these people's support and constant guidance in my life. And The God.

CONTENTS

1. Introduction

We live in a world extremely dependent on internet-based service, from as little as ordering food from a restaurant table to buying a house, brewing coffee at the beginning of our day to switch lights at night. Reliance on the internet has made our lives so much so that we can't conceive a day without it. Services like locating a lost device or staying up to date on our daily schedule through our devices in the office or at home are indispensable. The point is the dependency on web services and web services depending on each other to be up and running all the time has become an essential part of our life. An inter overlayed web service dependency is a service depending on other services to be highly available. For example, a service that identifies a person digitally linked to a person's identification number serves as a signature for bank transactions, contracts, or digital identification. Restricting such services stating maintenance, bug fix, or an update will result in financial losses in the private and public sectors, including the health and banking sectors.

The beginning of the 21st century marked an era where everyone depends on one or the other interconnected networks using standardized communication protocols-based services. The exponential growth in mobile apps, web-based services, and less expensive cloud services has already propelled big tech industries to migrate to cloud computing. The cloud provider is responsible for the security and maintenance of the cloud servers. The ease of managing, scaling, continuous deployment, Continuous Integration, and pay-as-you-go services make cloud computing the best choice for customers and developers [2]. Cloud computing is the new hot topic for developers because of the high availability, elasticity, and automation of services through cloud-native tools [2]. As more and more use internet-based services, the availability of the service makes the customer confident about the service provider. The server going offline for just a few minutes makes the company lose huge financial benefits, instigates companies to invest, and improves their service to avoid offline. Companies can achieve zero-service downtimes with automation of the deployment practice, like Continuous Integration and deployment pipeline. Hence, the Microservice Architecture, where the application is a collection of loosely coupled services communicating through API's (application programming interface) refer to Figure 1 [3]. It fosters quality, speed of application development, and fewer service interruptions during upgrading, bug fix, or new version release [3]. Therefore, developers and companies are moving to Microservice rather than developing the complete system at a time to produce a better product in less time. Furthermore, testing, Integration, and deployment through Microservice have become easy, and developers can concentrate on quality [2]. Because microservice is loosely coupled fragments of services, it is entangled with the concept of continuous development and continuous delivery. Achieving continuous development and delivery presents us with three processes designed to reduce or mitigate service interruptions during the development, upgrade, and bugfix process. The automatic deployment processes are Blue-green deployment, canary deployment, and rolling deployment. This paper focuses on evaluating the efficiency and efficacy of three automatic deployment processes in a software development life cycle for a web application. This research will also reduce the

research time spent to better understand different deployment processes and make the zero-downtime deployment process well known to the reader and help make well-informed decisions to deploy without service interruption.

1.1 Problem Statement and Hypothesis

The problem discussed in this paper is the zero-service downtime during the deployment process in the software development life cycle. Furthermore, this paper will also discuss the performance and compare the efficacy and efficiency of the cloud computing deployment process. The hypothesis we test in this thesis is whether a zero downtime of service is possible when updating a web application in cloud architecture using deployment strategies and whether one deployment strategy performs better than others.

1.2 Aim and Objective

This thesis investigates and evaluates the possibility of zero-service interrupts during the deployment of a new service, upgrading service in the deployment process, and producing the efficacy and efficiency of the deployment process that produces zero-service interrupts for updating an application.

Objective 1: SLR to find deployment strategies to achieve service deployment without service interruption or Zero downtime of services during new deployment.

Objective 2: Experiment to check whether all strategies updated their service with high availability of the services (No service downtime).

Objective 3: Compare deployment strategies time to produce efficacy and efficiency of the deployment process for an application.

1.3 Research Question

RQ1: How to deploy a new version of a web application in cloud architecture without any service downtime?
Motivation: The ambiguity in producing zero service downtime sometime requires further research on whether it makes zero service interruptions or downtime achievable. Preliminary research shows that in some cases, the deployment strategy produces results close to Zero downtime deployment time but fails to switch the network node quickly, and service delay occurs. In the case of another deployment process, not all case produces Zero downtime deployment time. Hence requires further investigation. This question will be answered by a Systematic Literature Review (SLR) for finding the strategies for deploying an application without service downtime and testing the strategies if they work by experimentation.

RQ2: How much does deployment strategy differ in deployment time when deploying a new version of a web application in cloud architecture?

Motivation: In the case of zero service interrupt deployment, further investigation needs to contemplate which deployment completes in the shortest time and the factors that affect the process. An experiment is conducted to find the best-performing strategy.

1.4 Ethical, societal, and sustainability aspects

The Developers, various enterprises, and open-source communities will benefit from this research by understanding whether zero service downtime achievable during the service deployment or service updates. This research will also reduce the research time spent to better understand different deployment processes and make the zero-downtime deployment process well known to the reader and help make well-informed decisions to deploy without service interruption.

1.5 Outline

This section describes how this document is structured. The first chapter presents a brief introduction to the relevance of the presented work in the real world. It also consists of aims, objectives, research questions, and problem statements. The second chapter presents the background of this study and all concepts and terms related to this study, like Microservices, the need for deployment strategy, and the tools used. The third chapter presents previous research in the field of deployment strategies. The fourth chapter consists of research methodologies selected to answer the research questions. The fifth chapter consists of the results and analysis of the experiment conducted. The sixth chapter is about discussing the results and answers to the research question. In the seventh chapter, the Discussions, Conclusions, and Future scope of this research have been explained rather elaboratively towards the end of this document.

2 Background

This section presents the concepts addressed in this thesis and helps understand the theme and approach followed in this thesis. The concepts related to cloud computing, continuous Integration, continuous deployment, and continuous delivery are presented here.

Cloud computing is access to infrastructure via the internet, like networking, storage, analytics, databases, Software, servers, and intelligence, without installing and maintaining them physically [13],[14]. There are three subgroups in cloud computing infrastructure as a service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) Services provided by commercial cloud services providers. The first commercial IaaS was Amazon Web Service [2], Consequently making commercial container orchestration through the internet very easy to access and use. The primary benefit of using cloud services is lower cost, Protection from data loss, higher availability, lower latency, improved performance, comprehensive security, and scalability [13].

According to [1], there are four stages in a cloud-based software development life cycle. The first is Container orchestration on the cloud with specified specifications; it is a virtual environment with specified resources. Resource specifications like Memory required, CPU, and network settings. It is done only once on a container, and the end consumer can deploy any number of workers. The second is keeping them running; if a failure occurs or a worker is down, spawning a new worker to manage the workload. The third is delivering or deploying the system. Fourth, monitoring the health of the worker nodes and clusters for failure or errors. In delivering or deploying the system, continuous Integration and continuous deployment (CI/CD) pipeline can also be utilized for Microservice Architecture (application as a collection of loosely coupled services) rather than a monolith software architecture. In Microservice Architecture, service delivery is shorter; the product is available quickly and with fewer service interruptions [4],[15]. The Software life cycle maybe a week or months-long [1]. When developing a huge Software with multiple Microservice, updating an existing service or deploying a new service will result in service interruption [1],[4],[17]. Each deployment will cause significant service downtime due to switching from old service to new [1],[4],[18]. The developer must adapt to a process that can handle less or no downtime to avoid downtime. So, the consumers or end-users will not experience any downtime of services, and the customer will retain valuable revenue. Emerging companies demand changes in their product rapidly and continuously to be the best in their sector [8]. The initial literature revive reveals zero-service downtime can be achieved in three ways but fail to produce zero downtime results every time [4]. The three processes for zero downtime deployment are Blue-Green deployment, Canary deployment, and Rolling Deployment [3],[4]. This paper aims to address the zero-service downtime deployment process efficacy and efficiency for a web-based application. A literature review reveals no similar research or study conducted.

2.1 Virtualization.

There are two types of virtualization hardware-level virtualization and operating system-level virtualization. Virtualization means the abstraction of computer resources. The benefit of virtualization is the flexible allocation of resources and the scaling of resources. Virtualization involves setting virtual machine software known as Hypervisor or Virtual Machine Monitor into the hardware component. The hypervisor controls the processor, Memory, and other components. It can run several Operating systems (OS) without source code on the same resources. The OS running on the virtual machine will seem to have its process with multiple cores, Memory, and network hub. Generally, the resources are allocated by the system admin. There can be two types of Hypervisors, one which is on an OS inside a Virtual Machine (VM) and the other on bare metal [2].

2.2 Containerization

Containerization is a virtualization technology enabling the deployment of applications on the cloud. A container is an environment where an application or a component (microservice) and all the library dependencies can be installed, which are the basic configuration for an application. It offers a higher level of abstraction for application lifecycle management, starting/stopping updating, and seamlessly releasing a new version of a containerized application. One example is Docker [4].

2.2.1 Docker

Docker is a project designed for deploying an application inside a portable container. It is an open-source project that provides a systematic and faster way to deploy containers in process isolation: CPU, Memory, I/O, network, etc. Docker allows running multiple containers simultaneously. The Docker container is deployed using a Docker image, an executable containing information and specification of the Docker container. Dockerfile contains specifications to configure the Docker image. Containers related to the same image share the image. Docker container contains multiple layers, one layer that can be read and written, another layer that is an underlying layer that can only be read. For example, Nginx image can be on an underlaying base UbuntuOS image [2].

2.3 Kubernetes

Kubernetes is an open-source tool for orchestrating, controlling, and managing containerized application deployments [16]. It simplifies Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). Moreover, it supports orchestrating computing, networking, and disk storage. Furthermore, it supports deploying incredibly diverse workloads for various types of applications. It also has an essential feature called canary-deployment. It is a feature that lets the deployment rollback before deploying the application entirely overall working nodes. Kubernetes is independent of the infrastructure it is running on. It is perfect for microservices automatic deployments [16]. It can scale containerized workloads, manage services, and automate container deployments in their clusters [2].

2.4 Cloud Computing (Open stack)

Open stack – It is an open-source software capable of controlling and managing data center's resources. It controls computing nodes, storage nodes, controllers, and networking. It is supported and runs by different organizations. It is a heterogeneous system that can run in different environments. It is composed of various REST (Representational State Transfer) services [2]. It is an infrastructure as a service (IaaS) provider. OpenStack has different tools or services which each concentrate on different services. Services like NOVA (Compute Service), ZUN (Container Service), MANILA (Shared file system) for storage are some services provided by Open Stack 3 open-source community [1]. Open Stack is an open-source cloud orchestration software that can be used across all platforms [2].

2.5 Prometheus

It is a monitoring and alerting tool that uses several metrics, including the number of requests, memory usage, and request durations [2].

2.6 Google Lighthouse

Google Lighthouse is an automated tool for measuring the condition of any web page. It can check the performance, accessibility, and search engine of a web page. It can be run in Chrome, from the command line, or as a Node module. Giving the Lighthouse a web link to test, it runs a series of tests against the web page and reports on how well the page did [2].

2.7 Microservices

Application is no single cohesive unit. It is a distribution of individual entities that work and communicates with each other. It is one way to build software applications by splitting them into multiple independent services. This way, developers can concentrate on the Quality of Services (QoS), Quality of Processes (QoP), and development speed of the application [3]. It also improves modularity, making applications simpler and more resilient. Because of their lightweight nature,

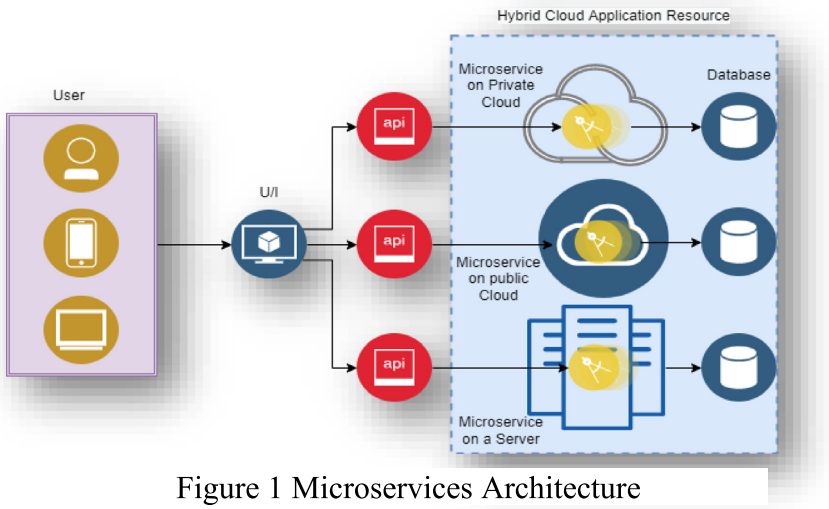

Figure 1 Microservices Architecture

microservices are based on container virtualization technology [4]. As shown in figure 1 the user communicates with the UI. The UI has the information to which API it needs to communicate to get desired information back, communicates with the API and the API fetches the information. Each microservice can be geographically situated in different places and with different types of servers(cloud or bare metal), yet API seamlessly communicates with the services and completes the desired task.

2.8 Automatic Deployment Strategies

2.8 .1 Blue-Green Deployment

Blue-green deployment technology provides agility for continuous delivery with near-zero or zero downtime. Thus, reducing the risk of service unavailability. This technology uses two different hosting services in two different environments. The aim is to shift incoming traffic from the current service version to the new version. After deploying the new version, before the old version is made offline, the new version is tested before traffic diversion. According to [5], the traffic is directed toward the new version after the testing and deployment, saving time and probably

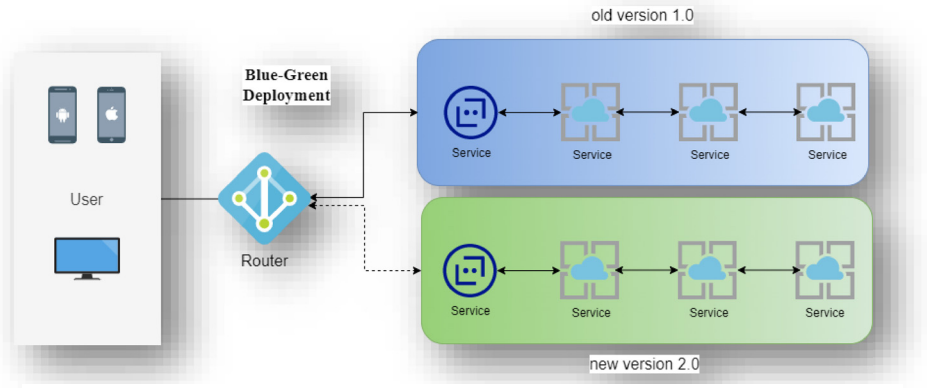

Figure 2: Blue-Green Deployment

achieving zero-downtime deployments [4]. As shown in figure 2 Blue version is the old version and the green version is the new version. when the resources are deployed the traffic is switched between the new and old versions just by routing the traffic.

2.8.2 Canary Deployment

The Canary Deployment model is similar to the blue-green model, but routing traffic is done in a phased model. When a new version is to be released a new resource with a new version running is deployed as it is shown in figure 3 below. For testing, the new version, 5 % of the traffic is routed initially. If the version doesn't crash or fail the whole 100 % of traffic is diverted to the new version. This can be customized depending on the application's needs. It is implemented by releasing it to specific worker nodes. later rolled out to all working nodes once the blue version is approved [4].

