

# **Automating Deployment of Various Microservices Kubernetes**

Project report submitted in partial fulfillment of the  
requirement for the degree of Bachelor of Technology

in

**Computer Science and Engineering/Information  
Technology**

By

Divyansh Joshi (191353)

Under the supervision of

Dr. Hari Singh Rawat

to



Department of Computer Science & Engineering and  
Information Technology

**Jaypee University of Information Technology**  
**Waknaghat, Solan-173234, Himachal Pradesh**

## Candidate's Declaration

I hereby declare that the work presented in this report entitled “**Automating Deployment of Various Microservices in Kubernetes**” in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science and Engineering/Information Technology** submitted in the department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from July 2022 to May 2023 under the supervision of **Dr. Hari Singh Rawat**, who is currently an Assistant Professor (SG) Department of Computer Science.

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Divyansh Joshi, 191353

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

Dr. Hari Singh Rawat  
Assistant Professor (SG)  
Department of Computer Science  
Dated:

**JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT**  
**PLAGIARISM VERIFICATION REPORT**

Date: .....

Type of Document (Tick):  PhD Thesis  M.Tech Dissertation/ Report  B.Tech Project Report  Paper

Name: \_\_\_\_\_ Department: \_\_\_\_\_ Enrolment No \_\_\_\_\_

Contact No. \_\_\_\_\_ E-mail. \_\_\_\_\_

Name of the Supervisor: \_\_\_\_\_

Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters): \_\_\_\_\_

\_\_\_\_\_

**UNDERTAKING**

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

**Complete Thesis/Report Pages Detail:**

- Total No. of Pages =
- Total No. of Preliminary pages =
- Total No. of pages accommodate bibliography/references =

(Signature of Student)

**FOR DEPARTMENT USE**

We have checked the thesis/report as per norms and found **Similarity Index** at .....(%). Therefore, we are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

(Signature of Guide/Supervisor)

Signature of HOD

**FOR LRC USE**

The above document was scanned for plagiarism check. The outcome of the same is reported below:

Copy Received on	Excluded	Similarity Index (%)	Generated Plagiarism Report Details (Title, Abstract & Chapters)	
	<ul style="list-style-type: none"> <li>• All Preliminary Pages</li> <li>• Bibliography/Images/Quotes</li> <li>• 14 Words String</li> </ul>		Word Counts	
Report Generated on			Character Counts	
		<b>Submission ID</b>	Total Pages Scanned	
			File Size	

Checked by  
 Name & Signature

Librarian

.....

**Please send your complete thesis/report in (PDF) with Title Page, Abstract and Chapters in (Word File) through the supervisor at [plagcheck.juit@gmail.com](mailto:plagcheck.juit@gmail.com)**

## **Acknowledgement**

Firstly, I express my heartiest thanks and gratefulness to almighty God for his divine blessing makes it possible for me to complete the project work successfully.

I am really grateful and wish my profound indebtedness to Supervisor **Dr. Hari Singh, Assistant Professor**, Department of CSE Jaypee University of Information Technology, Wakhnaghat whose guidance was the most valuable in order to complete this project. I would also like to thank Mr. Sukhneer Singh Guron, my manager who took me under his wing to work and cultivate knowledge in all domains.

I would also generously welcome each one of those individuals who have helped me straightforwardly or in a roundabout way in making this project a win. In this unique situation, I want to thank the various staff individuals, both educating and non-instructing, which have developed their convenient help and facilitated my undertaking.

Finally, I must acknowledge with due respect the constant support and patients of our parents.

Divyansh Joshi

191353

## Table of Content

<b>Title</b>	<b>Page Number</b>
<b>Candidate's Declaration</b>	I
<b>Plagiarism Certificate</b>	II
<b>Acknowledgement</b>	III
<b>Table of Content</b>	IV
<b>List of Figures</b>	V
<b>Abstract</b>	VI
<b>Introduction</b>	1-13
<b>Literature Review</b>	14-23
<b>System Design and Development</b>	24-44
<b>Experiments and Result Analysis</b>	45-49
<b>Conclusion</b>	50-53
<b>References</b>	54-55

## List of Figures

<b>Figure Number</b>	<b>Description</b>
<b>1</b>	Microservice Architecture vs Monolithic Architecture
<b>2</b>	Containerized Applications
<b>3</b>	Container Orchestration
<b>4</b>	Init Container Example
<b>5</b>	Jenkins Pipeline Flow
<b>6</b>	kubectl patch command
<b>7</b>	Memory limit error
<b>8</b>	Base Kustomization template
<b>9</b>	Custom template for kustomize
<b>10</b>	Overlay Kustomization
<b>11</b>	kubectl apply -k
<b>12</b>	Bash shebang
<b>13</b>	Shell shebang
<b>14</b>	Init Script Testing
<b>15</b>	Init Script Logs in a Cluster
<b>16</b>	Running Application UI
<b>17</b>	Running Workspaces UI

## **Abstract**

This abstract highlights the deployment process of various microservices which comprise our application “Workspaces”, including essential resources such as MongoDB, Kafka, and Ingress Nginx Controller, within a local minikube Kubernetes[1] cluster. The procedure involved containerizing applications using custom Dockerfiles and employing init containers to execute precheck tasks, such as verifying resource availability and importing prerequisite data. Subsequently, Kubernetes manifests were meticulously written for each microservice, encompassing resources like Deployment, Ingress, Service, and ConfigMap. A comprehensive "setup.sh" script facilitated the execution of the entire application stack, with separate scripts for managing resources and microservice apps. Configuration for different cluster environments, such as minikube[2], Azure AKS, and GKE, was achieved through Kustomize, leveraging new kustomization.yaml files and patches. Environment detection mechanisms were implemented for dynamic command execution. Furthermore, the setup was made accessible to external users through a separate Haproxy service, with the proper configuration of the haproxy.cfg file. Rigorous testing, debugging, and problem-solving were undertaken to ensure successful automation. The resulting solution runs independently on any development server equipped with a minikube cluster.

# Chapter-1

## INTRODUCTION

### 1.1 Introduction to Microservices, containers and container orchestration

Microservices are a common architectural approach for efficiently and scalable development of complex software systems. The concept that complex monolithic programs can be broken down into more manageable, loosely coupled services that can be developed and deployed independently forms the basis of microservices architecture. Each service controls a certain functionality and communicates with other services via protocols that are standardized across the industry, such as HTTP or REST.

Microservices are not a new concept; they have been around since the early 2000s. However, it has lately gained popularity as software development teams began to recognize the drawbacks of the traditional monolithic architecture. The monolithic approach was distinguished by a closely coupled design, where each module of the program was intricately connected to every other element. It was difficult to update one component of the program without simultaneously changing the others as a result. Large applications are difficult to develop, test, deploy, and manage as a result.

On the other hand, there are a number of advantages that microservices provide over the conventional monolithic architecture. First of all, they enable greater adaptability and agility in software development. The ability to develop and deploy each microservice independently of the others enables more frequent delivery of new features and updates in order to decrease time-to-market and enhance user experience. Microservices offer improved scalability since each service can be scaled separately to meet the changing requirements of the application. This method also helps to increase fault



tolerance because the failure of one service does not always mean that the entire application will also fail.

The microservices architecture also enhances the code's quality and maintainability. Each service focuses on a certain functionality, making the code simpler and easier to understand. This enables new developers to onboard more quickly, and it also makes it easier to find and fix issues. However, implementing a microservices architecture is not without its challenges. One of the main challenges is managing the complexity of a distributed system. Microservice designs can be challenging to administer, requiring specialized infrastructure and tools. Service discovery, inter-service communication, and ensuring data consistency between services are other challenges.

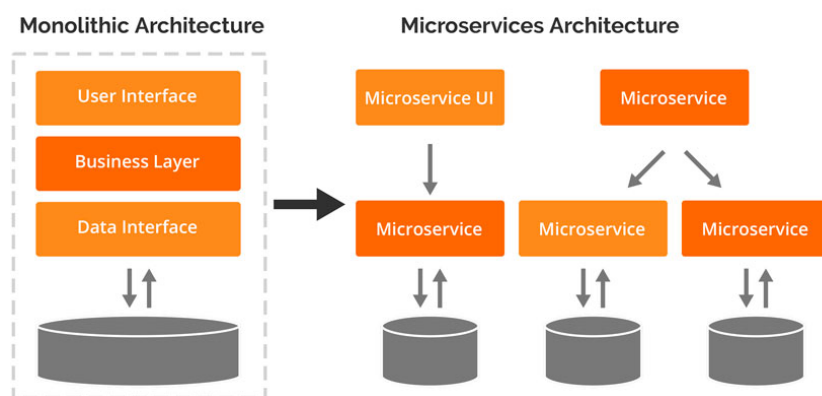


Fig-1 Microservice Architecture vs Monolithic Architecture

Microservices have largely altered the creation, implementation, and management of software systems. They give teams a more flexible and scalable way to create software, allowing them to respond to changing business needs more quickly. As this technology continues to grow, we might expect to see more improvements in microservices architecture and development methods.

Container technology has become a key component of modern software application development and deployment, especially when leveraging microservices architecture. When programs are contained within a container, a portable, lightweight, and self-contained piece of software that can run on any device or operating system, it is easier to bundle, distribute, and execute apps consistently across many contexts.

Containerization is ideal for microservices architecture due of its benefits. First off, containers provide an isolated environment for executing programs, enabling them to work consistently and stably regardless of the underlying infrastructure. As a result, moving programs across contexts—such as from development to production—is made easier without having to worry about infrastructure changes or incompatibilities. Additionally, containers offer a simpler and more efficient way to manage dependencies and configurations because they may bundle an application with all necessary dependencies and parameters. As a result, developers may now deploy an application anywhere without having to worry about installing and configuring various dependencies on numerous machines.

Another benefit of containers is that they offer better resource use and scalability. Running a lot of containers on one machine won't affect performance because each container is small and light. By adding more containers as demand increases, this reduces total infrastructure costs and makes scaling an application easier.

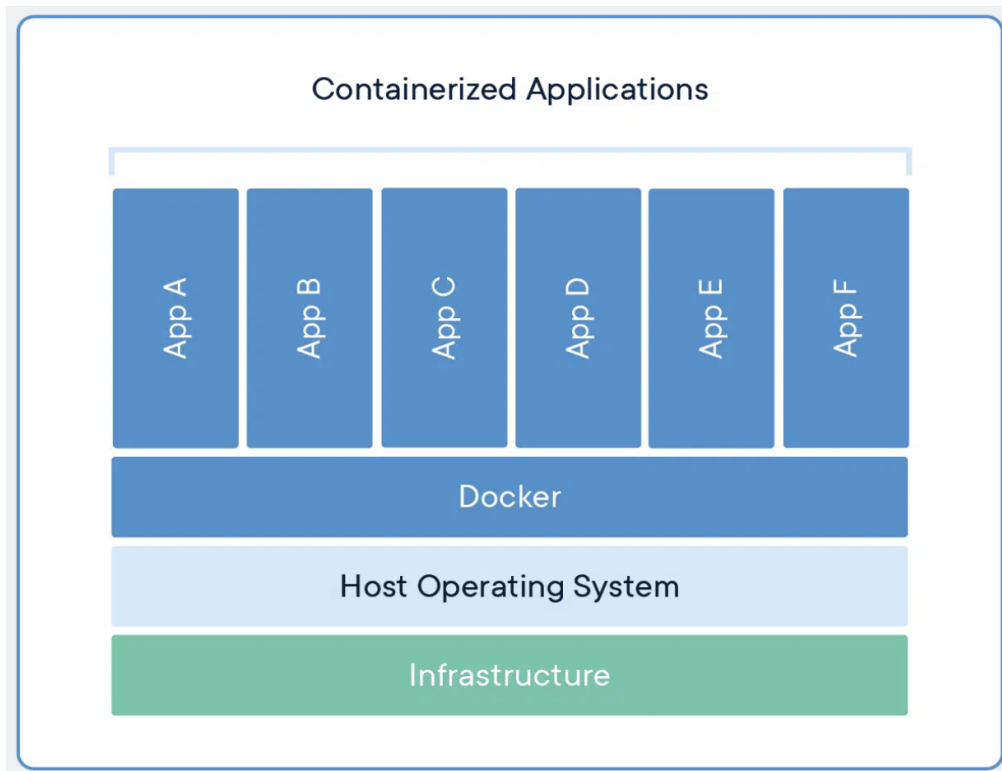


Fig-2 Containerized Applications

Containerization enables a more agile development process, which is essential for microservices architecture. Adjustments may be made and new features can be deployed more quickly and efficiently since one microservice can be developed and supplied independently of the others. Additionally, this approach reduces the time and effort required to test the entire application by making it easier to test and debug certain services.

For the creation of a microservices architecture, containerization technology is essential. Now that containers can be used to package, deploy, and run microservices, scalability, efficiency, and agility are all improved. We may predict that as microservices architecture continues to gain popularity, containerization will become more and more crucial for creating and delivering contemporary software applications.

A software element that may execute containers on a host operating system is a container runtime, commonly referred to as a container engine. Container runtimes are in charge of loading container images from a repository, keeping track of local system resources, isolating system resources for use by a container, and managing the lifespan of containers in a containerized architecture. RunC, containerd, Docker[3], and Windows Container are typical examples of container runtimes.

Common container runtimes typically work with container orchestrators. The orchestrator must deal with networking, security, and container scalability challenges in order to manage container clusters. The container engine is responsible for managing each individual container that is executing on each computing node in the cluster.

Kubernetes, a ground-breaking open-source container orchestration platform, has revolutionized how modern applications are currently deployed and maintained. It was initially developed by Google and then contributed to the Cloud Native Computing Foundation (CNCF), where it is now maintained by a large and active community of developers. This is based on Google's experience managing significant containerized workloads in production.

With the help of distributed infrastructure, Kubernetes automates the deployment, scaling, and management of containerized applications. Regardless of whether their apps are operating in a public cloud, a private data center, or a hybrid cloud environment, developers can quickly deploy and manage them with Kubernetes in a variety of scenarios. Its adaptable and potent design makes it possible to manage the whole application lifecycle, from development to production and beyond.

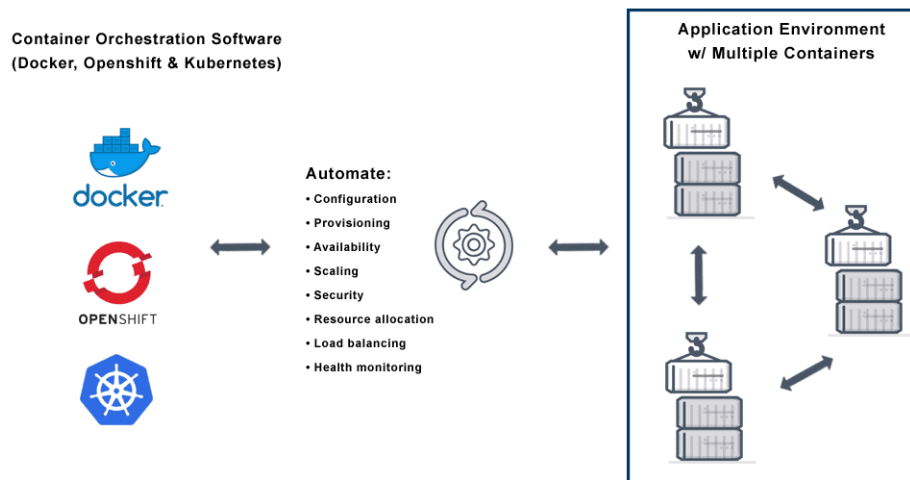


Fig-3 Container Orchestration

Kubernetes has a lot of potent characteristics that make it a desirable choice for the deployment and development of contemporary applications. It has built-in load balancing and service discovery features, for instance, which make it simple to expose containerized applications to the outside world. It also has a strong networking stack that makes it simple to connect containers together. Additionally, it has strong storage features that make it simple for developers to control persistent storage for their applications.

Overall, Kubernetes is used by a variety of companies, ranging in size from small startups to major enterprises, and has quickly established itself as the industry standard for container orchestration. Its popularity is a result of its capacity to facilitate developers' containerized application deployment and administration, freeing them up to concentrate on developing code rather than worrying about infrastructure. Kubernetes is positioned to play a significant part in the development and deployment of applications in the future thanks to its versatile and potent platform.

## 1.2 Problem Statement

In our application, currently only a few of the application's microservices are currently correctly containerized, but the business is not making full use of Kubernetes and the microservices architecture to speed up and streamline the development and deployment process. Currently, these microservices are deployed manually, which might cause a number of problems.

Because manual deployment of containerized microservices requires a lot of manual labor and can result in discrepancies between multiple deployments, it can be a time-consuming and error-prone operation. As a result, developers and operations teams must spend time manually installing and configuring each microservice, which can cause delays and raise expenses. Manual microservice deployment might result in security and compliance challenges in addition to time and financial problems. It might be challenging to guarantee that each deployment complies with relevant laws and standards and is secure without the right automation and configuration management technologies.

Additionally, manual deployment may restrict scalability and impede speedy modification. Since each deployment must be manually configured and controlled, scaling up or down quickly can be challenging with manual deployment. This may make it more difficult to adapt swiftly to shifts in demand or other demands.

The company is losing out on greater agility, scalability, and efficiency by not fully embracing Kubernetes and microservices design. While microservices design facilitates application modularization and decoupling, making it simpler to manage and scale individual components, Kubernetes offers a reliable and scalable platform for the deployment and management of containerized applications.

The company might spend money on a DevOps pipeline that uses Kubernetes and microservices architecture to handle this problem statement. This would

entail automating the deployment process with the help of tools like Helm, Kubernetes Operators, and GitOps, which can make the deployment process easier and more efficient. These tools allow for quicker iteration and scalability while also ensuring that each deployment is secure and consistent.

The problem statement underlines the need for organizations to fully utilize Kubernetes and microservices architecture in order to enable speedier and more efficient development and deployment of containerized applications. Businesses can gain from doing this by increasing their efficiency, scalability, and agility as well as cutting expenses, improving security, and complying with regulations.

### **1.3 Objectives**

For our application called “Workspaces” the fundamental idea of this project is to containerize each and every component of the application so as to convert it into a microservice. Further down the line, we’re aiming to use Kubernetes to create various manifests and configurations for the deployment of all the required microservice at one place in order to make the deployment faster and more efficient. Other than that, the main goal of this project is to deploy this entire application end to end on a local machine or a development server, so that any developer in the team could conveniently deploy the entire application stack locally and perform their development changes, debug and test the application at the same time.

A challenging part of creating this setup is automation. The use of tools and technology to automate various processes involved in the software development and deployment process is referred to as automation. It is a crucial DevOps practice since it aids enterprises in accelerating, streamlining, and standardizing their software development and deployment procedures. Code testing, deployment, monitoring, and maintenance are just a few of the parts of the software development lifecycle that may be automated, freeing

engineers to work on higher-value jobs and increasing output. Organizations may decrease errors, enhance quality, and produce software more rapidly and consistently by automating repetitive and time-consuming operations.

The most part of automation in our codebase is done through writing efficient and crisp manifests in launching Kubernetes resources and using various bash scripts in order to automate the entire flow of deployment. Furthermore, in Kubernetes a singular node cluster we can use on a local machine or a development server is called “minikube”. We’ll be using that to launch the entire application stack.

Minikube is a lightweight and easy-to-use Kubernetes distribution that allows developers to test and experiment with Kubernetes without the need for a full-scale production environment. With Minikube, it is possible to deploy various types of containers, including Docker, Kubernetes, and other containerized applications.

#### **1.4 Methodology**

The methodology for the "Workspaces" application project involves containerizing every component of the application, utilizing Kubernetes to create manifests and configurations for deployment, and automating various processes involved in the software development and deployment process.