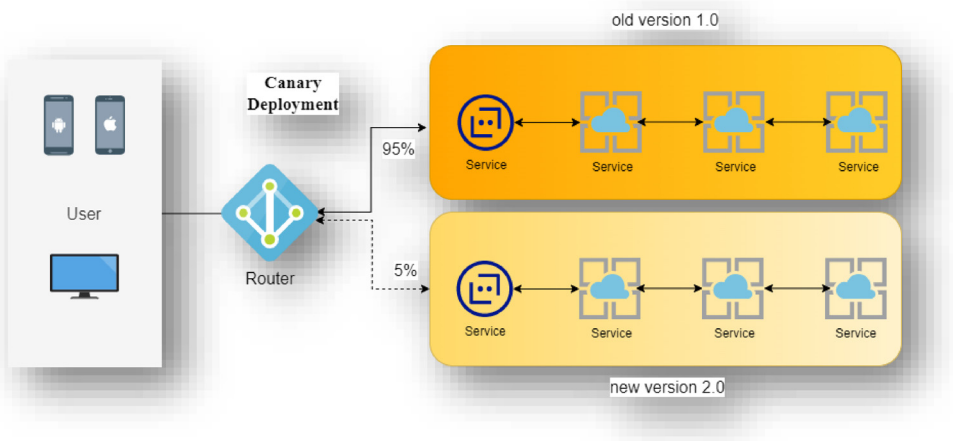

Figure 3: Canary Deployment

2.8.3 Rolling deployment

Unlike other deployment strategies rolling deployment replaces the application running on the same worker node one by one. The newer version of the application is deployed on the same node as the old node or the same resources, and if the new deployment fails, the update is rolled back to the older version. Rolling deployments work well only when the organizations have enough spare capacity to roll out the newer versions without decreasing the performance. If a significant patch needs to be delivered simultaneously to all users, it may not be suitable [3],[4].

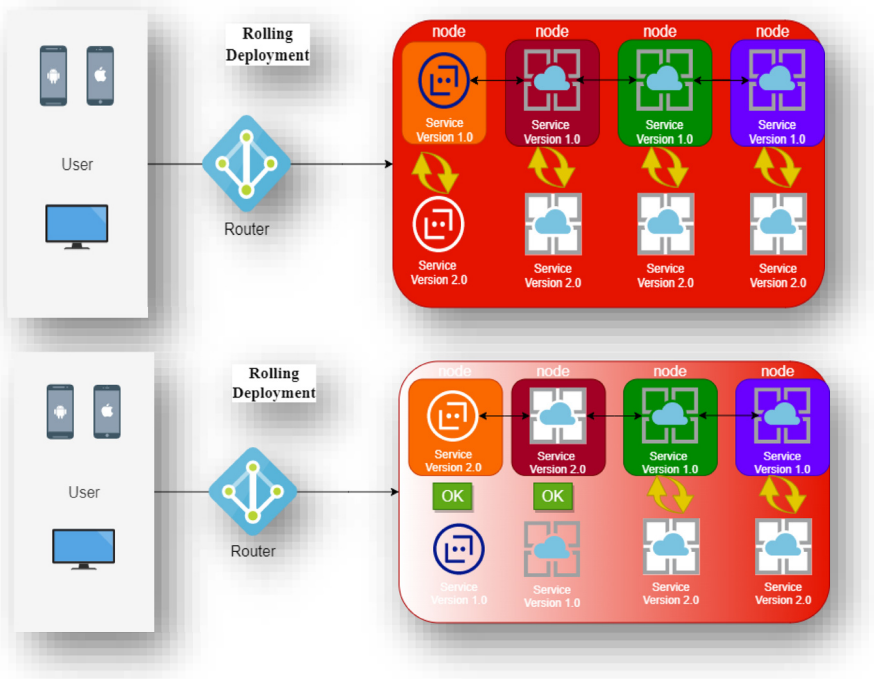

Figure 4: Rolling Deployment

2.9 Zero-downtime deployment

When a new version of an application is deployed without affecting the current services provided by the application, is called a zero-downtime deployment. Any

strategy that allows for deploying new application features and bug fixes without service interruption (no or zero downtime) is called a Zero-downtime deployment strategy. This type of deployment is used for rapid deployments, like Continuous Deployment. Continues Deployment is a part of Continuous Delivery, which is a part of Continuous Integration. Continues Integration is an approach where updates are constantly written into the code and tested automatically. Continues Delivery is a part of Continuous Integration to ensure software reliability. Continues Deployment is the final automation step in Continues Delivery, where a change in application code is automatically deployed after testing [1]. In a cloud environment, the Blue-Green deployment is agile means of DevOps Continuous Delivery with zero-Downtime [17],[18].

2.10 Kubernetes architecture for application deployment

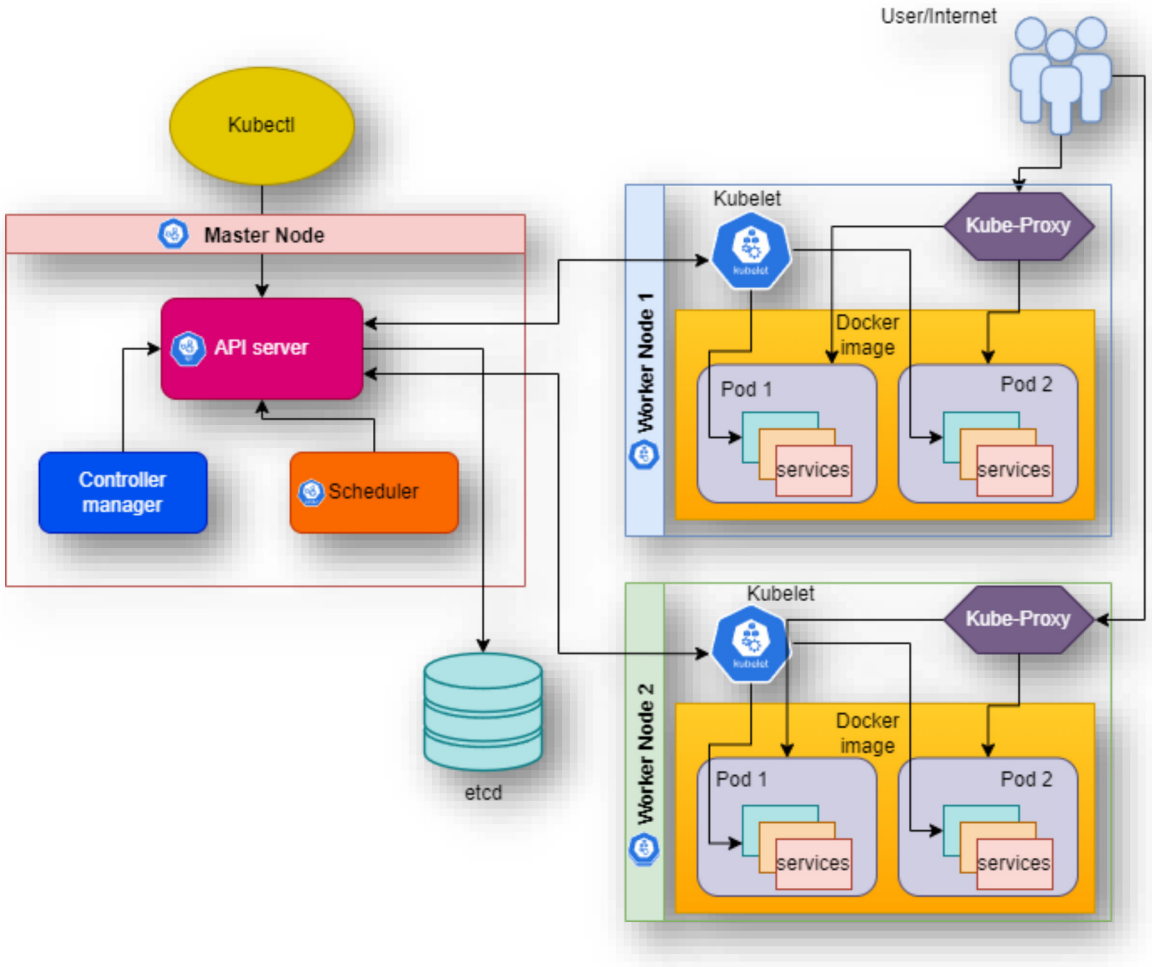

Figure 5 Kubernetes deployment architecture

The above Figure 5 a high-level Kubernetes architecture which is crucial to understand the functionality of deploying an application onto a pod. It consists of a master and any number of worker or slave nodes. The master node functions as a

controlling node and manages every aspect of the worker node setting changes or functionality. The master node consists of kube-apiserver, an etcd storage, kube and cloud controller-manager, and a kube-scheduler. Slave or worker node components are kubelet and kube-proxy which work on top of Docker [16].
Below mentioned are the components and functionalities of the master node components.

Kubectl: Kubectl is a command line interface that interacts with kube-apiserver to send commands to the master node. All commands are converted into an API call [16].

etcd: It is a distributed key-value storage that stores the Kubernetes cluster data. The data datils such as the number of pods, their state, namespace, API objects, and service discovery details. The data is only accessible through the API server for security. etcd notifies the cluster about configuration changes with the help of Schedulers [16].

Apiserver: It is a management system that receives REST requests for changes that need to be made to pods, services, number of replication sets/controllers, and other information which also serves as a frontend to the cluster. API server is the only system that communicates with the etcd, ensures data is stored in etcd, and is in sync with the service details in the deployed pods [16].

kube-controller-manager: It is responsible for processes in the background. To controller control, the number of replications in a pod, and services running in a pods container. It performs routine checks for changes. When a change in a service configuration occurs like replacing the Docker image and parameters in the configuration yaml. The controller observes the change and makes changes to the existing state of the pods [16].

cloud-controller-manager: It is responsible for managing controller processes with cloud providers. Any changes like set-up routers, load balancers, or volumes in the cloud infrastructure all are managed by the cloud-controller-manager [16].

scheduler: It schedules the pods on the various nodes based on resource utilization needs. It monitors the requirements and schedules the changes. For example, if the application needs 1GB of memory and 2 CPU cores, then the pods for that application will be scheduled on a node with at least those resources. The scheduler runs each time there is a need to schedule pods. It contains information on the total resources available and resources allocated to each node [16].

Below mentioned are the components and functionalities of the worker node components.

Kubelet: It is the main service running on the worker node, it frequently receives new or modified pod information from kube-apiserver and ensures that pods and

their containers are according to the information and also whether they are healthy and running. This component reports to the master on the health of the host [16].

kube-proxy: It is a proxy service running on all worker nodes to manage individual host subnets and expose services to the user or the internet. It is responsible for request forwarding to pods/containers and isolated networks in a cluster [16].

Pod: It is a collection of containers or services that are controlled and considered as a single application. A pod encapsulates application containers, their storage, network ID, and information on how to run the containers [16].

3 Related Work

This section presents the same (or similar) problem or uses the same approach to solve a different problem. The papers selected are the most recent.

In [1], a Controlled experiment of deployment automation in a high-availability architecture is presented. This experiment tests whether the automation process updates the existing version of the application without affecting the system's availability [14],[25]. The two automatic deployment systems tested are Blue-Green deployment and canary deployment. They are tested for low, medium, and high traffic with successful and failed deployments with a margin error rate of 0.00 - 0.05 as a successful success error rate of > 0.05 else failure. The author discusses no significant error rate change during normal operations or successful deployments. However, there is a slight increase in the error rate during medium to high traffic levels during successful deployment compared to service during normal operations [13]. The author concludes, it is possible to deploy a successful deployment with an automation deployment without affecting availability during higher traffic levels if provided enough computation resources without compromising the high availability [24].

In [3] the author presents the fast re-deployment of Microservices with Blue-Green deployment strategy for the Osmotic Computing paradigm, theorized in 2016 as Integration between a centralized Cloud layer and Edge and/or IoT layers. An osmotic Computing should be highly horizontally/vertically scalable, 24 hours 24 available, fault-tolerant, and secure. Here an osmotic system is designed with tolls using Docker for developing and shipping, Kubernetes, Agento, MongoDB. The Experiment is conducted in two different stages first is Kubernetes manifests split and Kubernetes manifests store and respectively manifest retrieve and recompose in MongoDB [20]. Second, evaluating the Pod's deployment and destruction time on IoT and Cloud nodes [12],[26]. The experiment concludes that the Cloud deployment and destruction scenarios are 20 seconds and 11 seconds faster than the IoT scenario with the Blue-Green deployment strategy.

In [4], according to the author, Blue-Green deployment can achieve zero downtime in the cloud infrastructure in multiple ways. Opting the best way to perform a Blue-Green deployment is essential [23]. It can be implemented by swapping DNS (Domain Name System) routing, Load Balancer swapping, and application swapping. In DNS-based swapping technique, the DNS configuration is changed to swap the traffic between the old and new versions. In load balancing swapping technique, DNS is kept constant, but the load balancer is pointed to the newer version. Similarly, DNS and load balancer are kept unchanged in the application swap technique, but the Docker image (preconfigured server environments) is changed [15]. The author observed that swapping time and roll back time are higher in the DNS-based Blue-Green deployment technique due to the DNS time to live parameter with a swapping time of 243, which is higher than other deployments, approximately 18 seconds gain compared to load balancer swapping deployments, which was due to ALB (application load balancer) health check. The load balancer technique performs better with a swapping time of 186, 57 seconds faster than the DNS-based Blue Green technique. The swapping time of

201 seconds was observed in the application swap technique, which is 15 seconds slower than the load balancer swap technique [27]. However, the adjustment to the ALB configuration needed to be more frequent in the load balancer swap technique for scaling up and down blue and green versions. The author concludes that the proper deployment needs to be used in the cloud platform based on the application's functionality.

In [5], an evaluation of automated service discovery, dynamic routing, and automated application deployment based on the Blue-Green deployment strategy is conducted. The target of the study is to enable service updates with zero maintenance window with one of the Blue-Green deployments. Firstly, they set up the environment with a simple Node.js API server which returns the service version when recalled; two identical environments with two service instances, both with different version configuration information, are deployed. Secondly, they deployed a tester that sends a Curl request every second and records its response. Request time and response time is written to a file from the switches and stored in the file. Third, they use a switcher to switch traffic between blue and green services. This paper concludes that the DNS-based solution performed worst when switching traffic with a switching time of 187 seconds [28],[16]. Even though the AWS load balancer and auto-scaling group solution were cost-effective, their switching time was 195. It took 165-170 seconds more switching time than the service discovery-based strategy. However, it took a long time to initialize a new instance [29]. The service Discovery solution and Cloud Foundry Remapping Router solution switching times were 22,9 and 25 seconds. This paper concludes that not all service produces the desired result, but a Service discovery-based strategy, out of four methods, can be used due to the stability of the deployment method.

In [6], tools are compared for Continuous Integration (CI) and Continues Delivery (CD) to achieve zero-downtime deployment of services. A performance evaluation of Gitlab and Jenkins CI by Dockerized Microservices on the AWS cloud platform is presented [35]. According to the author, Jenkins is easy to use because it supports different plugins, but as the application grows, the plugins increase, and it is harder to manage Jenkins [17]. However, in the case of Gitlab is configured in a simple YML file, so it is easy to manage even when the Microservice project base increase. Select the tool based on the five projects if it is simple to go for Gitlab, but if it is an enterprise project, this paper suggests using Jenkins as a CI tool for the deployment of services [30].

In [7] author has presented the design and implementation of a continuous integration scheme. They discuss the design process and Architecture Design for Continuous Integrated Delivery System Network Topology. Components like SVN Resource Repository, Jenkins, and Ansible are analyzed to use in the integration process [31]. The functionalities of the continuous integration delivery system are Version Control Component, Engineering Builds Components, Quality of the Code Review Component, System Service Component, Information Display Service Component, CI Master Control Services Component, Automatic Version Service Component. After design and analysis, the paper concludes that Continuous integration systems help shorten development time and improve software quality by an average of 25 min with Ansible and Jenkins.