Software code and all of its dependencies are bundled together into a single package known as a container through the process of containerization. Software applications can be deployed more quickly, consistently, and flexibly thanks to this containerized technique. The "Workspaces" project seeks to produce microservices that can be independently created, deployed, and scaled by containerizing each component of the application.



The "Workspaces" project uses Kubernetes to build different manifests and configurations for the deployment of all necessary microservices in a single location, which will speed up and improve deployment. Developers may focus on activities with a greater return on investment by having Kubernetes launch, scale, and manage containers for them automatically.

Enterprises may expedite, streamline, and standardize their software development and deployment processes by implementing automation, a critical DevOps strategy. Developers can boost production, reduce errors, and improve software quality by automating numerous activities involved in the software development and deployment process. The "Workspaces" project wants to launch Kubernetes resources with efficient and clear manifests, bash scripts, and numerous operations that can be automated.

To automate various processes involved in the software development and deployment process, we will:

- Write efficient and crisp manifests in launching Kubernetes resources.
- Use various bash scripts in order to automate the entire flow of deployment.

Minikube is a single-node Kubernetes cluster that can be run on a local machine or a development server. The "Workspaces" project intends to make it simple for every developer on the team to deploy the application stack locally and perform concurrent development modifications, debugging, and testing by leveraging Minikube to launch the whole application stack. As a result, testing and development will proceed more quickly and effectively, giving developers more time to find and fix any application problems.

The goal of the "Workspaces" application project was to change the program into a microservice architecture by containerizing each of its parts. The ultimate goal is to deploy the complete application stack end-to-end on a local workstation or a development server, making it easier for any developer on the

team to deploy the application stack locally and perform concurrent development modifications, debugging, and testing.

The team intends to achieve this using Kubernetes, a container orchestration system, which will make it possible to create the different manifests and configurations needed for the deployment of all relevant microservices in a single location. This will not only speed up and increase the efficiency of the deployment process, but also make it more standardized and manageable.

Automation, though, is a big problem for this endeavor. Automation is a crucial DevOps approach that uses tools and technologies to streamline a variety of software development and deployment processes. Engineers may concentrate on higher-value operations by automating repetitive and time-consuming procedures, which boosts the software development lifecycle's overall quality and efficiency.

The team will use a variety of methods to automate the deployment process in order to overcome this difficulty. This entails producing scripts to automate the entire deployment process, designing clear and efficient manifests for launching Kubernetes resources, and using a single Kubernetes cluster, known as "minikube," to launch the full application stack.

We can achieve our objective of deploying the entire application stack end-to-end on a local machine or a development server by putting these strategies into practice. This will enable all team members who are developers to test and develop changes quickly and effectively, boosting output and quality.

## **1.5 Organization**

At the beginning of our project, we had already containerized some of the larger APIs and UI components of our application. However, further development and testing needed to be done on a development server with a

Linux operating system, rather than on individual developer laptops. This was partly due to the complexity of deploying certain containerized applications on laptops with the ARM64 architecture.

In order to run all the necessary containerized parts of our application stack, we had to put up a special development server. The relevant dependencies and container management tools, including Docker and Kubernetes, had to be installed on this server. Additionally, we had to make sure the server had enough resources to meet the demands of operating several containers at once.

When the server was set up and the application stack was deployed there, developers had remote access to it, allowing them to perform testing, debugging, and development updates. This approach also ensured that the software would function properly when it was released into production by providing each team member with access to a uniform development environment. We are able to connect to a remote development server using SSH. Secure access to a remote computer's resources, including its files and applications, is provided by the SSH network protocol. It offers a safe, encrypted connection between the client and server to thwart illegal access and data theft.

In order to access a development server via a terminal, one must first acquire the IP address or domain name of the remote machine in order to connect using SSH. Once you are aware of this, you can connect to a remote machine using your terminal or command prompt by using the SSH command along with the machine's IP address or domain name.

An SSH command may look like this:

```
ssh username@192.168.1.100
```

SSH is essential for remote access and administration of Linux machines, as it provides a secure and encrypted way to connect to the machine and perform tasks without physically being present at the machine. This is particularly

useful for system administrators who need to manage multiple machines remotely, as it saves time and effort by eliminating the need to physically travel to each machine.

## **Chapter-2**

### **LITERATURE REVIEW**

The use of containerization technologies has significantly increased recently, particularly in the context of analyzing scientific data. Kubernetes is one such container orchestration software that has grown to be a popular choice for managing containers. The paper [4]"Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis," by the authors "Anton Tesliuk", "Sergey Bobkov" which investigates the benefits of utilizing Kubernetes for scientific data analysis as well as some of its practical implementations, will be the main focus of this research of the literature.

Launching and managing containerized apps using the open-source Kubernetes technology for container orchestration. For managing, scaling, and automating the deployment of containerized applications, it provides a strong basis. Kubernetes is widely used in business and academia, and its acceptance has been growing quickly over time. The study "Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis" discusses the benefits of using Kubernetes for scientific data analysis. According to the authors, Kubernetes provides a flexible and scalable framework for handling complex scientific data analysis processes.

Automating the deployment and management of containerized programs is one of the main benefits of using Kubernetes for scientific data processing. This frees researchers from having to worry about the infrastructure that supports them so they may concentrate on their data analysis responsibilities. Multiple teams working on the same project can easily share containers thanks to Kubernetes' centralized container management approach.

The advantages of using Kubernetes for this activity are covered in this research thoroughly. These are discussed in further detail below.

Automating the installation and upkeep of containerized applications: Kubernetes offers a standardized method of managing containers that can be readily shared among numerous teams working on the same project. This frees researchers from having to worry about the infrastructure that supports them so they may concentrate on their data analysis responsibilities.

- Large-scale scientific data analysis workflows can be managed using Kubernetes, a distributed platform for container management that can be used to expand data analysis processes across several nodes. This can speed up the analysis process overall and cut down on the time needed to evaluate massive datasets.
- Using containerization technology, researchers can design environments that can be easily shared and reproduced by others. This might encourage the exchange of data and knowledge in the sciences and strengthen teamwork among researchers.
- Data analysis operations can be scaled across several nodes using Kubernetes, which offers a distributed approach for managing containers. By doing so, the length of time needed to examine huge datasets can be shortened, and the overall effectiveness of the analysis process can be increased.
- Kubernetes offers a framework that is both scalable and flexible, making it possible to manage intricate scientific data analysis procedures. It is simple to adapt Kubernetes to the unique requirements of various scientific data processing approaches.

The paper "Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis" lists several use cases for Kubernetes in scientific data analysis. These usage scenarios demonstrate how Kubernetes can be scaled up and down to control complex data analysis processes.

Large-scale genomics data analysis pipeline: Kubernetes is utilized in this study to manage a large-scale genomics data analysis pipeline. There are many procedures in the pipeline, including variant calling, alignment, and data preprocessing. Using Kubernetes, the authors were able to automate the pipeline's deployment and administration, which cut down on the time required for data processing and improved the process' overall efficacy.

process for image classification using machine learning: Kubernetes is employed in the article to control the process for image classification using machine learning. The workflow entails a number of processes, including model training, model evaluation, and data preprocessing. The authors were able to install and manage the workflow automatically thanks to Kubernetes, which increased model accuracy and cut down on model training time.

process for high-throughput screening: Kubernetes is used in the essay to manage a process for high-throughput screening. The workflow consists of a number of steps, including feature extraction, data preprocessing, and statistical analysis. The authors were able to automate the workflow deployment and management using Kubernetes, which reduced the amount of time needed for data processing and increased the overall effectiveness of the analytic process.

A scalable and flexible platform is offered by Kubernetes for managing difficult scientific data analysis tasks. This allows researchers to concentrate on their data analysis activities without having to worry about the supporting infrastructure. Adopting Kubernetes for data analysis has many advantages, including reproducible research, flexibility and scalability, automation of containerized application deployment and maintenance, and management of enormous scientific data processing workflows. The report also covers Kubernetes use scenarios for complex data analysis workflow management, including as high-throughput screening procedures, machine learning workflows for image classification, and substantial pipelines for processing genomics data. These use cases show how adaptable and scalable Kubernetes is for controlling these processes. Overall, Kubernetes is a useful tool for

analyzing scientific data, and its use in the scientific community is expected to increase in the coming years.

In recent years, there has been a substantial increase in the adoption of containerization technologies for software development and deployment. Solutions for container orchestration are now a popular way to manage the complexity of containerized systems. The Google publication "Borg, Omega, and Kubernetes: Lessons learned from three container management systems over a decade,"[5] which provides details on the development of the three container orchestration platforms Borg, Omega, and Kubernetes, will be the main focus of this assessment of the literature.

Google developed the first platform for container orchestration named Borg at the start of the new millennium. Borg was developed to handle the massive scale of Google's data centers and is used to deliver and manage containerized applications. Borg included features including fault tolerance, workload management, and resource separation. Omega was developed as a successor to Borg, and was designed to address some of the limitations of Borg. Omega was a more flexible and modular platform that improved scheduling and resource management. Omega also provided features such as support for custom resource types and easier integration with external systems.

Kubernetes was developed via Google in 2014 as an open-supply field orchestration platform based totally on the instructions found out from Borg and Omega. Kubernetes presents a platform for automating deployment, scaling, and control of containerized programs. Kubernetes has grown to be extensively followed within the enterprise and academia and has a large and lively community of builders and users.

The scholarly exposition titled "Borg, Omega, and Kubernetes: Lessons gleaned from the triumvirate of container management systems spanning a decade" presents several sagacious insights into the progression of container orchestration platforms. These insights encompass:



The paramountcy of resource management:

One of the cardinal takeaways from the evolution of Borg, Omega, and Kubernetes is the indispensability of resource management. Container orchestration platforms must possess the capability to adroitly administer resources in order to ensure the optimal deployment and execution of applications.

Borg was the inaugural container orchestration platform incubated by Google, conceived to steward the colossal scale of Google's data centers. It furnished features like workload management, resource sequestration, and fault tolerance. An overarching takeaway from the development of It was the indispensability of resource management. Container orchestration platforms ought to have the capacity to manage resources with adroitness to ensure that applications are deployed and executed optimally. While it was efficacious in administering the resources of Google's data centers, it was not impervious to certain limitations.