In [11] author presents an automated CI/CD pipeline for deploying a Java-based web application in AWS. The author used the rolling-deployment process as a deploying strategy. The first step toward the automatic deployment process is to create a CI pipeline [32]. They used the open-source tool, Jenkins, as a CI tool. After installing the Jenkins server on the AWS EC2 instance, installing GitHub and Apache Maven plugins, and ensuring the interoperability of servers. After that Docker image called Tomcat8 was pushed to Docker Hub [18]. A dedicated Kubernetes cluster and monitoring tool CloudWatch was also set up [33]. Even though most pipelines use Jenkins for CI and CD processes, in this case, Jenkins opted for CI, and Ansible is for CD due to the ability of Ansible to execute commands on the target host directly. Also, due to the advantage of scalability and revert the configuration in case of failure. When a code change in GitHub is detected by Jenkins clones the repository and compiled with the help of Maven in a .war file [19]. Then the .war file to transferred from Jenkins to Ansible via SSH and copied onto the Tomcat image, and a new image is created. Thus, finishing the CI process. In the CD process, Ansible executes the playbook by creating a Kubernetes cluster; then, the Kubernetes cluster automatically deploys all the essential services mentioned according to the YAML file [34]. The concludes that the zero-service downtime was achieved with 37.6 seconds of deployment time as soon as a change was detected in the source code of the Java-based web application.

To summarize, [1] the author presented a comparison of two deployment processes Blue-Green deployment and canary deployment but did not address their deployment time and document their switching time from old to new which is important to understand their performance capability. The author did not compare the deployment process at all which also needs to understand the process. And also, in [3] author discussed the fast deployment of microservices IoT environment, which yielded 11 seconds faster than others. Using a deployment technique in pre-built in the Kubernetes application. But this [3] research did not measure normal Blue-Green deployment procedure switching time. The author in [3] is looking into the fast redeployment of the services but failed to compare it with multiple deployment strategies which may have been faster than just one blue-green deployment strategy. In [4] the author has looked into Blue Green deployment only similar to [3] where the author tries different ways of blue-green deployment itself to fast redeployment of services where the author tires DNS (Domain Name System) routing, Load Balancer swapping, and application swapping. The author did not also look into different deployment types which can be faster than Blue-Green deployment. An investigation of other deployment strategies is needed to understand all the deployment strategy's performance and where they can be used. Furthermore, in [5], the author discusses two types of DNS-based and service discovery-based Blue-Green deployment that yield different results, one performing better than the other. This paper also did not look into the other deployment types. In article [6], tools to automize deployment are compared, and in [7] compare, CI tools for faster integration into the CI pipeline. The above article shows the necessity of comparing other deployment strategies to understand and compare. Based on this literature review, we notice that there is enough research on the deployment process, but there is no research comparison of the deployment process for efficiency.

4 Methodology

In this section, we discuss the approach and methodology used. Furthermore, the following section 4.5 will present the experiment process. In this section SLR and experiment are conducted.

4.1 Systematic Literature Review

A systematic literature review is to identify, evaluate and interpret all research relevant to particular research. To answer research question one, it is crucial to understand the preexisting method for updating a new application. For this reason, an SLR is conducted related to the topic.

4.1.1 Motivation

The purpose of conducting a systematic literature review is to identify an existing framework and experimental design for deploying an application without causing any service drop by deploying strategies and understating their functionality.

4.1.2 Search Sources

The sources are the sites which the primary studies are selected from. The online database should have a search option using search as well as filters such as years, language, study field, and others. The source of SLR was the following online database: IEEE Xplore, BTH summon, Google Scholar, and ScienceDirect. The forward snowballing approach has been used for searching related articles.

4.1.3 Search Strings

Articles and conference papers with the following keyboard in their title or abstract are shortlisted. One of the most critical aspects while conducting a literature review is understanding the classification correctly as it plays a significant role while formulating the search string.

Search String 1: (“Cloud-Native”) AND (“Deployment process” OR “Services Automation”)
Search String 2: (“Cloud-Native”) AND (“Services Automation” OR “Microservices”)
Search String 3: (“Services Automation”) AND (“Microservices”)
Search String 4: (“Zero downtime”) AND (“Deployment Strategies”) OR (“Cloud-Native”).
Search String 5: (“Rapid deployments**”)** AND (“Cloud-Native”) OR (“Services Automation”)
Search String 6: (“Continuous Integration**”)** AND (“Continuous Deployment”) AND (“Pipeline Automation”)

These were versions of the search string used to search the database. For identifying the relevant literature an in-depth reading of the abstract and the final paragraph of the introduction section was done. If the understanding was vague, the conclusion and discussion were given a brief read.

4.1.4 Inclusion and Exclusion Criteria

Inclusion Criteria:
1. Articles should be in the English language.
2. Article should be published in the year 2010- 2022.
3. Articles in books, magazines, conferences, and journals.
4. Articles that associate with the problem domain.

Exclusion Criteria:
1. Papers regarding deploying Docker, New application, Kubernetes cluster.
2. Papers only mention service automation and load balancing.
3. Article where complete text is not available.
4. Abstracts and PPTs are not included.
5. Articles that are not in English.

Applying inclusion and exclusion criteria, papers were narrowed down. Next section presents the Systematic literature review.

4.1.5 SLR Results Review

This section shows the results of the SLR. The documents were read through the Abstract, Introduction, Method, and Results after the inclusion and exclusions were applied. All relevant articles are shortlisted. The SLR results are also mentioned in the result section.

Table 1 SLR Results

No	Author / Title	Findings
1	A. Buzachis et al. [3] Towards Osmotic Computing: a Blue-Green Strategy for the Fast Re-Deployment of Microservices	This research looks into the fast re-deployment of microservice that exist between IoT layers and the Edgy cloud layer. The strategy used in this research is the Blue Green strategy. The tools used are Kubernetes and Docker. The aspects of resource examined are network load, traffic, CPU, and Memory usage.
2	B. Yang et al. [5] Service Discovery-Based Blue-Green Deployment Technique in Cloud Native Environments	In this paper, the author assesses the continued delivery methodology i.e Blue-Green deployment for zero maintenance window. The author proposed an optimized blue-green deployment based on automated services discovery and dynamic routing. The strategy used in this research is the Blue Green strategy with load balancer swap and the Blue Green

		strategy with DNS discovery. Tools used AWS EC2, Zuul servers, and Eureka.
3	J. J. Carroll et al. [41] Preproduction Deploys: Cloud-Native Integration Testing	In this paper, the author discusses the testing of multiple microservice-based applications in combination with canary and blue-green deployment together in a preproduction environment in other words a semi-active environment for testing only. Where blue-green versions are deployed and the first 1% of traffic later 10% of traffic is tested. Strategies used are canary deployment and blue-green deployment. Services tested are Gateway microservice, Cloud microservice, and JavaScript application.
4	Khaled Jendi et al. [2]Evaluation and Improvement of Application Deployment in Hybrid Edge Cloud Environment	The author evaluates and improves different deployments in edgy cloud performance. This also depends on achieving efficient usage of cloud resources, reducing latency, scalability, replication, and rolling upgrade, and also load balancing between data nodes, high availability, and measuring zero downtime for deployed applications. Strategies used: Rolling deployment and canary deployment, Tools used: Docker, Kubernetes, Spinnaker, Google Lighthouse, Prometheus, Azure, OpenStack. Metrics observed or measured or compared: CPU, Memory, Request duration, Message Delivery, and Request Interruption.
5	Nilsson, Axel, et al. [1] Zero-Downtime Deployment in a High Availability Architecture.	In this project, an automation system is proposed to allow deployments on a high availability architecture while ensuring Zero Downtime. The created automation system is tested in an experiment to check if it produces what it is made for which is the high availability of services. Strategies used: Blue Green deployment and Canary deployment. Tools Used: Ansible, HAProxy, NodeJS, bind9, Java. Metrics measured or observed: Traffic error rate, Request Response time.
6	Chaitanya K. Rudrabhatla et al. [4] Comparison of zero downtime-based deployment techniques in public cloud infrastructure	This research presents Blue Green, Canary, and Rolling deployment techniques in the cloud platforms. Also, perform simulation of the Blue-Green deployment strategies using DNS routing swap versus Load Balancer swap versus newer image switch techniques.

		Strategies used: Blue-green deployment, canary deployment, and rolling deployment. Tools used: AWS Ec2, Kubernetes, Docker.

Reason for not selecting other deployments strategies:

Other deployment processes from LR which have not been selected for experimentation are A/B testing deployment, Recreate Deployment, Ramped Deployment, and Shadow Deployment [42],[43],[44]. The reason for not selecting the A/B testing deployment is it has many similarities with Blue Green and Canary deployment according to [42] and it involves testing two or more variations of an application for failure which is not related to our research area. As we are not testing the application fault but application deployment without service dropping[43],[44].

The reason for not selecting Recreate Deployment is that in this deployment strategy, the system admin shuts down the old version of the application completely, deploys the new version, and then reboots the whole system. Which causes service downtime [44]. Hence Recreate Deployment is not a Zero downtime service deployment strategy.

The reason for not selecting Ramped Deployment is Ramped deployment strategy is: the Ramped deployment strategy makes its application update by replacing individual instances of the old application version with the new instances from the new application version one at a time just like the rolling upgrade deployment strategy [44]. Since we have already selected rolling deployment we are not testing this deployment.

The reason for not selecting Shadow Deployment is the similarity to canary deployment. Because the admin deploys a new version while the old version is online. Initially, the traffic is not diverted but a test fork traffic is diverted and tested for stability. If the test is successful then the whole traffic is diverted. Since it is the same as the canary deployment and we are testing the Canary deployment we are skipping this deployment.

4.2 Alternatives Rejection Criteria:

Survey: The survey can be used for market research and polls for opinions but does not help in selecting a model. Surveys are done when models and tools are already being used but selecting them for a particular reason is solely dependent on users' choice [40]. In this paper, since the performance of deployment cannot be defined by the sample; hence this method is rejected.

Case Study: A single phenomenon is investigated in a case study in a particular time frame. The experiment is to sample the variables but in the case of the case study select the variables in a particular situation. It is usually used for the evaluation of software models or tools.

Hence, the above two research methods are rejected. A literature review with experimentation is selected as a research method for this research. The following explains the process of Research Approaches.

4.3 Research Approaches

To achieve the experiment and justify the purpose of the thesis, we must apply reason-based, analytical, and qualitative research methods.

4.3.1 Applied

The goal of the research is to find the efficiency of the solutions for the deployment problem. Furthermore, the applied research method will evaluate the solution found in the project.

4.3.2 Qualitative

Software deployment is a behavioral and experimental science that is concerned with deployment quality. In this project, we apply a variety of test cases to determine the quality of the suggested solution for continuous delivery framework deployment automation and distribution. The qualitative approach is used in this project because it is not based on a measured quantity. The qualitative method uncovers the motives of deployment automation to solve the deployment automation. It is done so using various ways. The qualitative technique allows researchers to investigate various deployment mechanisms and compare them. It also aids in the improvement of present procedures, resulting in a higher-quality solution.

4.3.3 Analytical Approaches and Empirical Approaches

Because deployment automation necessitates control and manipulation of background variables to provide the capability to assess and evaluate the results [5], this project employs both empirical and analytical methodologies. The analytical approach presents this research to analyze and critically evaluate various deployment approaches. The analytical research attempts to identify underlying relationships among deployment procedures and unite them to reach the best approach that meets the project's aim and purpose. It also aids in understanding and explaining the deployment automation strategy designed for this project.

It helps to address the question of how we can simplify the deployment process while keeping other important parts of deployment in place, such as load balancing, auto-scaling, continuous delivery, and multi-cloud infrastructure support. The empirical approach aims to experiment and evaluate the outcome of deployment automation and how specific background variables can influence the outcome of deployment simplification, resilience, dependability, and performance.

4.4 Research Process

The research process is made up of a series of actions and activities that are required for scientific study. These phases are sequential and interdependent. This implies that the outcome of one stage is crucial to the subsequent phases. The following are the steps involved: Specify the problem, do a thorough assessment of theory and existing research, develop a hypothesis, define the research, gather data, and analyze

the results. This procedure can be repeated until the researchers achieve satisfactory conditions in accordance with project objectives.

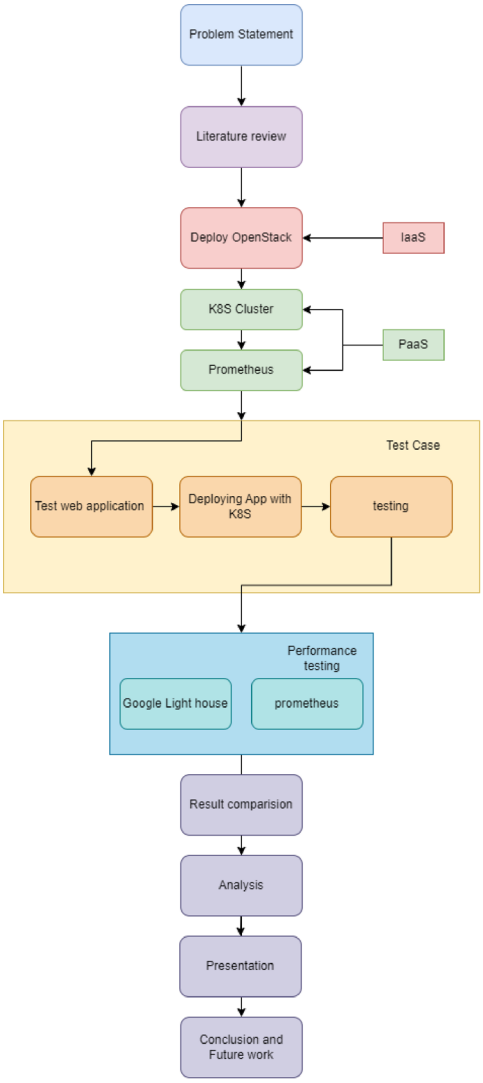

Figure 6 Research Process

4.5 Experiment

This section presents an automatic deployment test case performed in this thesis. In this experiment part, the selected three approaches from SLR are tested and the performance of the three approaches is evaluated. This method answers the question RQ2 and identifies the optimal approach for deploying a new version of a web application in cloud architecture. Each section is a step taken in this study. Setting up the environment, and configuration of tools to recognize and establish communication with each other is the first step we are going to take in this section. The selected strategies from SLR are Blue-Green deployment, Canary deployment, and Rolling deployment.

4.5.1 Variables and Hardware Environment

Independent variables: The deployment strategies and availability of services during the deployment of a new version of the application is the independent variable. Does deployment strategy produce zero downtime or not is the independent value.

Dependent Variables: The performance values/deployment time of each deployment type is the dependent variables as they depend on the deployment strategies to produce a zero downtime upgrade for an application.

Hardware Environment
The hardware specification of the computer system utilized to implement the experiment.

Operating system : Windows 10 Education
System Processor : AMD Ryzen 5 2500U
Base Clock Speed : 2.00 GHz
Number of cores : 8
RAM : 16 GB
System Type : 64-bit operating system, x64-based processor

4.5.2 Cloud Clusters Configuration for Experiment

Table 1 cluster configuration was used in manual and automatic deployment for Blue-Green, Canary, and Rolling deployments. The selected values are the minimum system requirement for the web application. Since our goal is to test the service downtime, I have selected minimum system requirements. All three deployment strategies and manual deployment use a similar cluster configuration for experimental accuracy. The reason for selecting Ubuntu as an operating system is it is a Linux-based operating system that is server-centric and lightweight. The Memory and CPU (Central Processing Unit) selected for the node are the minimum requirements for Ubuntu and other dependencies that are going to be installed on the nodes [2].

Deployment	Automatic/Manuel
Cluster type	Open Stack Cloud cluster
Central Processing Unit	2 Per node
Random Access Memory	4096 MB
Storage	10240 MB
Operating system	Ubuntu 20.04
Traffic Distribution	Octavia

Table 2 Cluster Configuration

4.5.3 Cloud Infrastructure (PaaS) OpenStack

In all three test cases, cloud deployment is the first step. Both steps IaaS and PaaS deployment are common for all three deployments. This step directly comes after problem formulation and extensive literature review. The selected cloud for this step is OpenStack because it is open source and free to manage in our environment, unlike Amazon Web Services, Microsoft Azure, and Google Cloud Platform, which are public cloud providers and also a result of SLR. I used the Yoga version of OpenStack. Since Open Stack is an open-source cloud provider all of its components are distributed individually in which each component's functionality is different. We are going to install a packaged version available as Devstack which contains all individual components of OpenStack. The first step would be to setting up the

Figure 7 Installed server

environment to install the open stack. I am using Virtual Machine (VM) called Oracle VM VirtualBox is a type-2 hypervisor which is an open-source product of Oracle Corp. Virtual box is installed with its extension pack for networking and other I/O drivers. ISO image of Ubuntu Server 20.04 LTS flavor is downloaded from the website. After the necessary file are downloaded from the internet. A VM containing 2 cores Processor, Memory of 8GB, Hard Drive space of 40GB and an extra network port with Host only configuration is setup for later accessing the GUI of open stack Cli. When starting the VM for the first time a bootable ROM is requested by the VM to setup the Operating System.