Omega was forged as a successor to Borg, with the intention of rectifying some of Borg's shortcomings. Omega was a more pliable and modular platform that enabled superior utilization and scheduling of resources. Omega also furnished features like bolstered provision for bespoke resource types and a more facile integration with external systems. One of the key takeaways from the evolution of Omega was the necessity for modularity and flexibility in container orchestration platforms. Platforms must be capable of accommodating dynamic exigencies and furnishing a pliable and extensible framework that can be tailored to satisfy the specific requirements of diverse applications.

An outstanding advantage of Kubernetes is its provenance as an open-source platform. This attribute has fostered a prodigious and engaged community of developers and users, who have participated in the advancement and refinement of the platform. The open-source character of Kubernetes has also

culminated in the evolution of an opulent ecosystem of tools and services that can be leveraged to augment the functionality and usability of Kubernetes.

Lessons learned from three container management systems over a decade" provides myriad insights into the evolution of container orchestration platforms. These takeaways encompass:

The criticality of resource management:

One of the pivotal lessons gleaned from the evolution of Borg, Omega, and Kubernetes is the primacy of resource management. Container orchestration platforms must be adept at managing resources efficiently to ensure the optimal deployment and execution of applications.

The indispensability of modularity and flexibility:

Another key takeaway is the indispensability of modularity and flexibility in container orchestration platforms. Platforms must be able to acclimate to changing requirements and furnish a flexible and extensible framework that can be tailored to satisfy the unique demands of different applications.

The weightiness of fault tolerance:

Container orchestration platforms must be fault-tolerant and provide mechanisms for identifying and recuperating from failures. This is especially consequential for large-scale systems where glitches can significantly impact the availability and reliability of applications.

The benefits of open-source development:

The evolution of Kubernetes as an open-source platform has spawned a mammoth and dynamic community of developers and users. This has culminated in the growth of an opulent ecosystem of tools and services that can be harnessed to enhance the functionality and usability of Kubernetes.

Containerization technologies have witnessed a tremendous surge in popularity in recent years, with Kubernetes emerging as one of the most widely adopted container orchestration platforms. In the present literature review, we shall scrutinize the paper "A Formal Model of the Kubernetes Container Framework"[6] authored by Gianluca Turin and Andrea Borgarelli. This paper is remarkable for providing a highly sophisticated and formalized model of the Kubernetes container framework. The model is meticulously constructed and analyzes the intricate workings of Kubernetes in immense detail.

The paper commences with an in-depth overview of the Kubernetes container framework, elucidating the crucial components that make it functional, such as pods, nodes, and services. After that, the formal paradigm they developed for Kubernetes is being discussed. As the basis for their model, they employed the pi-calculus, a process calculus renowned for its precision in characterizing concurrent systems. The formal model in the paper offers a comprehensive foundation for expressing the generic behavior of the Kubernetes system in a precise and methodical manner. The authors then thoroughly evaluate Kubernetes' performance using the paradigm, highlighting any security flaws as well as its benefits and drawbacks.

After giving a full explanation of the formal model's development in the Maude language, an important tool for formal verification and analysis of concurrent systems, the work is noteworthy for its thorough explanation of this process. The authors exhibit the efficiency of the implementation in validating a variety of key Kubernetes system properties, including as liveness and deadlock-freedom. Numerous significant contributions are made in the book that are essential to the formal analysis and study of container orchestration platforms. One of the primary contributions is a highly developed and sculpted model of the Kubernetes container system; enabling a detailed and rigorous analysis of its behavior and capabilities. The Kubernetes system's possible issues and weak areas are identified using the normal model, which may then be rectified to improve the system's overall security and dependability. This study also demonstrates the effectiveness of the Maude language, which can

be used with a variety of distributed systems and container orchestration platforms, for formal verification and analysis of concurrent systems.

An in-depth analysis of the Kubernetes container framework is provided in the paper "A Formal Model of the Kubernetes Container Framework." The article provides a formal model of the system in order to identify potential issues and weaknesses that may be rectified to improve the system's overall reliability and security. The use of their language for formal verification and analysis of concurrent systems has also been demonstrated to be quite effective. The paper's invaluable insights into the development and application of container orchestration platforms can serve as a blueprint for new systems. Overall, the work has been extremely beneficial to the study of container orchestration systems.

The research paper "Microservices: Architecture, Container, and Challenges" by authors– "Guozhi Liu" and "Bi Huang" offers a complete analysis of the microservices architecture, covering its essential features, advantages, and disadvantages as well as its relationship with container technologies. Internal architecture of Kubernetes and how it came to rise is described in length. The paper starts off by providing a full introduction to the microservices architecture, emphasizing its unique features including the use of compact, independent and deployable services as well as its ability to expand individual services separately. Benefits of microservices are also described in quite detail, which include increased flexibility and scalability.

After discussing the heavy details and intricacies of the Kubernetes architecture, the research continues by looking at container technologies, which are often used for managing and delivering microservices. This work also describes the process and internal computations of containers. The authors provide an in-depth analysis of container technology, taking into account its evolution and historical background. As well as being emphasized are the advantages of utilizing containers to deploy microservices, including the effective use of resources, easier deployment and scaling.

The study paper then investigates the microservices architectural issue. These challenges are listed by the authors and include dealing with security issues, guaranteeing consistency, and successfully managing the complexity of a distributed system. In this study, some particular solutions are offered to these problems including the usage of load balancing and service discovery techniques. The article also looks at potential future developments in container technology and microservices design. As the adoption of microservices increases, the authors claim that container technologies will become even more important for the deployment and management of microservices.

- It provides a thorough overview of the main characteristics and advantages of microservices architecture.
- Second, it gives a general introduction of container technologies and how they may be used to deploy and manage microservices, highlighting their advantages in terms of efficient resource consumption and deployment simplicity.
- Managing the complexity of a distributed system, guaranteeing consistency and stability, and resolving security issues are just a few of the difficulties that come with microservices design that are covered in detail in this paper. The authors provide approaches for dealing with these difficulties, such as the application of load balancing and service discovery methods.

Moreover, the study offers important perspectives on the development of container technologies and microservices architecture. The authors predict that as container technologies become progressively more crucial for the deployment and management of microservices, the popularity of microservices architecture will increase. Overall, this paper contributes

significantly to the fields of microservices design and container technologies by offering in-depth research, tactical answers to problems, and perspectives on the direction of the industry.

## Chapter 3 - System Design and Development

A microservice-based application architecture consists of numerous independent services that interact with one another via a messaging system, such as Kafka, and store data in a database system, such as MongoDB. Before initiating a microservice, the init container for each microservice checks that the required resources are available.

The following gives a high-level explanation of the system design for such an application:

**Service discovery:** Service discovery allows microservices to find and communicate with one another without the clients being aware of where specific services are located. Utilizing a service registry or discovery service like Consul[7] or etcd can help with this. A microservice registers with the service registry when it first starts up, providing information about its location, state, and other metadata. By using a registry query, other microservices can then locate and get in touch with the required service.

**Messaging System:** For the development of distributed applications, Kafka is a well-known messaging platform. For asynchronous microservice communication, it provides a scalable, fault-tolerant, and high-performance method. Each microservice is capable of publishing to and subscribing to a single or several Kafka topics, which act as message queues. For every message that a microservice publishes to a topic, Kafka constructs a partition to store it and makes it available to all subscribers. Afterward, the subscribers can respond to the message as necessary. Since it offers features like message preservation, splitting, and replication, Kafka is a strong option for creating systems that are designed to be used on a wide scale.

**Data storage:** The NoSQL database system MongoDB[8] offers a scalable and flexible solution to store and retrieve the data needed by the microservices. Using the appropriate driver or ORM library, any microservice can communicate with MongoDB. MongoDB is a popular option for developing contemporary applications because it provides features like document-oriented storage, horizontal scaling, automatic sharding, and a sophisticated query language.

**Container Orchestration:** To control the deployment and scaling of microservices, a container orchestration system like Kubernetes can be employed. Microservices can be packaged as Docker containers and installed on a node cluster using Kubernetes. It offers functions that make managing big microservice applications simple, like automatic scaling, rolling updates, service discovery, load balancing, and health checks. Following is an example of a Kubernetes pod manifest.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Fig 3.2 - Pod Manifest

**Init containers[9]:** Containers that conduct a single-time job before the main container starts are known as init containers. An init container can be used in a microservice-based application to determine whether the necessary resources,