The below steps are from standard documentation provided by the following tools manufacturers and may have resembling similarities with other sources because the command is almost the same to install and set up the tools.

4.5.3.1 Installing DevStack

1: *Update and Upgrade the System*

log into Ubuntu 20.04 system using SSH protocol and update & upgrade the system. The following command is used to update the system. It is done so that the system is up to date with the latest security and software. Reboot the system.

Command - `apt update -y && apt upgrade -y`

Command - `sudo reboot`

2: *Creating a user "OPENSTACK" with sudo privilege*

It is important to create a new user so that most of the settings like network and directory settings are confined to one user rather than affect all or the root users. Creating a new user called "openstack" with sudo privilege.

Command - `adduser -s /bin/bash -d /opt/openstack -m openstack`

Assigning Sudo privileges to the user with no password requirement. So that when we install the tools and software it won't interrupt the setting process with the admin password for every tool.

Command - `echo "openstack ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/openstack`

3: *Downloading DevStack and installing GIT*

After creating the user 'openstack' switching to the stack user using the following command.

Command - `su – stack`

Installing Git.

Command - `apt install git -y`

Cloning repository as shown.

Command - `git clone https://git.openstack.org/openstack-dev/devstack`

4*: Editing configuration file updating passwords and Host IP*

Changing directory to Devstack directory then Navigating to samples/local.conf configuration file using the following commands.

Command - cd devstack

Command - nano local.conf

Editing the following content in local.conf file then saving and exiting the editor.

```
ADMIN_PASSWORD=2222
DATABASE_PASSWORD=2222
RABBIT_PASSWORD=2222
SERVICE_PASSWORD=2222

HOST_IP=192.168.56.114
```

5: *Installing OpenStack*

The below command will be used to install the open stack. The home page and API access server end points are given in the Figure 8 and Figure 9 below.

Command - ./stack.sh

This Devstack Package include the following features:
OpenStack Dashboard called Horizon, Compute Service: Nova, Image Service: Glance, Network Service: Neutron, Identity Service: Keystone, Block Storage Service: Cinder, Placement API: Placement.

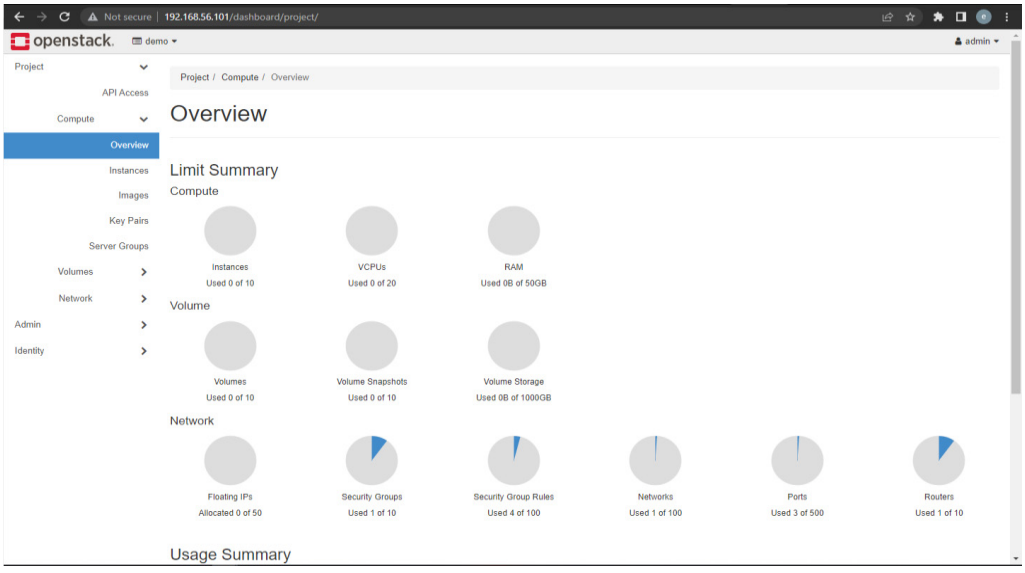

Figure 8 Home Page

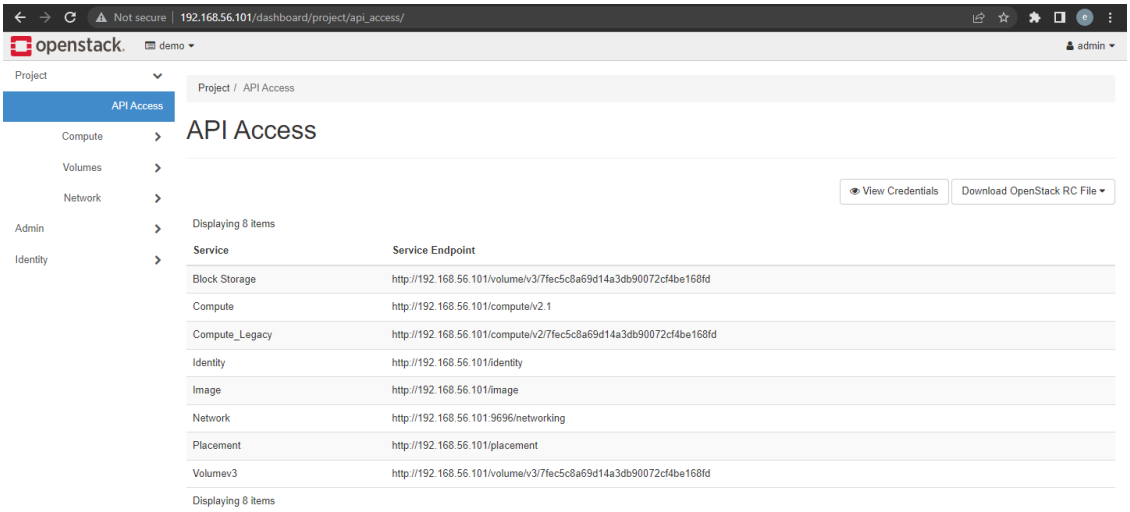

Figure 9 API Access

4.5.4 Installing Docker

Installing the docker according to the docker documentation. These are the five steps provided below.

Step 1: *Installing Docker*

Update the package list the following command is used to update the system. It is done so that the system is up to date with the latest security and software. Reboot the system.

Command - apt-get update
Command - apt-get install -y apt-transport-https

Installing Docker:

Command - apt-get install docker.io

Repeating the same process on all nodes.

Check the installation:

Command - docker–version

Step 2: *Enable Docker, Set Docker to launch at boot:*

Command - systemctl enable docker

Verifying Docker is running sd you can see in the Figure 10 :

Command - systemctl start docker && systemctl status docker

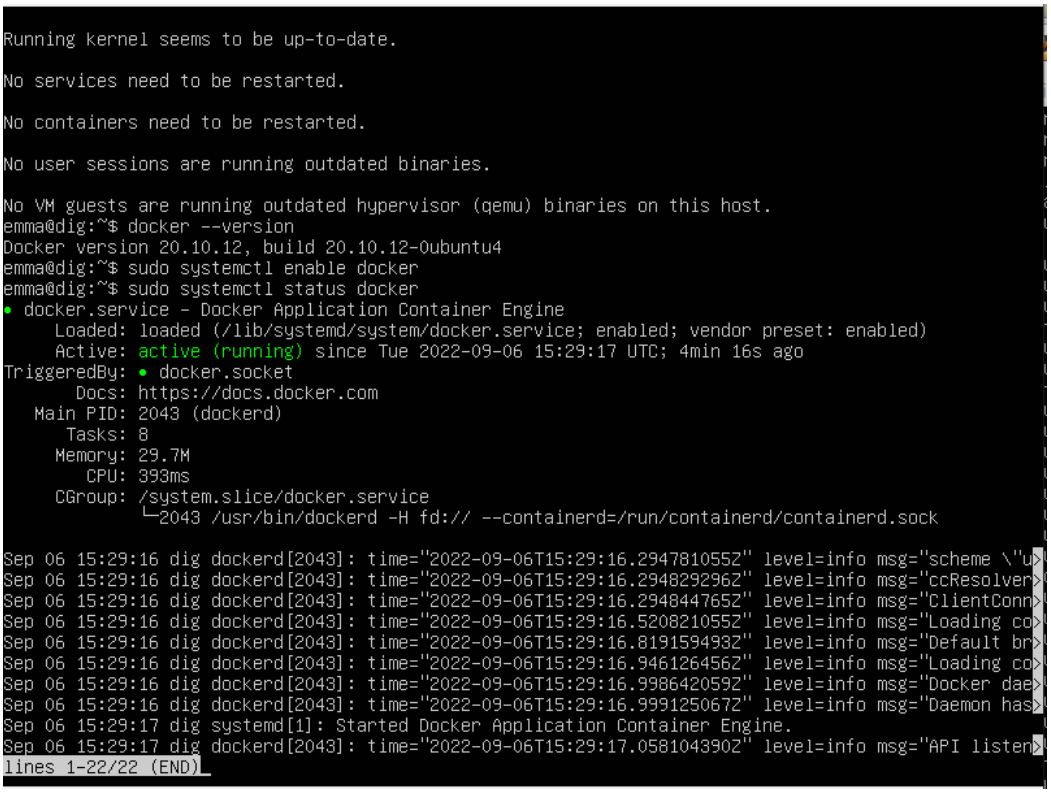

Figure 10 Docker Active

4.5.5 Installing Kubernetes

After OpenStack is deployed, which is an IaaS, we deploy Kubernetes, which is a PaaS. The PaaS Kubernetes is installed on ubuntu.

Step 1: *Adding Signing Key*

It is important to authenticate the file that is being downloaded from the repository. Hence the signing key is downloaded from Google the official provider of the software. This is done by adding a signing key.

1. Adding a signing key:

Command - curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add

Step 2: *Adding Repositories.*

Adding Kubernetes repositories.
Command - apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"

Step 3: *Kubernetes Installation Tools*

Kubernetes Admin (Kubeadm) helps initialize a cluster. It fast-tracks setup by using community-sourced best practices. Kubelet starts containers and runs on every node and gives command-line access to clusters. Kubectl is a command line tool.

1. Installing Kubernetes tools:

Command - apt-get install kubeadm
Command - apt-get install kubelet
Command - apt-get install kubectl
Command - apt-mark hold kubeadm
Command - apt-mark hold kubelet
Command - apt-mark hold kubectl

2. Verify Kubernetes with:

Command –kubeadm version

Kubernetes Deployment

Step 1: *Begin Kubernetes Deployment*

Disabling the swap memory on the server because Kubernetes does not work on a swap-based device. Following command to disable:

Command - swapoff –a
Command - sudo mount -a
Command - free -h

Step 2: *Assign Hostname for Node*

The nodes are named master and worker.

Command - hostnamectl set-hostname child

Command - hostnamectl set-hostname worker01

Step 3*: Initialize Kubernetes Node*

Switching to the master and entering the command to initialize Kubernetes on the master node. Also adding a flag to specify the pod network. IP 192.168.56.115/15 default IP that the kube-flannel uses. This will be used to join the worker nodes to the cluster. Shown in Figure 11.

Command - kubeadm init --pod-network-cidr=192.168.56.115/15

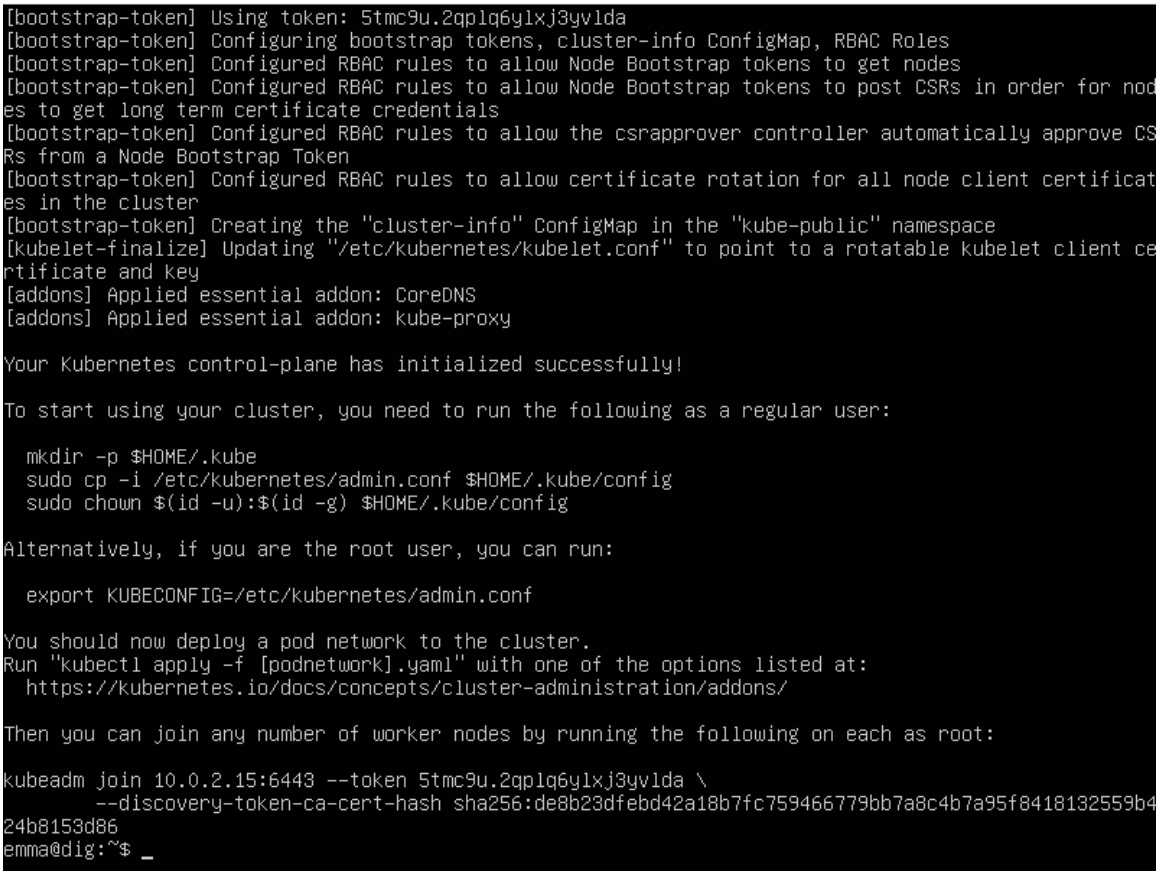

Figure 11 Kubeadm join

Creating a directory for the cluster.

Command - mkdir -p $HOME/.kube

Command - cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

Command - chown $(id -u):$(id -g) $HOME/.kube/config

Step 4: *Deploying Network*

A Pod Network is a way to establish communication between different nodes in the cluster. For example connection between host and pods, between pods to pods, between pods and servicers, and external to service. Container Network Interface (CNI) plugins to manage their network and security capabilities. There are many different existing vendors that provide their services some with basic features others with sophisticated solutions kube-flannel is one of them. We install the kube-flannel by first disabling the default port firewall. Then installing the pod network.

Command - sudo ufw allow 6443

Command - sudo ufw allow 6443/tcp

Command - sudo kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

After the process is complete. Verifying its running and communicating. Shown in Figure 12.

Command - kubectl get pods --all-namespaces

```
emma@dig:~$ mkdir -p $HOME/.kube
emma@dig:~$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
emma@dig:~$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
emma@dig:~$ sudo kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentat
ion/kube-flannel.yml
The connection to the server localhost:8080 was refused - did you specify the right host or port?
emma@dig:~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                READY   STATUS    RESTARTS   AGE
kube-system   coredns-565d847f94-6d88m            0/1     Pending   0          9m3s
kube-system   coredns-565d847f94-tbxxh            0/1     Pending   0          9m3s
kube-system   etcd-worker01                       1/1     Running   0          9m18s
kube-system   kube-apiserver-worker01             1/1     Running   0          9m18s
kube-system   kube-controller-manager-worker01    1/1     Running   0          9m20s
kube-system   kube-proxy-jw85z                    1/1     Running   0          9m3s
kube-system   kube-scheduler-worker01             1/1     Running   0          9m18s
emma@dig:~$
```

Figure 12 Kube resources

Step 5: *Joining Nodes to Cluster*

Switch to the master system and below command is used to join all the node to cluster.

Command - kubeadm join 10.02.15:6443 --token 5tmc9u.2qplxj3yvlda\ --discovery-token-ca-cert-hash sha256:de8b23dfebd42a16b7fc759466779bb7a8c4b7a95f8418132559b424b8153d86

To ensure deployment, the check is done by kubectl. The following command is used to check the running nodes after a few seconds. As shown in Figure 13 below.

Command - kubectl get nodes

```
emma@dig:~$ kubectl get nodes
NAME                STATUS      ROLES                   AGE         VERSION
worker01            Ready       control-plane,master    17m         v1.25.0
child               Ready       <none>                  80s         v1.25.0
emma@dig:~$ kubectl get nodes
```

Figure 13 Nodes

4.5.6 Installing Prometheus

It is an open-source monitoring system that collects metrics from services and stores them in a database. It is a powerful tool when combined with Grafana which helps visualize data in different types. Prometheus can be greatly expanded by installing tools for generating additional metrics. Tools like *Node_Exporter* produce metrics about CPU, memory, disk usage, I/O, and network statistics. Similarly, tools like *Blackbox_Exporter* - generate metrics like endpoint availability and response time.

Firstly, we will start by creating two new users namely Prometheus for Prometheus and Node_exporter for Node-exporter, create a directory after /etc/ Prometheus and /var/lib/Prometheus and set ownership as Promo user.

Command – wget https://github.com/prometheus/prometheus/releases/download/v2.38.0/prometheus-2.38.0.linux-amd64.tar.gz

Unzipping the downloaded file by the below command

Command – tar -xzf prometheus-2.38.0.linux-amd64.tar.gz

Navigate to the directory */ prometheus-2.38.0.linux-amd64* and install it by executing the command.

Command - . /prometheus

Prometheus console can be viewed by opening *localhost:9090*

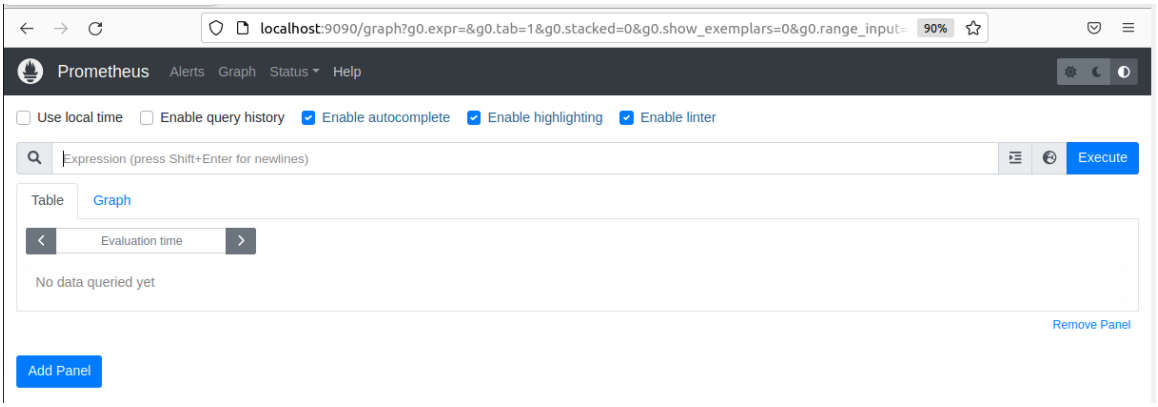

Figure 14 Home page Prometheus

In our second step, we will install Node-exporter, Downloading Prometheus from the webpage.

Command - wget https://github.com/prometheus/node_exporter/releases/download/v1.4.0-rc.0/node_exporter-1.4.0-rc.0.linux-amd64.tar.gz

Unzipping the downloaded file by the below command

Command – tar -xzf node_exporter-1.4.0-rc.0.linux-amd64.tar.gz

Navigate to the directory */ node_exporter-1.4.0-rc.0.linux-amd64.tar.gz* and install it by executing the command.

Command - . / node_exporter

After that the above installation we need to configure Prometheus to recognize the node exporter by updating the prometheus.yml config file.

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
scrape_configs:
  - job_name: 'node_exporter'
    static_configs:
      - targets: ['localhost:9100']
```

Figure 15 Updated prometheus.yml config file

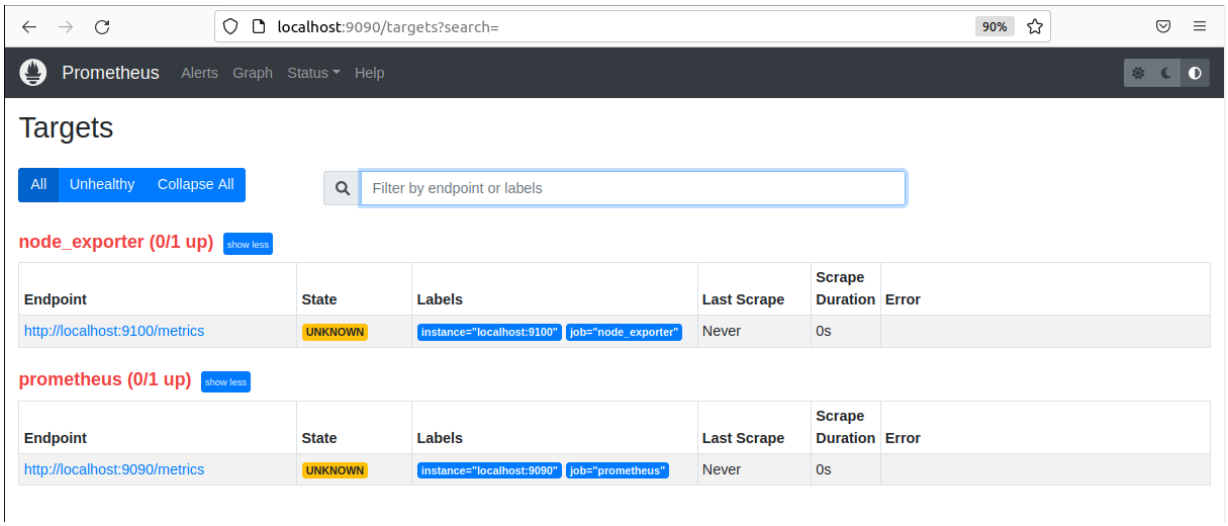

Figure 16 After Noad exporter configuration 1

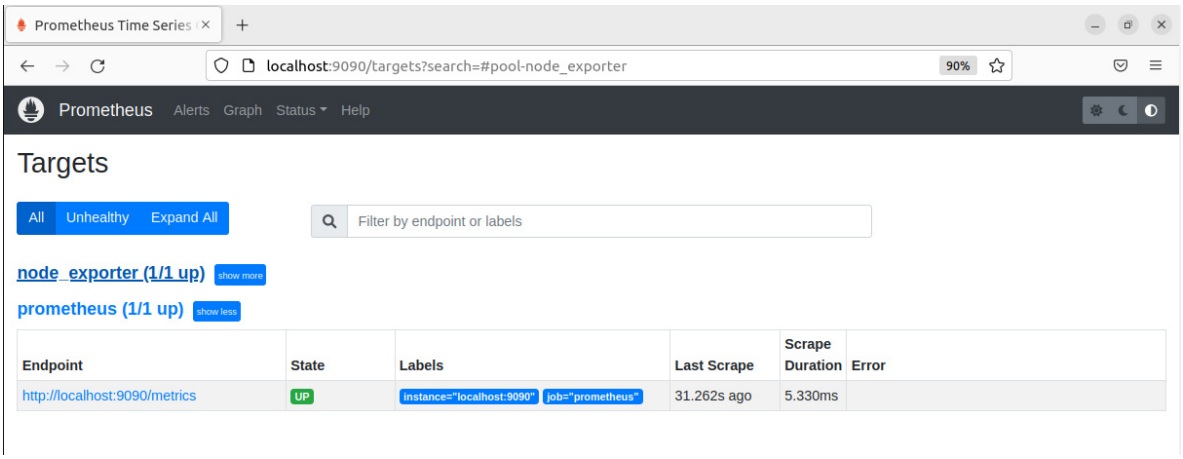

Figure 17 After Noad exporter configuration 2

4.5.7 Application deployment

In this section, we deploy the application. The application used is a containerized application written as microservices to be deployed in Kubernetes.

Two files are created to deploy the application app.service.yml and app.deployment.yml.
Both contain information about how many nodes, port numbers, and their types.
The code spits are given in Figures 18 and 19.

Command - kubectl create - app.deployment.yml
Command - kubectl create -app.service.yml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app
  template:
    metadata:
      labels:
        app: app
    spec:
      containers:
      - name: app
        image: flask
        imagePullPolicy: Always
        ports:
        - containerPort: 3000
```

Figure 18 deployment.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: flask
spec:
  ports:
  - port: 3000
    targetPort: 3000
  selector:
    app: app
```

Figure 19 service.yaml

4.6 Deploying a New Version for Testing

In this step, we update the existing version of the web application using the Kubernetes built-in function for the Blue-Green Deployment strategy.

4.6.1 Blue-Green Deployment

Deploy the new version of the application. We will call the old version as App Blue version, the one which is already deployed which is up and running, and say the new version as App Green. We will first deploy the green version and divert the traffic to the new one. Following Figure 20 shows the yaml file for deploying the green version. Figure 21 Shows diverting the traffic from the blue to the green version.

Command: $kubectl apply -f appgreen.yml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: appv2-blue
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: appv2
        role: blue
    spec:
      containers:
      - name: appv2
        image: flask : v2.2.2
        ports:
        - containerPort: 3000
        readinessProbe:
          httpGet:
            path: /
            port: 3000
```

Figure 20 green yaml file

This is the following code to switch the traffic from blue to green. The yaml content is shown in Figure 21 below.
Command: $kubectl apply -f appservice.yml

```
kind: Service
apiVersion: v1
metadata:
  name: appv2
  labels:
    app: appv2
    role: green
    env: prod
spec:
  type: LoadBalancer
  selector:
    app: appv2
    role: green
  ports:
    - port: 80
      targetPort: 3000
```

Figure 21 Version switch yaml file

4.6.2 Rolling deployment

To update a version using rolling deployment. Version 2 of the application is already in the repository, ready to be pulled. Command: $kubectl apply -f app.yml is used to start the rollout. apple.yaml file contains all the information about the location and cluster information on how and what to deploy. As shown in Figure 22.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: appv2-rolling-update
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: appv2
        role: rolling-update
    spec:
      containers:
      - name: appv2-container
        image: flask
        ports:
        - containerPort: 3001
        readinessProbe:
          httpGet:
            path: /
            port: 3001
  strategy:
    type: RollingUpdate
    rollingUp      maxSurge: 50%
```

Figure 22 Rolling deployment yaml file

4.6.3 Canary Deployment

Like the Blue-Green deployment, the canary deployment is different with only the amount of traffic that is being diverted between blue-green versions. For testing the stability, a small percent of traffic is diverted to the new version. Firstly, we check for the running version by using the command: kubectl get service which will display the external command. Command: $kubectl apply -f app-canary.yml to deploy the new as shown in Figure 23. In the first stage 20% of traffic flow through v1 as shown in Figure 24. The reason for selecting 20% is because the workload selected is small compared to the real-time workload and to stress the system a little more to check if the system fails. If the deployment is successful then 100% of traffic is diverted by executing service V2 as shown in Figure 25. Bellow Figure 23 shows the content of the YAML file. The result of this experiment is presented in section 5 below.

```
apiVersion: extensions/v2beta2
kind: Deployment
metadata:
  name: appv2-canary
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: appv2
        role: canary
    spec:
      containers:
      - name: appv2
        image: flask : canary
        ports:
        - containerPort: 3000
        readinessProbe:
          httpGet:
            path: /
            port: 3000
```

Figure 23 Canary deploy yml

```
apiVersion:extensions/v2beta2
kind: Service
metadata:
  name: appv2-canary
spec:
  type : LoadBalancer
  - route:
    - destination:
        port:80
          number: 3000
        subset: v1
      weight: 80
    - destination:
        host:
        port:80
          number: 3001
        subset: v2
      weight: 20
```

Figure 24 Canary service v1 20% yml