such as Kafka and MongoDB instances, are available before the microservice launches.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app.kubernetes.io/name: MyApp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep
3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup myservice.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc
.cluster.local; do echo waiting for myservice; sleep 2;
done"]
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup mydb.$(cat
/var/run/secrets/kubernetes.io/serviceaccount
```

Fig-4 Init Container Example

The microservice container is started after the init container has executed a Bash script that verifies the readiness of the necessary resources. This makes it easier to make sure the microservice is only launched when it can effectively communicate with the other services on which it depends.

Monitoring and logging: Metrics and logs from each microservice and Kafka instance can be gathered using a monitoring system like Prometheus. The monitoring system gives administrators immediate visibility into the functionality and performance of the system and notifies them of any problems. Key metrics including CPU use, memory usage, network traffic, and request delay may be tracked using Prometheus. Additionally, it can be utilized to produce warnings based on pre-established thresholds or abnormalities.

**Continuous Integration and Delivery:** The Pipeline plugin for Jenkins[10] can be used to define a CI/CD pipeline. Developers can define a pipeline as code with this plugin, and both the pipeline and the application code can be checked into source control.

Typically, a Jenkins pipeline is made up of stages that correspond to the various CI/CD process processes, such as building, testing, deploying, and releasing the application. One or more steps, which can be shell commands, scripts, or Jenkins plugins, can be included in each stage.

With Jenkins, the pipeline may be defined as follows to construct a CI/CD pipeline for a microservice-based application:

Typically, a Jenkins pipeline is made up of stages that correspond to the various CI/CD process processes, such as building, testing, deploying, and releasing the application. One or more steps, which can be shell commands, scripts, or Jenkins plugins, can be included in each stage.

With Jenkins, the pipeline may be defined as follows to construct a CI/CD pipeline for a microservice-based application:

- Checkout: During this phase, the source code is downloaded from the Git repository.
- Build: Using a Dockerfile, this stage creates the Docker image for the microservice. The runtime environment, dependencies, and application code may all be included in the Docker image.
- Test: During this phase, the microservice's unit, integration, and end-to-end tests are conducted. The test results can be reported using Jenkins plugins like JUnit and Cucumber.

- Deploy: Using the Kubernetes CLI (kubectl) or a Jenkins plugin like Kubernetes Continuous Deploy, this stage deploys the microservice to a Kubernetes cluster. To reduce downtime, the deployment can be performed as a rolling update.
- Release: The microservice may be put into production at this level. To construct a release package and upload it to a release repository, use a Jenkins plugin like GitHub Release or Artifactory.
- 

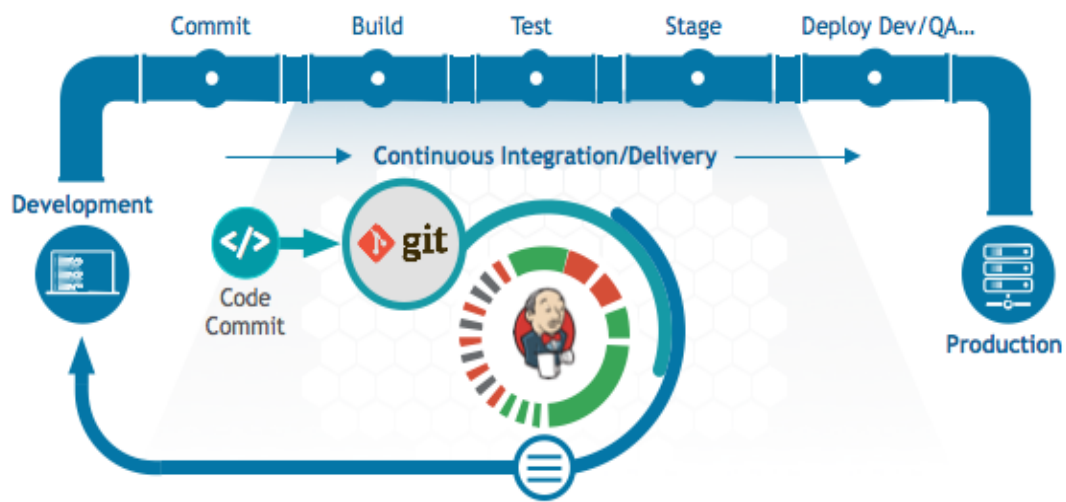


Fig-5 Jenkins Pipeline Flow

### 3.2 Development Flow

The first stage in developing the microservice-based application was to write Bash scripts for each microservice to check whether or not the required resources were available. For instance, if a microservice required Kafka and MongoDB, a Bash script was created to check their availability. A command in the deployment manifest for the particular microservice's init container was used to introduce these Bash scripts.

Prior to the main container, a separate container known as the init container runs in a pod. Its objective is to create the conditions necessary for the primary container to operate. In this case, the init container runs the Bash script to see if the required resources are available. The init container will keep running the script until it finds the resource, even if it is not active. Once the resource is available, the main container and subsequently the pod will start.

It was necessary to create the Bash script, add it to the init container of the deployment manifest, and then copy it into the container image of the relevant microservice. The Dockerfile was modified appropriately to accomplish this. If this were done, the Bash script would be available inside the container image itself. The Bash script to check the resources needed to be built as well as the suitable things to be imported into the MongoDB database. We were able to do this by utilizing the Alpine mongo tools package, which gave us access to tools like mongodump and mongoimport. Similar to the Bash script we created for resource checks, we created Bash scripts to import data into MongoDB and added them to the init container in the deployment manifest.

After the scripts were installed, the necessary services for the application had to be set up and written deployment, configmap, service, etc. manifests. Among the services provided were the NFS server, Kafka, MongoDB, and Ingress Nginx controller. Following the creation of these manifests, we had to create equivalent manifests for the application microservices. These manifests specified the expected state of the microservices, including the number of replicas, resource allocations, and other details.

We used third-party tools like Helm to simplify configuring for MongoDB and the Ingress controller. App installation and management on Kubernetes clusters are made simple with Helm, a Kubernetes package manager.

To sum up, the development pipeline entailed creating Bash scripts for the microservices, copying them into the container image, and setting up the deployment manifests for each of the necessary services. The deployment also

included the Ingress controller and NFS server. Before being put into use in a real-world setting, the scripts were tested on a local minikube environment.

The setup scripts must be defined for each microservice and resource after the manifests for the microservices have been prepared and completed. In the Kubernetes cluster, these setup scripts, which are often bash scripts, automate the process of configuring a microservice or resource. In our situation, these scripts are used to, among other things, establish users and databases for MongoDB, as well as to check whether or not the necessary resource is now active.

We use third-party tools, like as Helm[11], to establish a configuration that we can then use to create the required resources, which streamlines the process of writing these scripts. To create many users and databases for various backend services, for instance, we use Helm in the MongoDB setup script.

In a separate setup, each resource setup script is executed in a directory called "common-resources." on the same level as. This keeps the setup scripts structured and makes managing them simpler.

A top-level setup.sh script is used to boot up the whole stack for "Workspaces" at the conclusion of the setup procedure. It runs the scripts indicated above. This script first executes the setup script for common resources, which creates all of the resources needed by the application, and then it executes the setup script for the application as a whole, which merges all of the various backend and frontend microservices.

The process continues with replacing the default ingress service type needs to be changed from a LoadBalancer to a NodePort because all services in a Kubernetes environment are deployed on a single node. This is done inside of a bash script while configuring the ingress controller.

This could be done by a single kubectl command.

```
→ ~ kubectl patch svc ingress-nginx-controller -n ingress-nginx -p \
'{"spec": {"type": "NodePort", "ports": [{"protocol": "TCP", "name": "http", "port": 80, "nodePort": 30080}]}}'
```

Fig-6 “kubectl patch” command

To make a service accessible to other services within the cluster or to the outside world in a Kubernetes environment, you can either use a ClusterIP, a NodePort, or a LoadBalancer. A ClusterIP exposes the service on a cluster-internal IP address, a NodePort exposes the service on the static port of each node's IP address, and a LoadBalancer gives the service a public IP address from a third-party load balancer.

We are utilizing a NodePort to expose the service on a static port on each node in the ingress controller. As a result, using the node's IP address and NodePort, we may access the ingress controller from any node in the cluster. If the node's IP address changes, having to memorize numerous NodePorts can be troublesome. This problem can be solved by providing a single entry point for all incoming traffic using a reverse proxy, like HAProxy. HAProxy uses a single IP address and port to listen for incoming traffic, which is then forwarded based on the URL or other considerations to the appropriate location.

For our situation, we may set up HAProxy to send traffic to the ingress controller's NodePort via a single IP address and port. The ingress controller could then be accessed from a single URL, regardless of the node or NodePort it is using to operate.

Knowing which NodePort the ingress controller service has been assigned is necessary in order to configure HAProxy to send traffic there. In order to include the NodePort in our HAProxy configuration file and make sure that traffic is routed to the proper port, we declare the NodePort independently in our setup script.

Using HAProxy to provide a single entry point for incoming traffic can generally make it easier to access the ingress controller and other services in a

Kubernetes system, especially when there are many nodes and services involved suitable port. That is why here, we're specifying the nodeport separately since we want to put that inside our haproxy configuration file as well in order to access the application from a single URL.

Once the setup scripts for the custom Kubernetes resource, "workspace," were completed, the subsequent crucial step was to configure the k8s controller. This entailed cloning the controller through its corresponding repository and then using the "make deploy" command, in conjunction with the corresponding Docker image, to install the controller on our cluster. This procedure was essential to guarantee the correct creation and effective management of our custom resource.

Furthermore, a setup script for the controller also had to be written, which could streamline the process and save time in the long run. The script facilitated the configuration of the controller without the need to manually go through the setup process each time.

However, like any intricate system, several issues and challenges emerged during the setup process. One such challenge was the constant need to deploy, debug, and test the deployments repeatedly after certain updates. To solve this problem, teardown bash scripts were written for each resource and microservice, enabling us to erase the corresponding k8s resources effortlessly.

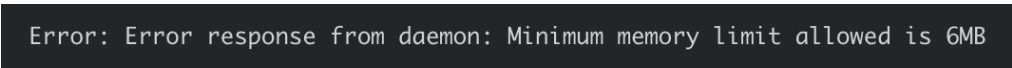
Additionally, combined scripts for the written teardown scripts were created for the "common-resources," "apps" directories, and on the top level, which further optimized the process and made it more efficient.

To optimize the application further, intermediate steps were necessary, such as creating a script to edit a cloud provider inside a static MongoDB collection. This facilitated the required modifications to the cloud provider configuration without repeating the setup process.

Another critical application that was part of this project was the "admin panel." To set up the panel, another user and database had to be created inside the MongoDB configuration, which was executed using helm. This ensured the seamless integration of the admin panel with the rest of the application and effective management.

After setting up the controller and taking all necessary steps, the entire application would boot up smoothly by running the command "kubectl apply -f workspace.yaml." This would create a "project" and enable the creation of a "workspace" inside the project, providing access to a VM in the form of an IDE such as Jupyter Lab, Ubuntu, VSCode, among others.

However, as mentioned earlier, certain challenges needed to be addressed. One such challenge was not clearly defining a memory limit for our own resource, resulting in the workspace not starting properly. A workspace was not starting and showing a very common error of "6MB memory limit" was observed for a long time.