```
apiVersion:extensions/v2beta2
kind: Service
metadata:
  name: appv2-canary
spec:
  type : LoadBalancer
  - route:
    - destination:
        port:80
          number: 3001
        subset: v2
      weight: 100
```

Figure 25 Canary service v2 100% yml

5. RESULTS AND ANALYSIS

This section presents the results of experiments from chapter 4. The presented results are from the manually deployed Kubernetes cluster and automatic deployment. The tool used for performance evaluation is Google lighthouse, and the monitoring tool is Prometheus.

5.1 SLR results

Table 1 contains the list of articles obtained from the SLR. SLR helped in selecting appropriate strategies, tools, and metrics to measure the performance of strategies for this research. It also demonstrated that blue-green deployment with DNS swapping also performed better compared to load balancer techniques and software reconfiguration [5]. The studies show that values of CPU and RAM are important to understand the functionality of the deployment process[3],[4]. The articles used Blue Green, Canary, and Rolling deployment as the main deployment strategies to test their theory[1],[2],[4],[5],[41]. They also used tools like Docker, Kubernetes, Ansible, Promentues, AWS, Azure, Google Light House, and OpenStack, to conduct their experiment[1],[2],[3],[4],[5].

5.2 Test Results

The experiment results are presented below. It is best to compare multiple metrics while comparing the performance of more than one strategy. The main reason for comparing multiple factors and metrics is to check the hypothesis that other factors affect the deployment process, and also test the hypothesis that deployment strategies produce zero downtime for services during the deployment process. It is essential to check the hypothesis because we will have proof that they all produce zero downtime of service, and their efficiency in time can be obtained by experimentation with a single application.

5.2.1 Blue-Green Deployment

The experiment was conducted with three workloads to check the stability during the deployment. The three workloads selected are 100,1000 and 10000 users. The reason for selecting these workloads is to check whether the service drops during different workloads.

5.2.1.1 CPU Usage

The CPU usage metric shows the consumption and utilization of the CPU availably during the deployment process. The presented data in Figure 26 show whether the Blue-Green deployment process directly affects the CPU utilization of the cluster.

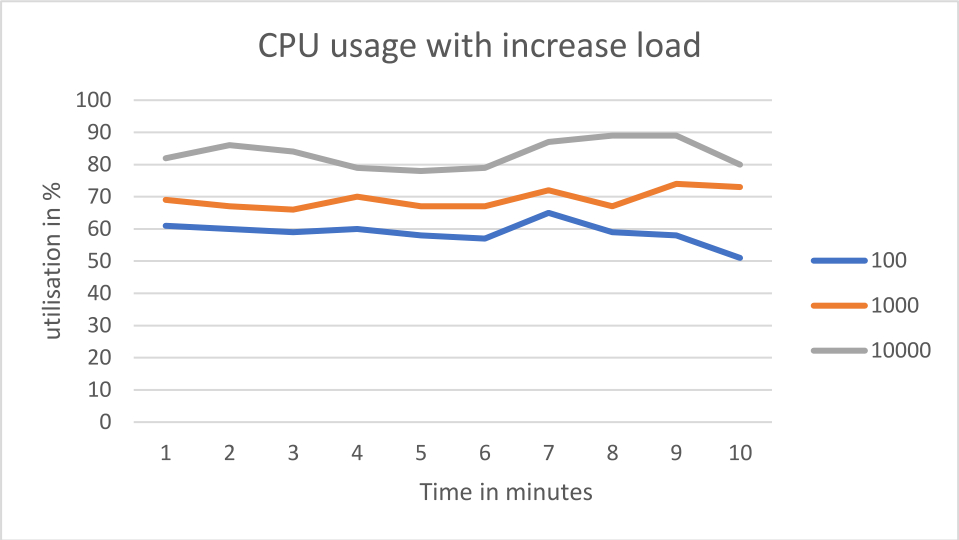

Figure 26 CPU Usage Blue Green

The Blue-Green deployments experiment shown above proves that the deployment doesn't correlate with CPU consumption. The deployment has no direct effect on CPU usage, as shown in Figure 26. Blue-Green deployment does not affect the CPU utilization in the automated deployment process with negligible change in CPU usage.

5.2.1.2 Memory usage

Memory usage plays an essential role in the Blue-Green deployment automation process. Since an essential component of the application and the other process, request processing is managed by the Memory. Hence checking whether the Memory is being affected by the deployment process is essential. Figure 27 shows the RAM (Random Access Memory) usage during the Blue-Green deployment process.

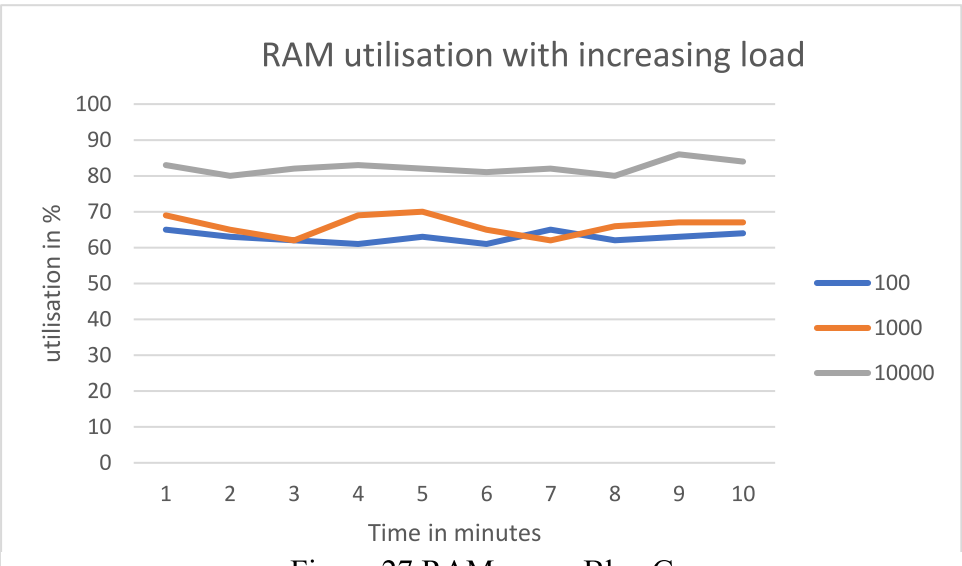

Figure 27 RAM usage Blue Green

The above graph shows the Memory used when deploying a new web application. Figure 27 shows the RAM usage in manual clusters. The pipeline creates a backup in the Memory before the deployment, thus increasing the Memory. Furthermore, the pipeline engine also does consume vast memory when deployed in the same application cluster. Figure 27 shows that the pipeline consumed 15% more Memory than the manual cluster.

5.2.1.3 Service Availability During Blue-Green deployment

The critical metric and essential to answer the research question is service downtime while using an automatic deployment strategy. This metric checks whether the service drops while a new version of the application is spawned automatically, and the traffic is diverted to the new version. The deployment process is done without interruption in the 10000 workloads. The screenshot depicts the x-axis as a service utilization of the master cluster and Y axis as the time.

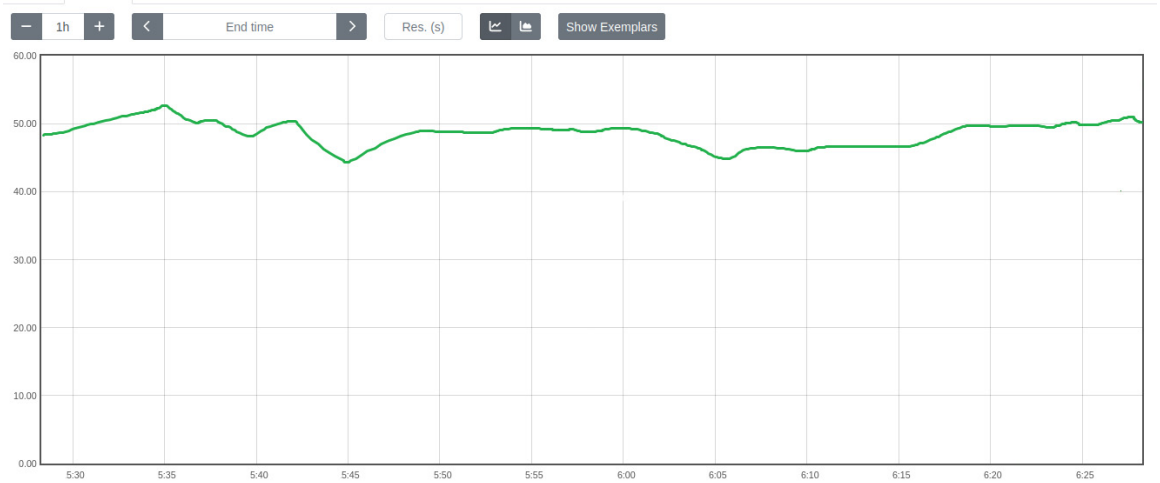

Figure 28 Service availability Blue Green

Figure 28 shows no service interruptions during the Blue-Green deployment process, and the process did not result in any service drop during deployment. The application was 100% online all the time. All three deployments showed similar results within Blue-Green Deployment. No services drop at all.

5.2.2 Canary Deployment

The experiment was conducted with three workloads to check the stability during the deployment. The three workloads selected are 100,1000 and 10000. The reason for selecting these workloads is to check whether the service drops during different workloads.

5.2.2.1 CPU Usage

The CPU usage metric shows the consumption and utilization of the CPU availably during the deployment process. The presented data in Figure 29 show whether the deployment process directly affects the CPU utilization of the cluster.

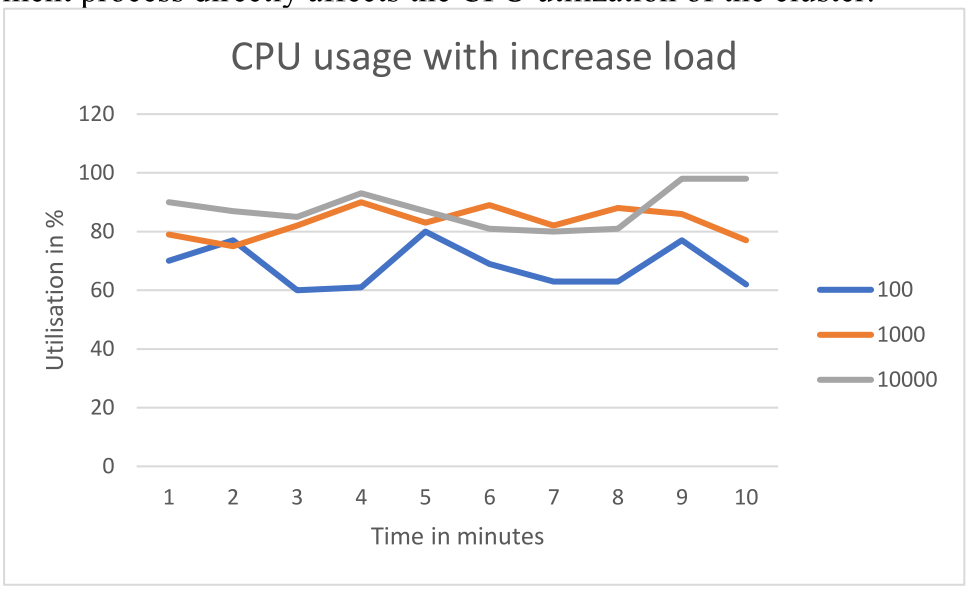

Figure 29 CPU usage Canary

The canary deployments experiment shown above proves that resource deployment doesn't correlate with CPU consumption. The deployment has no direct effect on CPU usage, as shown in Figure 29. Canary deployment does not affect the CPU utilization in the automated deployment process with negligible CPU usage change.

5.2.2.2 Memory usage

Memory usage plays an essential role in the canary deployment automation process. Since an essential component of the application and the other process, request processing is managed by the Memory. Hence checking whether the Memory is being affected by the deployment process is essential. Figure 30 shows the RAM usage during the canary deployment process.

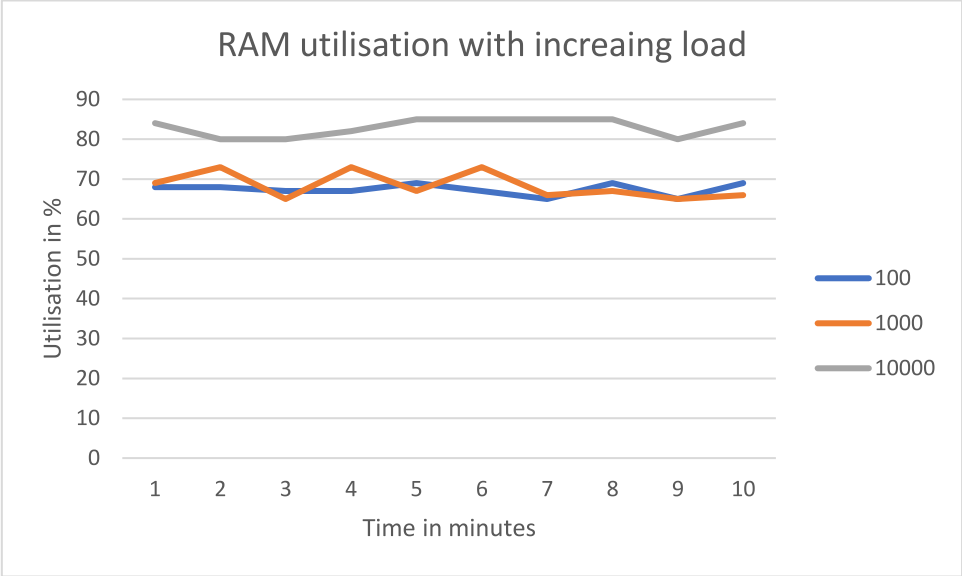

Figure 30 RAM usage Canary

The above graph shows the Memory used when deploying a new web application to figure out the RAM usage in manual clusters. The pipeline creates a backup in the Memory before the deployment, thus increasing the Memory. Figure 30 shows that the pipeline consumed 15% more Memory compared to the manual cluster.

5.2.2.3 Service Availability During Canary deployment

While using a canary deployment strategy, this metric shows whether the service dropped while a new version of the application is spawned automatically, whether the traffic is diverted to the new version and whether the deployment process is done without interruption. The deployment process is done without interruption in the 10000 workloads. The screenshot depicts the x-axis as a service utilization of the master cluster and Y axis as the time.

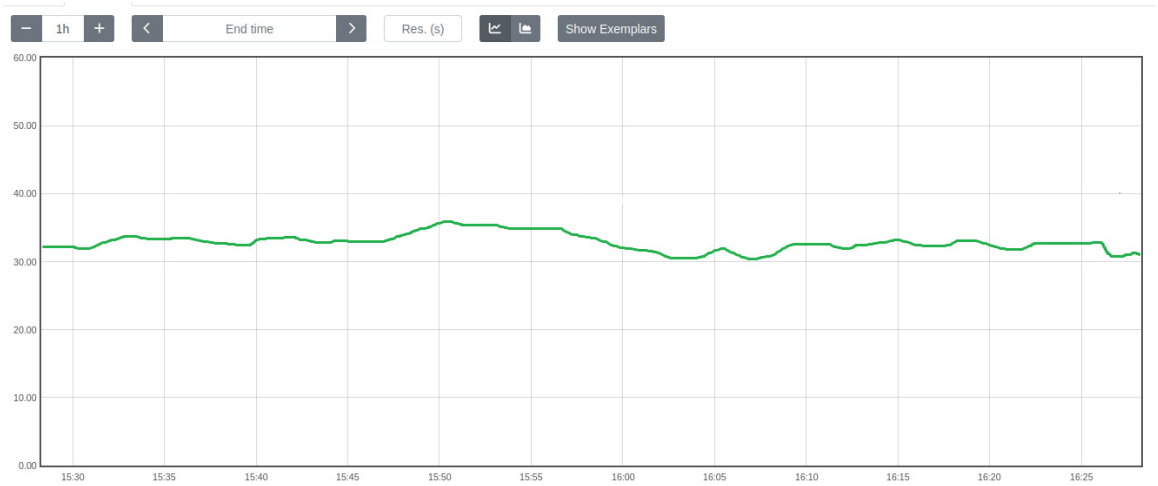

Figure 31 Service availability Canary

Figure 31 shows no service interruption during the Canary deployment process, and the process did not result in any service drop during deployment. The

application was 100% online all the time. All three deployments showed similar results within Canary Deployment. No services drop at all.

5.2.3 Rolling deployment

The experiment was conducted with three workloads to check the stability during the deployment. The three workloads selected are 100,1000 and 10000. The reason for selecting these workloads is to check whether the service drops during different workloads.

5.2.3.1 CPU Usage

The CPU usage metric shows the consumption and utilization of the CPU availably during the deployment process. The presented data in Figure 32 show whether the Rolling deployment process directly affects the CPU utilization of the cluster.

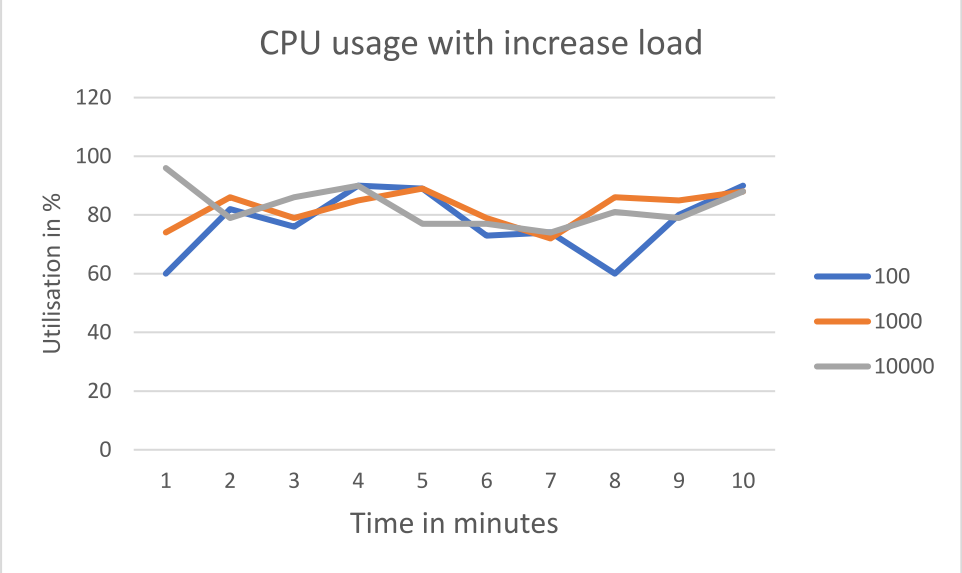

Figure 32 CPU usage Rolling

The Rolling deployments experiment shown above proves that resource deployment doesn't correlate with CPU consumption. The deployment has no direct effect on CPU usage, as shown in Figure 32. Rolling deployment does not affect the CPU utilization in the automated deployment process with negligible CPU usage.

5.2.3.2 Memory usage

Memory usage plays an essential role in the Rolling deployment automation process. Since an essential component of the application and the other process, request processing is managed by the Memory. Hence checking whether the Memory is being affected by the deployment process is essential. Figure 33 shows the RAM usage during the Rolling deployment process.

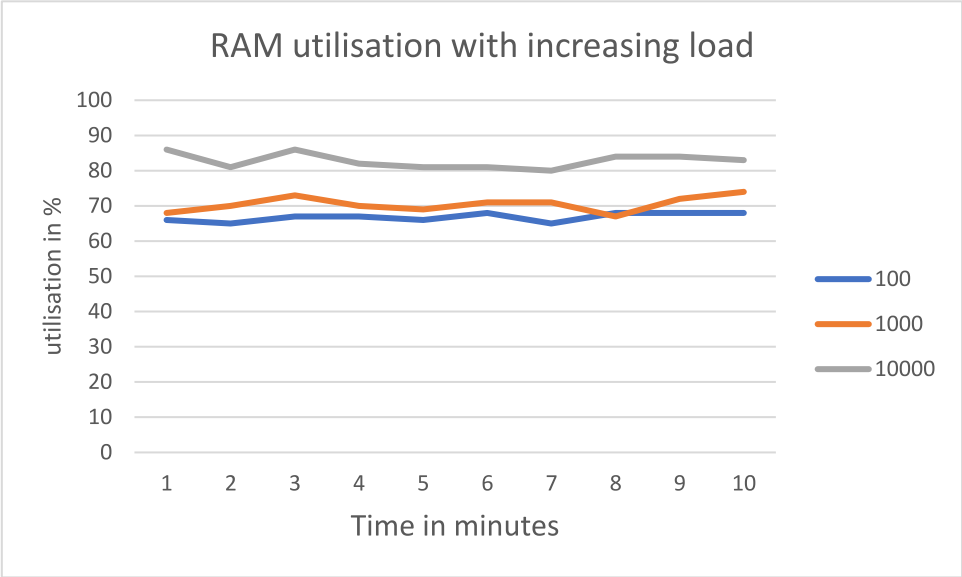

Figure 33 RAM usage Rolling

The above graph shows the Memory used when deploying a new web application. Figure 33 shows the RAM usage in manual clusters. Furthermore, the pipeline engine also does memory consumption when deployed in the same application cluster. It is also clear from Figure 33 that the Rolling pipeline consumed 15% more Memory than the manual cluster in Figure 33.

5.2.3.3 Service Availability During Rolling deployment

While using a canary deployment strategy, this metric shows whether the service dropped while a new version of the application is spawned automatically, whether the traffic is diverted to the new version and whether the deployment process is done without interruption. The deployment process is done without interruption in the 10000 user workloads. The screenshot depicts the x-axis as a service utilization of the master cluster and Y axis as the time.

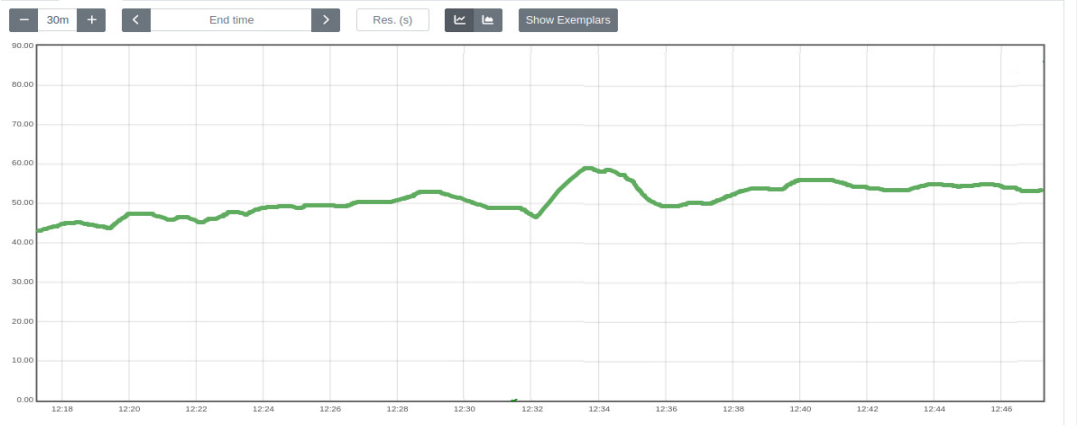

Figure 34 Service availability Rolling

Figure 34 shows no service interruption during the Rolling deployment process, and the process did not result in any service drop during deployment. The application was 100% online during the whole deployment process. All three deployments showed similar results within Rolling Deployment. No services drop at all.

5.2.4 Without deployment strategy / manual deployment

The experiment was conducted with three workloads to check the stability during the deployment. The three workloads selected are 100,1000 and 10000. The reason for selecting these workloads is to check whether the service drops during different workloads. The result in this section is from the manual process of the deployment strategy when code is executed to initiate the master Kubernetes cluster to start the deployment process.

5.2.4.1 CPU usage

The CPU usage metric shows the consumption and utilization of the CPU availably during the manual deployment process. The presented data in Figure 35 show whether the manual deployment process directly affects the CPU utilization of the cluster.

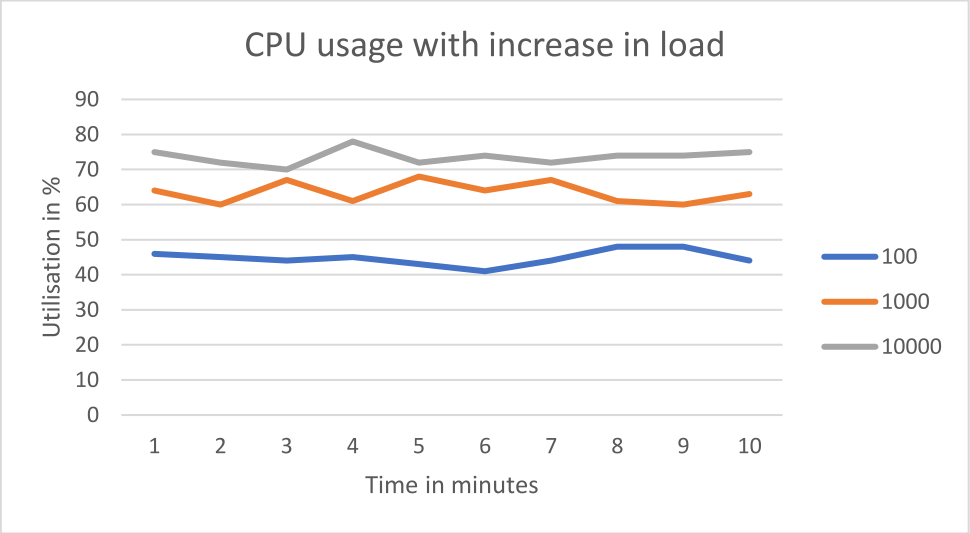

Figure 35 CPU usage manual

The manual deployment experiment shown above proves that resource deployment does correlate with CPU consumption. The deployment directly does not affects CPU usage, as shown in Figure 35, compared to automatic deployments, as shown in Figures 26, 29, and 32. Manual deployment does not affect the CPU utilization in the manual deployment process, but a 20% decrease in CPU usage compared to the automatic deployment process in Figures 26, 29, and 32 has been observed.

5.2.4.2 Memory usage

Memory usage plays an essential role in the deployment process. Since an essential component of the application and the other process, request processing is managed by the Memory. Hence checking whether the Memory is being affected by the deployment process is essential. Figure 36 shows the RAM usage during the deployment process.

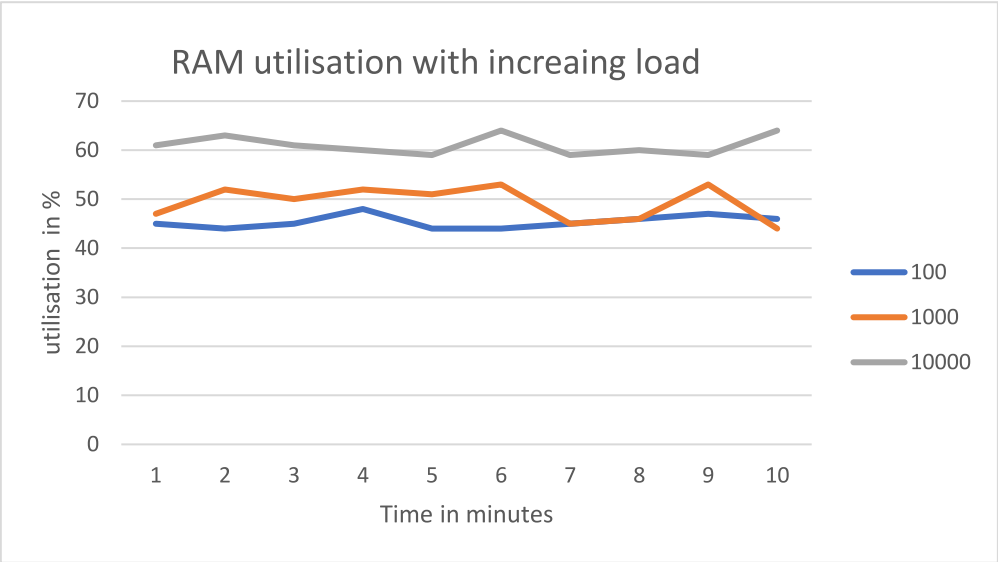

Figure 36 RAM usage Manual

By comparing figure 27, figure 30, figure 33, and Figure 36, we observe 15% less memory consumption due to service interruptions during the deployment process.

5.2.4.3 Manual Deployment Service Availability