```
Error: Error response from daemon: Minimum memory limit allowed is 6MB
```

Fig-7 Memory limit error

This issue was identified after some time and eventually resolved.

The next step was to add more microservices to each release manager and the entire application stack after finding and fixing all faults connected to the development of the workspace. "Workspaces-admin-panel" and "storinator" were two of these microservices. Precheck scripts were added to a clean Dockerfile to guarantee that the resources utilized by the application were up and running before containerizing these microservices. A Deployment, Service, ingress, and Configmap, among other resources, had to be established in addition to everything else needed for these new services.



The configmap was set up in a variety of ways for these apps. A file called a configmap holds all of the environment variables required by the various containers in the application pod. By sending them through a ConfigMap, the environment variables were transferred into the environments of the containers. Other methods used for other projects included mounting an “.env” file inside the main application container, putting a “config.json” file inside the application container, and instantly pushing the variables to the application environment.

To ensure that the microservices were performing at their highest level, extensive testing was conducted. Unit, integration, and system tests had to be run to ensure the microservices were consistent with the entire application stack and performing as expected. Any issues that arose during testing were immediately corrected to ensure that the microservices were performing at their highest level.

Extreme care was taken to pay attention to every little aspect throughout the process in order to make sure that the microservices were incorporated into the larger application stack without any issues. The team had to collaborate to ensure that the microservices were appropriate for the current infrastructure and that any possible hazards were identified and handled.

The freshly introduced applications were rigorously tested using a technique that was put in place. It was first necessary to create the databases that would be used by the respective applications or APIs. Either an initialization container script was employed for this purpose, or the necessary components were directly added to the pertinent app-level static collection data. The creation of each Kubernetes resource associated with the related application was then carefully thought out. It was essential to check the proper port settings inside the Service and Ingress components in order to enable ongoing communication between the program and its external environment.

After the aforementioned setup was complete, the API's functioning was tested in the next step. This was accomplished using a standardized methodology, and the health check endpoint of the API was assessed. Every application went through the identical process, with a curl request with the format:

```
$ curl http://localhost:8080/storinator/healthz
```

was made with the expectation that it would succeed and produce a 200 answer. The same process was carried out again to assure proper operation of the admin panel endpoint.

At this critical juncture, it is crucial to ensure that all necessary microservices and resources have been properly integrated into our deployment architecture. The following work requires automating each stage of deployment across different Kubernetes cluster configurations. This strategic decision is driven by the overarching objective of providing complete automation within the release manager, regardless of whether the deployment is targeted at a local Minikube, Azure AKS, or a GKE cluster. The development of environment-based automation is also necessary because of the significant context-specific variations in hostname/IP addresses, disk mounting paths, and other environment variables.

This approach allows for the patching of primary resource manifests and the subsequent application of diverse overlays specific to each environment.

We have access to a wide range of tools, all of which can help us complete this automation project. Helm is one such tool that enables a template-based approach that allows the definition of particular values that are suitable for each situation. But we chose Kustomize[12] on purpose as our preferred option. Templates are not necessary because of Kustomize, a feature included in the "kubectl" command line tool offered by Kubernetes. It was chosen since it is more scalable and sophisticated by nature than Helm. Thanks to a simple, template-free procedure, Kustomize gives developers the freedom to choose

the level of personalization they want. For the particular circumstances they are working on, they have yaml files.

Both Helm and Kustomize are tools that aid in managing Kubernetes deployments and applications, but they take different approaches and offer different features.

Helm is a package manager for Kubernetes that utilizes charts, which are collections of files describing a set of Kubernetes resources. It adopts a template-based methodology, where charts have pre-defined templates with spaces for movable values. Users can define multiple settings for diverse situations using these templates' parameterization capabilities. The installation, upgrading, and maintenance of apps on Kubernetes clusters via charts is made possible by Helm, which streamlines the deployment procedure. As a result, packaging and distributing programs are made simpler by the higher level of abstraction it offers. However, Helm's template-based method occasionally adds complexity, particularly when dealing with large-scale deployments and handling numerous configurations.

On the other side, Kubernetes' "kubectrl" command line interface has a built-in utility called Kustomize. It uses a method devoid of templates and places an emphasis on customisation and configuration management. With the use of Kustomize, developers may create overlays and patches that alter already-existing Kubernetes resources without the need for additional template files. It makes use of the base and overlay paradigm, where a base configuration is defined as a collection of resources, and overlays are then used to introduce desired alterations. A more precise and scalable method of managing configurations and customizations across several environments is offered by Kustomize. By permitting gradual changes and the composability of settings, it provides flexibility and simplicity. For handling complex deployments with several environments and variants, Kustomize is especially helpful.

while Helm relies on templates and charts for packaging and managing applications, Kustomize focuses on customization and configuration management through overlays and patches. Helm provides a higher level of abstraction and simplifies the deployment process, while Kustomize offers greater flexibility and scalability for managing configurations in diverse environments. The choice between Helm and Kustomize depends on the specific needs and complexities of the deployment scenario.

The current strategy revolves around establishing a "base" directory within the release directory of each microservice, wherein a fundamental template is defined. Consequently, multiple overlays can be applied based on the target cluster, such as "overlays/minikube" and "overlays/azure". Furthermore, future expansions envision the comprehensive definition of setups tailored to distinct development, staging, and production environments, facilitating the application of separate alterations and overlays in accordance with their respective requirements.

In order to achieve Kustomize, we first applied these intricate configuration sets to a select few microservices, principally our core base API and the virtual machine provider. Initially, the pertinent files particular to each application were arranged and divided into distinct overlays of the primary program structure.

Use "/events-subscriber" in the main application directory as an example. Previously, this release directory contained all of the imperative manifests, such as deployment.yaml, configmap.yaml, service.yaml, and others. But by splitting it into "/base" and "/overlays," we were able to define a template procedure of adding Kustomize to any application for our release manager.

All of the previously existing manifests are cleverly stored in the "/base" directory; the only exception is the configmap, which we choose to dynamically create using our Kustomize setup. In addition, we added a crucial file called "kustomization.yaml" to the base directory. This file is essential to Kustomize's ability to identify and coordinate any required setups or

automation tasks. We concentrate the definition of key resources in one consolidated area by listing all base resources in this kustomization.yaml, making maintenance easier and increasing overall effectiveness.

```
apiVersion: kustomize.config.k8s.io/v1beta1

kind: Kustomization

resources:
- deployment.yaml
- ingress.yaml
- service.yaml
```

Fig-8 Base Kustomization template

In this context, a set of useful files is utilized, particularly in relation to overlays, such as "overlays/minikube," to customize the basic resources in line with the unique requirements of each configuration. One file that is essential to this process is the ".env" file. It functions as a channel for describing the environment variables that the application container will use.

```
Application_Name
├── base
│   ├── kustomization.yaml
│   ├── deployment.yaml
│   ├── service.yaml
│   └── ingress.yaml
└── overlay
    └── minikube
        ├── kustomization.yaml [ mandatory ]
        ├── set_memory.yaml [ mandatory ]
        ├── replicas.yaml [ optional ]
        ├── patch.yaml [ optional ]
        ├── config.json [ optional ]
        └── .env [ optional ]
```

Fig-9 Custom template for kustomize

Similar principles are applied to our customized configuration. The creation of a customized configmap unique to that specific overlay is made possible by the inclusion of the proper ".env" file within the overlay directory. The "configMapGenerator" attribute offered by the kustomize tool is used to accomplish this.

The idea for a configmap came from the knowledge that different situations typically require distinct sets of variables. It's also advised against storing all environment variables in the source code repository itself. Variables and secrets are expected to be controlled and safely stored in the future utilizing a safe technology like Vault. This plan ensures the separation of duties and enhances the security of sensitive information. During the deployment process, these variables and secrets will be deleted from Vault to ensure their availability only when absolutely necessary for the proper functioning of the application.

```
Kustomization.yaml file contents :

resources:
- ../base/

configmapGenerator:
  files:
    - .env [ if required ]
    - config.json [ if required ]
    - readme.html [ if required ]

images:
- name: registry./** CONFIDENTIAL **/<app-name>
  newTag: v1.1.0 [ mandatory ]

patchesStrategicMerge:
- replicas.yaml
- set_memory.yaml [ mandatory ]
```

Fig-10 Overlay Kustomization

When making adjustments to the base deployment, such as changing the mounting path or altering environment-specific data, a dedicated patch.yaml file is the go-to resource. However, the crown jewel of these files is the

referred "set-memory.yaml". By allowing us to set the memory and CPU needs as well as the maximum amounts of resources that can be employed, this amazing file performs a crucial job. In the realm of Kubernetes, setting appropriate memory and CPU limits and requests for each pod is not simply a best practice; it is a need. A single pod could ferociously consume all resources in a cluster if the proper limits aren't established, leaving other processes famished and exposed. Thus, the inclusion of this distinct file becomes imperative.

The "set-memory.yaml" file is being separated from the rest of the codebase for two main reasons:

- First and foremost, it is to guarantee that the complete software is still readable by every member of the prestigious Site Reliability Engineering team, allowing for simple comprehension.
- Secondly, it intends to create a standardized procedure whereby the development of a special file for configuring memory requests and limitations becomes a custom, one that is highly regarded due to its critical significance in preserving system performance and stability.

Within the complicated network of our template there is another very significant file with the evocative name—"replicas.yaml". As its name implies, this crucial component's major function is to specify the minimum number of Pods, or replicas, that must be deployed for a given application. In the vast Kubernetes cosmos, there are numerous circumstances where having several copies is essential. These replicas act as vigilant watchmen who are prepared to step in as soon as a Pod falters or is captured by failure. Through the orchestration skills of the Deployment object, Kubernetes effortlessly tries to maintain the necessary number of replicas, ensuring the everlasting dependability of our deployments.

This arduous task is deftly executed behind the scenes, courtesy of the Kubernetes orchestration engine.

The “replicas.yaml” file is separated for two reasons. We can first carefully monitor the target replica count for each individual application. In the holy ground of development environments, one Pod can be enough to support the growth of our application. On the other hand, the need for many copies emerges in the staging and production environments, where the nature and purpose of the microservice define the course of action, boosting our application's resilience and enhancing its capacity to handle a variety of workloads.

The venerable "patch.yaml" file served as the focal point of our initial resource creation efforts, where we methodically and precisely built all of our patches. However, as we moved forward, a paradigm shift steered us in the direction of a more sophisticated plan of action. We recognized the beauty of a dual entity—the revered.env and the venerable config.json. The ethereal spirit of environmental variable modifications was carried through these holy items. The firm "patch.yaml" remained steadfast, providing as a shelter for them while other patches adorned our codebase. We were able to use Kustomize's esoteric powers with unrivaled dexterity over any microservice thanks to this harmonious trio of files, which also provided us with a structured, linear framework.

We perform the sacred incantation "kubectl apply -k," which pushes Kustomize to direct the creation process, to bring our painstakingly created resources to life within an overlay. It is crucial to follow the sacred naming pattern and give our files the honorable suffix "kustomization.yaml" (or.yml) since Kustomize determines its function and aligns its occult energies using this sanctified name.



```
> kubectl apply -k overlays/dev
service/frontend-service created
deployment.apps/frontend-deployment created
horizontalpodautoscaler.autoscaling/frontend-deployment-hpa created
```

Fig-11 kubectl apply -k command

Our arsenal also includes the indispensable command "kustomize build," which creates all the resources and applies all the patches. During the debugging phase, this command is really helpful since it enables us to check the accuracy of the manifests that were generated and quickly address any potential problems.

In order to ensure the execution of the appropriate command for the respective overlay, a mechanism to detect the cluster environment needed to be implemented. To address this requirement, an additional script was introduced alongside the top-level "setup.sh" script. This script was designed to identify the cluster environment and store the corresponding cluster name within the "CLUSTER\_ENV" environment variable.

The purpose of incorporating this environment variable was to enable conditional checks and facilitate different actions within the setup scripts for various resources and applications based on the cluster environment. For instance, certain microservices might be unnecessary for the Azure AKS setup but essential for the minikube setup. By utilizing the "CLUSTER\_ENV" variable, environmental checks could be seamlessly integrated into all relevant bash scripts, ensuring appropriate customization and compatibility with the specific cluster environment.

Our main goal—successfully developing a solid and seamless local orchestration framework for automating deployments on the prestigious

minikube platform—has been triumphantly accomplished after meticulous execution and extensive testing of the entire setup.

We have carefully negotiated the complex maze of complexities that Kubernetes has to offer over this arduous trip, and we have carefully created a thorough deployment workflow. Our unrelenting dedication to quality is evident in the smooth integration of numerous microservices, the thorough setup of resource-specific settings, and the exact orchestration of each component within the dynamic ecosystem of minikube.

Our local deployment automation's resounding success is a monument to the unshakable commitment, unrelenting pursuit of quality, and unrivaled teamwork displayed by our great cohort of software engineers. In future the application could be easily deployed and even make the entire development process faster ahead, thanks to the solid foundation of our local deployment automation design. With the knowledge and experience gained from this amazing initiative, we are prepared to approach further deployments with assurance, adaptability, and the unrelenting dedication to quality that characterizes our team.

We have rigorously optimized each component of our deployment pipelines through constant debugging and careful error-resolution, ensuring flawless performance and dependability. Due to our fervent desire for excellence, we have successfully combined the strengths of Kustomize and Helm to template and tailor our deployments while preserving unmatched scalability and configurability.

A vast expanse of knowledge had to be traversed, emerging as skilled practitioner in the field of deployment automation, wrestling with the complexities of helm charts and Kustomize overlays to mastering the art of establishing environment variables, secrets, and resource limitations.

In conclusion, our effort to automate local deployments on minikube has been a monument to the indomitable human spirit, powered by a constant drive to

overcome obstacles and push the limits of what is thought to be feasible. Not only is the perfect execution of our deployment strategy a key indicator of our success, but also the priceless learnings and profound growth that each member of our great team has experienced. Together, we have improved deployment automation to a new level and permanently altered the ever-changing field of software engineering.

## **Chapter-4**

### **EXPERIMENTS AND RESULT ANALYSIS**

A thorough set of experiments was methodically carried out and then meticulously assessed in order to achieve our goal of containerizing the various components of the "Workspaces" application and enabling their deployment through the Kubernetes platform. These tests had three main goals: to evaluate the effectiveness of our automation efforts, to evaluate the performance of the deployment process, and to analyze the practicality of local deployment using Minikube.

#### **4.1 Containerization and Microservice Conversion:**

In this experiment, the containerization and creation of microservices for each individual component that makes up the program were given special attention. We painstakingly created Docker images for each component, assuring their independence, scalability, and isolation by adhering to industry-leading containerization techniques.

To thoroughly assess the functionality and effectiveness of the painstakingly created initcontainer scripts, numerous exhaustive tests were carried out utilizing a wide variety of Docker commands. These scripts were manually executed on a local machine and within a cluster after being smoothly integrated into the container via the Dockerfile. These testing techniques were used to carefully determine the expected behavior and performance of the scripts in various situations.

In the context of script composition, it is imperative to underline the relevance of precisely defining the appropriate "shebang", be it bash or the related shell script. Notably, several Alpine Linux-based apps don't include the bash shell in their main application containers.

The shebang, also known as the hashbang or the interpreter directive, is a special line placed at the beginning of a script file in Unix-like operating systems. It serves as an instruction to the system on how to execute the script by specifying the path to the interpreter or the shell that should interpret the script.

The shebang line starts with a hash symbol (#) followed by an exclamation mark (!). Immediately after the exclamation mark, the path to the interpreter or shell is specified. For example, a common shebang line for a bash script would be:

```
#!/bin/bash
```

Fig-12 Bash shebang

The script should be run using the Bash interpreter, which may be found at the pathname /bin/bash, according to this.

```
#!/bin/sh
```

Fig-13 Shell shebang

The experimental outcomes conclusively demonstrated the successful containerization of the application, with each microservice effectively fulfilling its intended functionalities. Due to the incorrect shebang declaration, the scripts were unable to achieve executable format, needing careful analysis and correction of this component to ensure flawless execution and optimal performance.

If the script is correctly configured, in a non-kubernetes environment it would give the following output which repeatedly checks for a service and hence doesn't start the main application container.

```
Server:      8.8.8.8
Address:    8.8.8.8:53

** server can't find localmongo: NXDOMAIN

** server can't find localmongo: NXDOMAIN

Waiting for MongoDB
Server:      8.8.8.8
Address:    8.8.8.8:53

** server can't find localmongo: NXDOMAIN

** server can't find localmongo: NXDOMAIN
```

Fig-14 Init Script Testing

It is crucial to precisely define the initcontainer section within the deployment.yaml manifest for the particular container in order to test the initcontainer script within a Kubernetes cluster. The configuration and operation information for the initcontainer are described in this section of the manifest.

The initcontainer operates independently after the deployment is started before the main container starts. The initcontainer's job is to carry out required initialization procedures or set up the environment for the main container. It might involve operations like copying files, creating dependencies, or running particular tasks.

**4.2 Testing Inside a Kubernetes Cluster:**

The initcontainer script finishes after successfully completing its assigned responsibilities during testing if it operates as intended. The resource is then detected as being up and running, indicating that the container's initialization was successful. In contrast, if any problems are encountered when the

initcontainer script is being run, the container initialization process may fail, and suitable troubleshooting and debugging measures must be done to find and fix the underlying issue.

designing and testing the initcontainer in the deployment carefully. The Kubernetes cluster can make sure that the required containerized application is ready and is successfully executed using the yaml manifest.

```
Server:      10.96.0.10
Address:    10.96.0.10:53

Name:      workspace-cluster-kafka-bootstrap.kafka.svc.cluster.local
Address: 10.99.210.54

Broker workspace-cluster-kafka-bootstrap.kafka.svc.cluster.local:9092 is up!
```

Fig-15 Init Script Logs in a Cluster

After every resource and microservice which comprise the application is up and running, the application can be tested in any web browser with the hostname of the particular development server which is being used for running the minikube (or Azure AKS) cluster.

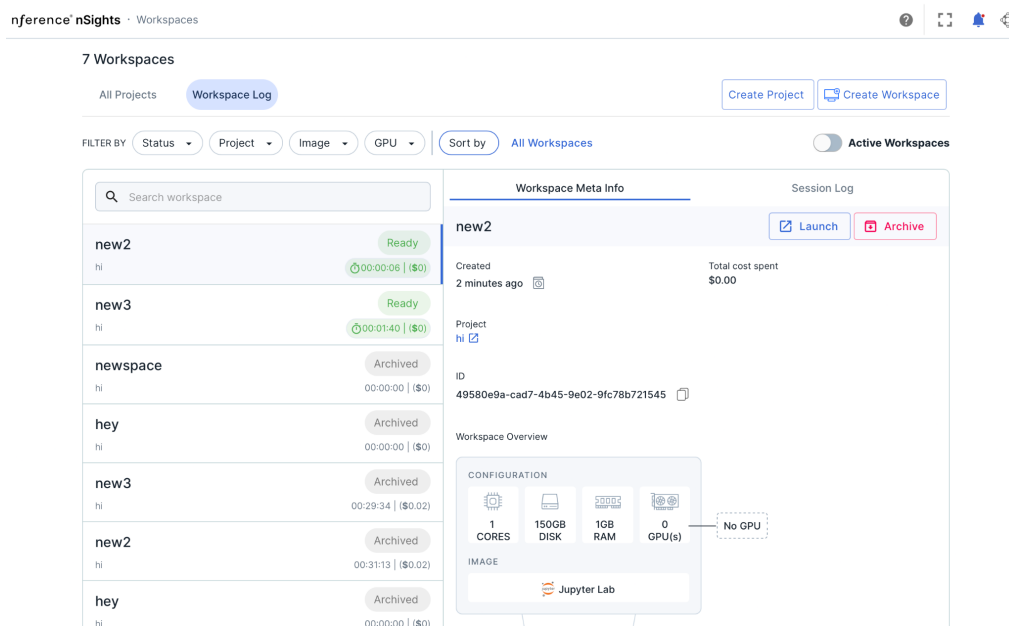


Fig-16 Running Application UI

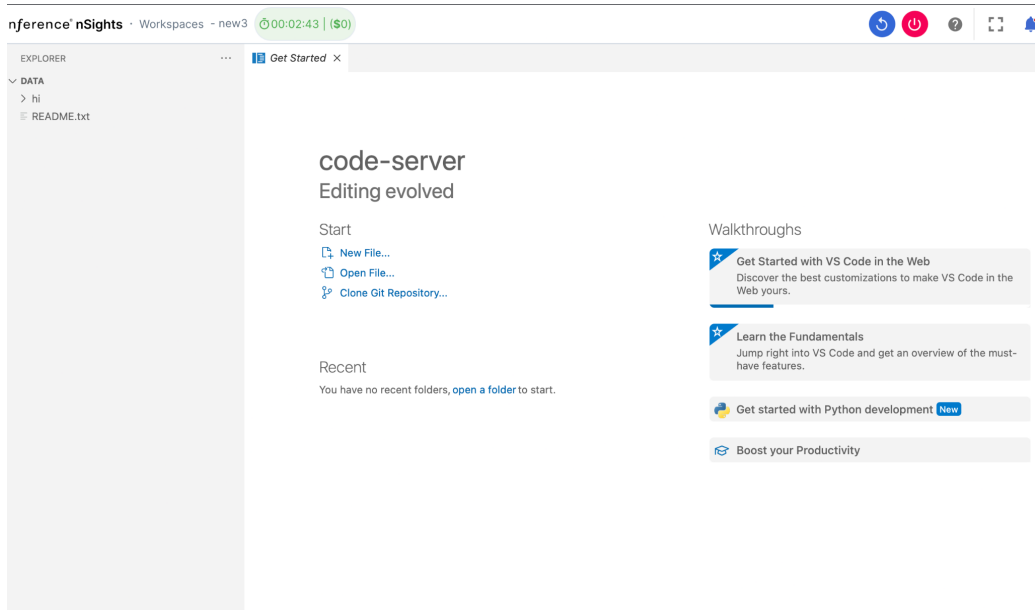


Fig-17 Running Workspace UI



## **Chapter-5**

### **CONCLUSION**

To conclude this report, our deployment procedure culminated with the painstaking deployment of a wide range of microservices, which comprise our application “Workspaces” inside the boundaries of a local minikube Kubernetes cluster, including essential resources like MongoDB, Kafka, and Ingress Nginx Controller. The thorough containerization of these programs, which was carried out using specially created Dockerfiles to ensure their encapsulation and promote portability between environments, marked the beginning of this complex journey.

We developed and deployed deftly written scripts to be executed inside init containers, collaborating with the microservices, to guarantee the successful completion of essential precheck tasks. These meticulously written scripts played a significant part in easing crucial activities, including loading static collections as prerequisites into the database and confirming the availability of resources used by certain applications. By leveraging the capabilities of init containers, the setup process attained enhanced reliability and seamless orchestration.

After all the applications were successfully containerized, our attention turned to carefully creating detailed Kubernetes manifests for each microservice. These manifests, which included crucial resources like Deployments, Ingress, Services, and ConfigMaps, functioned as blueprints that specified the desired state of the deployment. Each microservice's specific manifests are housed in dedicated directories that were created to maintain a well-structured and organized codebase.

A versatile "setup.sh" script was painstakingly created to further ease the deployment procedure. This script served as the main orchestrator, making it simple to execute the complete application stack. Its capabilities went beyond

simple execution because it cleverly divided the deployment operation into various phases. First off, by making sure that Deployments, Ingress, Services, and ConfigMaps were correctly configured and instantiated, it successfully handled the deployment of individual resources. By giving each component fine-grained control, this modular architecture promoted adaptability and reuse.

The "setup.sh" script also made it easier to deploy microservice apps and made sure that each microservice was operational within the Kubernetes cluster. The deployment of the full application stack might be automated and orchestrated by running this script. We created a very effective and well-organized deployment process for our microservices through rigorous attention to detail, the construction of thorough Kubernetes manifests, combined with the design of a modular and flexible deployment script. This method not only made it easier to manage the application stack, but it also laid the groundwork for future iterations' scalability, dependability, and ease of maintenance.

Our journey required the adaption of Kubernetes manifests to cater to several cluster environments, including minikube, Azure AKS, and GKE, in addition to the complexity of the deployment process. We used Kustomize, a potent tool for modifying Kubernetes setups, to do this. Our configuration changed fluidly to conform to the specifics of each Kubernetes environment with the addition of fresh kustomization.yaml files and the application of strategic fixes.

We incorporated environment detection techniques into our workflow to guarantee that the deployment procedure remained adaptable and agile. These techniques allowed the execution of instructions tailored to the current cluster environment on a dynamic basis. By using this automated identification, we reduced the potential of errors while deploying to different Kubernetes settings and removed the need for manual intervention.

To make our setup available to outside users and to enable access through web browsers, we used a unique Haproxy service. This crucial element served as a load balancer and reverse proxy for our system, efficiently routing incoming traffic to the appropriate microservices. In order to exactly configure Haproxy to match our needs, we manually modified the haproxy.cfg file. This necessitated creating routing rules, outlining backend services, and honing load balancing strategies in order to maximize speed and ensure error-free communication between external users and our microservices.

By carefully addressing the configuration requirements of diverse cluster settings using Kustomize, including environment detection techniques, and expertly configuring Haproxy, we were able to create a robust and flexible deployment procedure. With the aid of this thorough strategy, we were able to successfully deploy our application stack across various Kubernetes settings while preserving the best possible performance, scalability, and accessibility for outside users.

As thorough testing and debugging were crucial components of ensuring the dependability and stability of the complete system, the deployment process was not without its difficulties. For the purpose of addressing particular problems and improving overall performance, iterative adjustments and optimizations were regularly implemented.

We carefully scrutinized the interactions, performance indicators, and error handling capabilities of the deployed microservices through diligent testing processes. We were able to find and fix any potential setup bottlenecks, vulnerabilities, or inconsistencies thanks to our thorough testing strategy. To identify and fix any problems, lengthy debugging sessions were carried out concurrently. We continuously monitored the system's behavior and used logs, error messages, and monitoring tools to identify the underlying reasons of any irregularities. We were able to implement the necessary patches through meticulous analysis and troubleshooting, ensuring the deployed microservices ran without a hitch.

We took comments from a range of stakeholders, including developers, testers, and end users, into account during this iterative process. Their insightful opinions and observations were extremely helpful in determining the final deployment configuration and in improving the system's usability, performance, and scalability.

Through extensive testing, debugging, and incremental improvement, we were able to create a deployment arrangement that was reliable and stable. Developers now have quick and scalable settings for development, testing, and experimentation thanks to the solution's smooth operation on any development server with a minikube cluster.

The configuration and provisioning of the application stack are greatly accelerated by the automated deployment process, which also reduces manual labor and permits quick iteration cycles. Developers can now quickly deploy the complete application stack locally, freeing them up to concentrate on writing, testing, and debugging application code, which is their primary responsibility. This improved workflow encourages a collaborative development environment, increases productivity, and shortens time to market.

To sum it up, the deployment setup has been strengthened by the thorough testing, debugging, and iterative refinement processes, ensuring its stability, reliability, and adaptability. The solution is now prepared to assist the development team in its pursuit of effective and scalable software development, enabling them to quickly and confidently produce high-quality products.

## REFERENCES

- [1] “Kubernetes Documentation,” *Kubernetes.io*, 2019.  
<https://kubernetes.io/docs/home/>
- [2] “Minikube Documentation” *Kubernetes*.  
<https://kubernetes.io/docs/tutorials/hello-minikube/> (accessed May 13, 2023).
- [3] “Docker documentation,” *Docker Documentation*.  
<https://docs.docker.com/reference>
- [4] A. Tesliuk, S. G. Bobkov, V. A. Ilyin, Mikhail Krasavin, Alexey Poyda, and Vasily Velikhov, “Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis,” 2019 Ivannikov Ispras Open Conference (ISPRAS), Dec. 2019
- [5] “Borg, Omega, and Kubernetes - ACM Queue,” *Acm.org*, 2019.  
<https://queue.acm.org/detail.cfm?id=2898444>
- [6] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou, and Z. Li, “Microservices: architecture, container, and challenges,” 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), Dec. 2020, doi: <https://doi.org/10.1109/qrs-c51114.2020.00107>.
- [7] “Consul Documentation”  
<https://developer.hashicorp.com/consul/docs>
- [8] “MongoDB Documentation”  
<https://www.mongodb.com/docs/>
- [9] “Init Containers” *Kubernetes*.  
<https://kubernetes.io/docs/concepts/workloads/pods/init-containers/>

[10] “Jenkins Pipeline” *Pipeline*. <https://www.jenkins.io/doc/book/pipeline/>

[11] “Helm Documentation” *helm.sh*. <https://helm.sh/docs/>

[12] “Kustomize Documentation” *kustomize.io*  
<https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/>

ORIGINALITY REPORT

5%

SIMILARITY INDEX

5%

INTERNET SOURCES

2%

PUBLICATIONS

%

STUDENT PAPERS

PRIMARY SOURCES

1

[www.ir.juit.ac.in:8080](http://www.ir.juit.ac.in:8080)

Internet Source

1%

2

[ir.juit.ac.in:8080](http://ir.juit.ac.in:8080)

Internet Source

1%

3

[www.ijraset.com](http://www.ijraset.com)

Internet Source

<1%

4

[erepository.uonbi.ac.ke](http://erepository.uonbi.ac.ke)

Internet Source

<1%

5

[www.coursehero.com](http://www.coursehero.com)

Internet Source

<1%

6

[www.aquasec.com](http://www.aquasec.com)

Internet Source

<1%

7

"Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles", Springer Science and Business Media LLC, 2020

Publication

<1%

8

[thesai.org](http://thesai.org)

Internet Source

<1%