The metric to measure the service availability during the deployment process when done in manual steps rather than automatic deployment is presented in this section. When the deployment is initiated manually by executing Kublet code to switch the version, obtained results are presented in figure 37 below, representing service availability and service drop during the deployment procedure. The deployment process is done with a service interruption in the 10000 workloads as shown in figure 37. The screenshot depicts the x-axis as a service utilization of the master cluster and Y axis as the time.

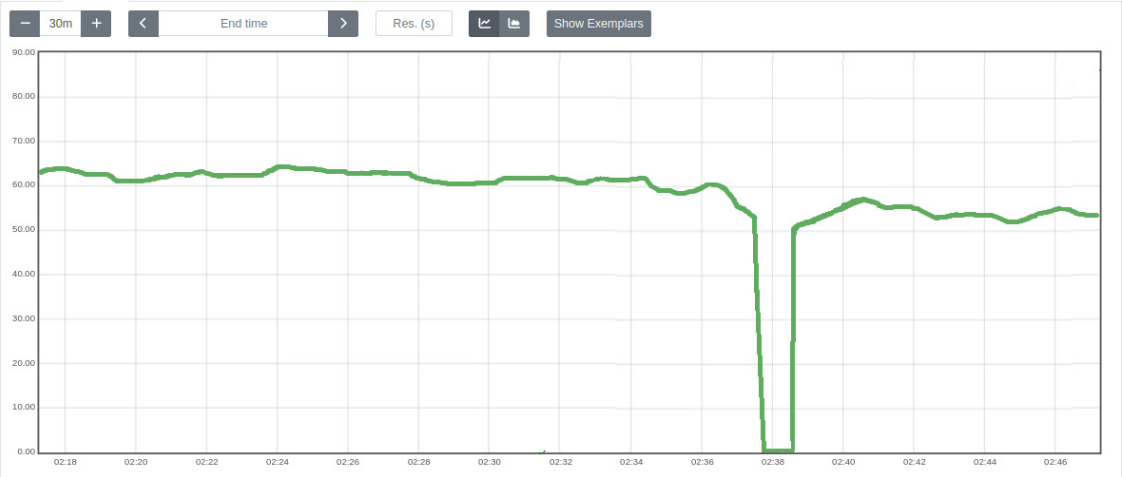

Figure 37 Service availability Manual

As it is visible from the service availability graph for the manual deployment process, the service drop does occur for all deployments workload. Service drop occurs between 12:27:56 and 12:28:19 seconds, which is a 23-second interval. All three deployment workloads showed similar results within manual Deployment.

5.2.5 Deployment strategies results

metrics	Deployment type			
	Blue Green	Canary	Rolling	manual
Response time (RT)	116.5	128.6	95.9	17.2
Standard deviation of RT	14.8	12.5	13.2	3.0
Overlap Duration (OD)	15.5	12.1	43.5	-
Standard deviation of OD	4.2	4.3	9.3	-
Switch time (ST)	105.2	501.6	255.5	58.9
Standard Deviation of ST	13.5	38.5	27.6	4.2
Service Downtime (SD)	-	-	-	23
Standard deviation (SDT)	-	-	-	7.6

Table 3 Deployment Results

The reason for conducting the experiment and obtaining these results is to test the hypothesis that one deployment performs better than others. It is essential to check the hypothesis because it will experimentally exhibit the DevOps community to

select better-performing deployment strategies for updating a web application without service downtime. The results produced by the experiment are presented in table 3. The results show that Blue-Green deployment is the fastest way to deploy new services without dropping services, with a switch time of 105.2 sec, which is four times faster than canary deployment. Canary is four times slower than Blue-Green and one time faster than Rolling deployment. All deployments have different Response times to start the deployment after the new version is uploaded to the version controller. Manuel deployment seems to be the fastest but produces service downtime, which is undesirable for updating an application with service interruptions. Overlap duration is the time when the deployment is spent changing old ones to new ones. The point to be observed is that the Canary deployment is similar in resources used to the Blue-Green deployment. Nonetheless, the only difference is the incremental traffic routing strategy in the Canary is to check the load balancing of the new version, which is the reason for slow deployment. Due to the slow traffic diversion process, switching from the old to the new version will increase the switching time.

6 DISCUSSION

6.1 Answering RQ1

An important aspect of doing an in-depth analysis of the literature is, first and foremost, gaining knowledge on the past approaches; secondly and most importantly, there is limited research on the deployment process. It is well-documented that switching services from old to new will cause a service outage [2]. Software development moving toward deploying service without interruption is the best action and choice.

The SLR shows that three deployment processes can achieve high availability of service during new application deployment. The three deployment processes identified were Blue-Green deployment, canary deployment, and rolling deployment. All articles from SLR used different methods and metrics to measure the functionality in different scenarios but the most observed were CPU, RAM, and service uptime. After the experiment, the following can be concluded as mentioned below.

The answer to whether deployment strategies produce zero downtime deployment during bug fixing or updating an application is yes. The thesis proves from Section 5.2.5 presents data that all three deployments produce zero downtime during deployments without new version failure when using automatic deployment procedure. The service drop is observed when an automatic deployment type is not used. Hence Blue-Green Deployments, canary deployments, and Rolling deployments produce zero downtime during service upgrades when there is no external influence on the process.

6.2 Answering RQ2

The three strategies discussed in the thesis are used to mitigate service downtime during bug fixes or update applications to a newer version. Previous studies were able to understand one or the other strategies clearly but failed to address all three types for efficiency.

This study found that even though all three deployment strategies have been found to be producing zero downtime service deployment. However, the switching interval between the old and new applications seems to be vastly different. In case of the new version failure, the deployment can fail and produce non-Zero downtime for some users.

In the case of Blue-Green deployment, the switch seems to be fast, with a switching time of 105.2 seconds, as shown in Table 3, five times faster than the Canary Deployment, but the resource requirement for the new version is equal to the old one to balance the traffic load. So, we need multiple resources of the same

capabilities. Every deployment cost per resource allocation will be more. But in case of version failure, the switchback happens after a few seconds, which is a service downtime. The time duration of the newer version being online will be non-zero downtime.

In the case of canary deployment, it seems to be producing zero downtime deployment, but the switching time between old and new in the initial phase is high compared to Blue-Green deployment. From beginning to end, running both resources for traffic is resource-demanding. All components for both versions should be up and running for the duration of the process, which routes traffic incrementally. Routing traffic for a longer duration for both resources will be more resource-demanding and costly. But in this case, if the new version fails, the downtime felt by the users is less than the whole user traffic load. Canary deployment was the slowest, with a switching time of 501.6, as shown in section 5.2.5 Table 3 of all their deployment in our case—the case of updating a web application without service drop.

The Rolling deployment strategy also produced zero service downtime during the version deployment. We found out that this strategy is better for microservice architecture because of the architecture of microservice. Each component can be and is run on a single node in a cluster of nodes. Each node running an individual service can be updated individually. Resource allocation will be less. If the version fails, the one being updated the update can be rolled back to the old version. The case of managing dependency between the nodes and APIs is time-consuming. According to the results rolling deployment took 255.5 seconds to switch from old to new.

All three deployment strategies have different criteria affecting their process of deployment. Blue-Green deployment is fast and straightforward but requires double resources, Canary take more time during traffic routing with double the resources, and Rolling can be deployed on the same nodes and same resources the old application is running but need human intervention at different stage also mentioned in [4] in this case can be fully automized with tools like Ansible. Apart from these results, I have also looked into manual deployment; much to my interest, manually turned out to be the fastest of all deployments with 58.9 seconds, but service downtime was observed.

6.3 Validity Threats

6.3.1 Internal Validity

Internal validity is concerned with research conducted to answer the question. In our research, three deployment types within Blue Green deployment can be carried out[4]. Opting for the best one to conduct the experiment depends on the strategy being less time-consuming and the research conducted by other Authors [4]. But we

are unaware of the author's condition and other aspects like deployed applications and resources for the cloud cluster selected in this research [4]. Hence, selected the least deployment time-consuming deployment for our research.

6.3.2 External Validity

External validity refers to the extent to which the results obtained from the experiment could be generalized to different applications and cloud environments. In this context, cloud providers, databases, and the tools used in this study can be the major threats as they might be outdated or vary due to hardware and software resource limitations compared with other resource providers. A literature review was conducted, and the best open-source tools from the previous research have been selected to conduct the research.

6.3.3 Conclusion Validity.

Conclusion validity threat is concerned with the accuracy of the results we conclude from the research. In my research, we treated this validity by following the proper steps in implementation and selecting the correct method to conduct the research. Also, we selected performance metrics to interpret the results correctly and reached conclusions.

7 CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this thesis, an experiment is conducted to validate and compare the performance of three application deployment strategies that can be used in any software development life cycle: Cloud-based, Server-based, or Microservice software Architecture-based deployment [20]. The experiment is conducted by deploying a new web application version while the old version is online by using the deployment strategies such as Blue-Green Deployment, Canary Deployment, and Rolling Deployment. Two aspects of the deployment strategies have been observed by experiment. Does the service drop during the update process and which deployment is faster is observed? These strategies are used to mitigate service downtime during bug fixes or update applications to a newer version in all three cases. Previous studies were able to understand one or the other strategies clearly but failed to address all three types for efficacy and efficiency. The experiment conclude that all strategies deployed a new version without any downtime and Blue Green was the fastest of all three.

This study found that even though all three deployment strategies have been found to be producing zero downtime service deployment, the switching interval between the old and new applications seems to be vastly different. In the case of application failure, the deployment can fail and produce non-Zero downtime, partly because of the way the deployment strategy is executed.

In the case of Blue-Green deployment, the switch seems to be fast, but the resource requirement for the new version is equal to the old one to balance the traffic load. So, we need two resources with the same capabilities. Every deployment cost per resource allocation will be more. But in case of version failure, the switchback happens after a few seconds, which is a service downtime. The time duration the newer version was online will be the non-zero downtime.

In the case of canary deployment, it seems to be producing zero downtime deployment, but the switching time between old and new for the initial testing phase is high compared to Blue-Green deployment. From beginning to end, running both resources for traffic is resource-demanding. All components for both versions should be up and running for the duration of the process, which routes traffic incrementally. Routing traffic for a longer duration for both resources will be more resource-demanding and costly. But in this case, if the new version fails, the downtime felt by the users is comparatively less than the whole user load. The small percentage of user traffic routed to the new version in the initial phase will experience downtime before switching back to the old version.

The Rolling deployment strategy also produced zero service downtime during the version deployment. We found out that this strategy is better for microservice

architecture because the microservice architecture is loosely coupled services components of an application. Each component can be and is run on a single server pod in a cluster of nodes. When each service of an application is running on different nodes, there is no need to update or make changes to the whole application, or the system is mitigated, and updating a single component running on a single node will be resource efficient. It is easier to update individual pods where individual microservices are running but managing dependency between the nodes and API's is time-consuming. In the Rolling deployment strategy, any number of components can be updated by replacing old applications with new ones running on the individual node or the same old resources in a cluster.

All three deployment strategies have different criteria affecting their process of deployment. Blue-Green deployment is fast and straightforward but requires double resources, Canary take more time during traffic routing with double the resources, and Rolling can be deployed on the same nodes and same resources the old application is running but need human intervention at different stage also mentioned in [4] in this case can be fully automized with tools like Ansible.

7.2 Future Work

This thesis is proof of concept and protects the idea for further investigation. Therefore, there are many possible future works we could discuss. In this section, I have mentioned a few future studies out of many that could be conducted using this concept.

The study is conducted with a single application on a single node with a Kubernetes cluster as a master node to measure the basic efficiency of the deployment strategies. It would be interesting to see how strategies behave with multiple applications, nodes, and clusters.

One of the future directions would be to increase the CPU and RAM capacity of the cluster to check whether the performance of the deployment strategies. Increasingly in the case of Rolling deployment as it is dependent on the master node to replace the old version of the application on the same node or resources.

For future work, the same strategies can be extended to be tested for an android or mobile application update which closes while updating.

References

[1] Nilsson, Axel. "Zero-Downtime Deployment in a High Availability Architecture: Controlledexperiment of deployment automation in a high availability architecture." (2018).

[2] Jendi, Khaled. "Evaluation and Improvement of Application Deployment in Hybrid Edge Cloud Environment: Using OpenStack, Kubernetes, and Spinnaker." (2020).

[3] A. Buzachis, A. Galletta, A. Celesti, L. Carnevale and M. Villari, "Towards Osmotic Computing: a Blue-Green Strategy for the Fast Re-Deployment of Microservices," 2019 IEEE Symposium on Computers and Communications (ISCC), Barcelona, Spain, 2019, pp. 1-6, doi:10.1109/ISCC47284.2019.8969621.-

[4] C. K. Rudrabhatla, "Comparison of zero downtime based deployment techniques in public cloud infrastructure," 2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile,Analytics and Cloud) (I-SMAC), Palladam, India, 2020, pp. 1082-1086, doi: 10.1109/ISMAC49090.2020.9243605.

[5] B. Yang, A. Sailer, S. Jain, A. E. Tomala-Reyes, M. Singh and A. Ramnath, "Service Discovery Based Blue-Green Deployment Technique in Cloud Native Environments," 2018IEEE International Conference on Services Computing (SCC), San Francisco, CA, 2018, pp. 185-192, doi: 10.1109/SCC.2018.00031.

[6] Singh, Charanjot, et al. "Comparison of Different CI/CD Tools Integrated with Cloud Platform." 2019 9th International Conference on Cloud Computing, Data Science & Engineering(Confluence). IEEE, 2019.

[7] W. Yiran, Z. Tongyang and G. Yidong, "Design and implementation of continuous integration scheme based on Jenkins and Ansible," 2018 International Conference on Artificial Intelligence and Big Data (ICAIBD), Chengdu, China, 2018, pp. 245-249, doi: 10.1109/ICAIBD.2018.8396203.-

[8] Chen, Lianping. "Continuous delivery: Huge benefits, but challenges too." IEEE Software32.2 (2015): 50-54.

[9] Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V. P., Itkonen, J., Mäntylä, M. V., & Männistö, T. (2015). The highways and country roads to continuous deployment. Ieee software, 32(2), 64-72.

[10] S. Mysari and V. Bejgam, "Continuous Integration and Continuous Deployment Pipeline Automation Using Jenkins Ansible," 2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE), Vellore, India, 2020, pp. 1-4, doi: 10.1109/ic-ETITE47903.2020.239.-

[11] A. Cepuc, R. Botez, O. Craciun, I. -A. Ivanciu and V. Dobrota, "Implementation of a Continuous Integration and Deployment Pipeline for Containerized Applications in Amazon Web Services Using Jenkins, Ansible and Kubernetes," 2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet), Bucharest, Romania, 2020, pp. 1-6, doi: 10.1109/RoEduNet51892.2020.9324857.

[12] C. R. KOTHARI “Research Methodology Method and Techniques”, 2nd ed, Jaipur, India, New Age International (P) Limited Publisher, 1990, ch 1, 6

[13] IBM “Cloud Computing” [Online]
Available : https://www.ibm.com/cloud/learn/cloud-computing
[Accessed On: 10 april 2022]

[14] Azure “overview” [Online]
Available : https://azure.microsoft.com/en-in/overview
[Accessed On: 10 april 2022]

[15] Codefresh “continuous-deployment/ci-cd-pipelines-microservices” [Online]
Available : https://codefresh.io/continuous-deployment/ci-cd-pipelines-microservices/ [Accessed On: 11 april 2022]

[16] Kubernetes “Open source system” [Online]
Available : https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[Accessed: 11 april 2022]

[17] H. Kang, M. Le and S. Tao, "Container and Microservice Driven Design for Cloud Infrastructure DevOps," 2016 IEEE International Conference on Cloud Engineering (IC2E), 2016, pp. 202-211, doi: 10.1109/IC2E.2016.26.

[18] K. Bakshi, "Microservices-based software architecture and approaches," 2017 IEEE Aerospace Conference, 2017, pp. 1-8, doi: 10.1109/AERO.2017.7943959.

[19] L. Chen, "Continuous Delivery: Huge Benefits, but Challenges Too," in IEEE Software, vol. 32, no. 2, pp. 50-54, Mar.-Apr. 2015, doi: 10.1109/MS.2015.27.

[20] A. Buzachis, A. Galletta, A. Celesti, L. Carnevale and M. Villari, "Towards Osmotic Computing: a Blue-Green Strategy for the Fast Re-Deployment of Microservices," 2019 IEEE Symposium on Computers and Communications (ISCC), Barcelona, Spain, 2019, pp. 1-6, doi: 10.1109/ISCC47284.2019.8969621.-

[21] Lucké, Balduin, and Morton McCutcheon. "The living cell as an osmotic system and its permeability to water." Physiological Reviews 12, no. 1 (1932): 68-139.

[22] Chan, Lai-Kow, and Ming-Lu Wu. "Quality function deployment: A literature review." European journal of operational research 143, no. 3 (2002): 463-497.

[23] Thomas, Stephen J., and In-Kyu Yoon. "A review of Dengvaxia®: Development to deployment." Human vaccines & immunotherapeutics 15, no. 10 (2019): 2295-2314.

[24] Dearle, Alan. "Software deployment, past, present and future." In Future of Software Engineering (FOSE'07), pp. 269-284. IEEE, 2007.

[25] Feitelson, Dror G., Eitan Frachtenberg, and Kent L. Beck. "Development and deployment at facebook." IEEE Internet Computing 17, no. 4 (2013): 8-17.

[26] Younis, Ossama, Marwan Krunz, and Srinivasan Ramasubramanian. "Node clustering in wireless sensor networks: Recent developments and deployment challenges." *IEEE network* 20, no. 3 (2006): 20-25.

[27] Gajbhiye, Pradnya, and Anjali Mahajan. "A survey of architecture and node deployment in wireless sensor network." In *2008 First International Conference on the Applications of Digital Information and Web Technologies (ICADIWT)*, pp. 426-430. Ieee, 2008.

[28] Zhang, Haitao, and Cuiping Liu. "A review on node deployment of wireless sensor network." *International Journal of Computer Science Issues (IJCSI)* 9, no. 6 (2012): 378.

[29] Xu, Kenan, Hossam Hassanein, Glen Takahara, and Quanhong Wang. "Relay node deployment strategies in heterogeneous wireless sensor networks." *IEEE Transactions on Mobile Computing* 9, no. 2 (2009): 145-159.

[30] Savor, Tony, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. "Continuous deployment at Facebook and OANDA." In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 21-30. IEEE, 2016.

[31] Rodríguez, Pilar, Alireza Haghighatkhah, Lucy Ellen Lwakatare, Susanna Teppola, Tanja Suomalainen, Juho Eskeli, Teemu Karvonen, Pasi Kuvaja, June M. Verner, and Markku Oivo. "Continuous deployment of software intensive products and services: A systematic mapping study." *Journal of Systems and Software* 123 (2017): 263-291.

[32] Leppänen, Marko, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V. Mäntylä, and Tomi Männistö. "The highways and country roads to continuous deployment." *Ieee software* 32, no. 2 (2015): 64-72.

[33] Claps, Gerry Gerard, Richard Berntsson Svensson, and Aybüke Aurum. "On the journey to continuous deployment: Technical and social challenges along the way." *Information and Software technology* 57 (2015): 21-31.

[34] Kalloniatis, Christos, Haralambos Mouratidis, and Shareeful Islam. "Evaluating cloud deployment scenarios based on security and privacy requirements." *Requirements Engineering* 18, no. 4 (2013): 299-319.

[35] Roloff, Eduardo, Matthias Diener, Alexandre Carissimi, and Philippe OA Navaux. "High performance computing in the cloud: Deployment, performance and cost efficiency." In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pp. 371-378. IEEE, 2012.

[36] Fittkau, Florian, Sören Frey, and Wilhelm Hasselbring. "CDOSim: Simulating cloud deployment options for software migration support." In *2012 IEEE 6th International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA)*, pp. 37-46. IEEE, 2012.

[37] Islam, Shareeful, Moussa Ouedraogo, Christos Kalloniatis, Haralambos Mouratidis, and Stefanos Gritzalis. "Assurance of security and privacy requirements for cloud deployment models." *IEEE Transactions on Cloud Computing* 6, no. 2 (2015): 387-400.

[38] Rahman, Akond Ashfaque Ur, Eric Helms, Laurie Williams, and Chris Parnin. "Synthesizing continuous deployment practices used in software development." In *2015 Agile Conference*, pp. 1-10. IEEE, 2015.

[39] Shahin, Mojtaba, Muhammad Ali Babar, Mansooreh Zahedi, and Liming Zhu. "Beyond continuous delivery: an empirical investigation of continuous deployment challenges." In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 111-120. IEEE, 2017.

[40] C. Wohlin, M. Höst, and K. Henningsson, “Empirical research methods in software engineering,” in Empirical methods and studies in software engineering, Springer,

2003, pp. 7–23

[41] J. J. Carroll, P. Anand and D. Guo, "Preproduction Deploys: Cloud-Native Integration Testing," 2021 IEEE Cloud Summit (Cloud Summit), 2021, pp. 41-48, doi: 10.1109/IEEECloudSummit52029.2021.00015.

[42] www.snapt.net “load-balancing-for-blue-green-rolling-and-canary-deployment” [Online]
Available : https://www.snapt.net/blog/load-balancing-for-blue-green-rolling-and-canary-deployment
[Accessed On: 1 October 2022]

[43] www.koyeb.com “blue-green-rolling-and-canary-continuous-deployments-explained ” [Online]
Available: https://www.koyeb.com/blog/blue-green-rolling-and-canary-continuous-deployments-explained
[Accessed On: 1 October 2022]

[44] www.plutora.com “deployment-strategies-6-explained-in-depth”
Available: https://www.plutora.com/blog/deployment-strategies-6-explained-in-depth
[Accessed On: 1 October 2022]

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden