

Upgrade in Kubernetes Clusters - State of Practice and Analysis from Availability Perspective

Shresthi Garg

A Thesis

In the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

October 2022

© Shresthi Garg, 2022

CONCORDIA UNIVERSITY

School Of Graduate Studies

This is to certify that the thesis prepared

By: **Shresthi Garg**

Entitled: **Upgrade in Kubernetes Clusters - State of Practice and Analysis from Availability Perspective**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____	Chair
Dr. J. Paquet	
_____	Internal Examiner
Dr. O. Ormandjieva	
_____	Internal Examiner
Dr. J. Paquet	
_____	Thesis Supervisor
Dr. F. Khendek	
_____	Thesis Supervisor
Dr. M. Toeroe	

Approved by: _____
Dr. L. Kosseim, Graduate Program Director
Department of Computer Science and Software Engineering

_____ 2022

_____ Dr. M. Debbabi, Dean,
Faculty of Engineering and Computer Science

ABSTRACT

Upgrade in Kubernetes Clusters – State of Practice and Analysis from Availability Perspective

Shresthi Garg

Many systems must run without interruptions, yet they must upgrade to address issues, such as fixing bugs or adding new functionalities. Delivering uninterrupted services is not just the responsibility of the system providing the services but also the environment hosting the system. So, it is essential to understand and analyze the service availability guaranteed during an upgrade by the orchestration platform that hosts the system.

Kubernetes is a popular orchestrator of containerized workloads and services. It is essential to understand and analyze the impact of upgrades in a Kubernetes cluster: the effect of an upgrade on the service availability, how a failure during an upgrade is taken care of by Kubernetes and by the tools managing the Kubernetes cluster; and the effects of an upgrade process failure.

This thesis investigates, quantifies these impacts, and analyzes the causes. This thesis identifies three upgrade levels in a Kubernetes cluster: Kubernetes cluster version upgrade, Kubernetes application upgrade, and container runtime upgrade. We evaluate and analyze the state of the practice of upgrades for each level by performing various experiments under different (failure) scenarios. For each experiment, the manual collection of event timestamps and then the calculation of evaluation metrics (using collected timestamps) is a tedious and time-consuming task. To tackle this issue, we devise and implement an “Auto-Metric collector” tool

that automates this process of event collection and metric calculation. The results of our experiments and analysis highlight the shortcoming of Kubernetes in identifying upgrade process failure and taking remediation measures. We propose potential solutions for some of the identified shortcomings.

Acknowledgments

This work is an achievement; while it bears my name, it would not have been possible without the help of these wonderful people: whom I would now like to thank. First and foremost, I thank God for guiding me and giving me strength through it all. As a close second, I would like to thank my family: My father, who would always inspire me to see the big picture; My mother, who would be my best friend and cheer me up on days that wouldn't seem easy; My brother, who would always be there for the advice whenever I needed one and make me understand how hard work would always pay off, my best friend and partner, Mayank who has been so patient, supportive and my biggest cheerleader throughout this journey.

If someone had asked me in the year 2019 about the mysteries of research, I would have shrugged it off; now that I could only unravel it, it is because of my supervisors, Dr. Khendek and Dr. Toeroe. I thank them for their guidance, for shaping my ideas, and for their belief in me.

I would also like to thank my colleagues of MAGIC research lab, especially Siamak, for his advice, support, and friendship.

This work has been conducted within the NSERC/Ericsson Industrial Research Chair in Model-Based Software Management, which is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson and Concordia University.

Table of Contents

List of Figures	xi
List of Tables	xiii
List of Acronyms	xv
1 Introduction.....	1
1.1 Thesis Motivations.....	1
1.2 Thesis Contributions	2
1.3 Thesis Organization	3
2 Background.....	5
2.1 Service Availability	5
2.1.1 Service Availability during Upgrades.....	6
2.2 Containers	6
2.3 Introduction to Kubernetes	7
2.4 Kubernetes Cluster Architecture.....	7
2.5 Failure Handling with Kubernetes.....	12
2.5.1 Pod Failure.....	12
2.5.2 Node Failure	13
2.6 Upgrades in Kubernetes Clusters.....	14
2.6.1 Kubernetes' Cluster Version Upgrade.....	15
2.6.2 Kubernetes' Application Upgrade	15
2.6.3 Container Runtime Upgrade.....	15
3 Evaluation Method and Setup.....	16
3.1 Evaluation Method.....	16
3.2 Evaluation Cluster Setup.....	17

3.2.1	Cluster Architecture.....	17
3.2.2	Application Deployment.....	18
3.3	Evaluation Scenarios.....	19
3.3.1	Upgrade Scenario	20
3.3.2	Failure Scenarios	20
3.4	Metrics for Evaluation	21
4	Auto-Metric Collector – a Tool to Automate the Process of Metric Collection.....	23
4.1	Problem Statement.....	23
4.2	Evaluation of Existing Tools	24
4.3	Collecting Metrics Via Auto-Metric Collector.....	25
4.3.1	Architecture	25
4.3.2	Operations.....	27
4.4	Conclusion	28
5	Kubernetes Cluster Version Upgrade	30
5.1	Current Practice of Upgrade	30
5.1.1	Kubeadm.....	31
5.1.2	kOps.....	34
5.2	Evaluation of Kubernetes’ Cluster Version Upgrade	37
5.2.1	RQ1: Evaluate the Impact on Application Services during Kubernetes’ Cluster Version Upgrade	38
5.2.2	RQ2-1: Evaluate the Impact of Kubernetes’ Cluster Version Upgrade in the Presence of Pod Failure, on its Failure Recovery Actions and on Application Services....	41

5.2.3	RQ2-2: Evaluate the Impact of Kubernetes' Cluster Version Upgrade, on its Worker Node Failure Recovery Actions and on Application Services.....	48
5.2.4	RQ3: Evaluate the Impact of a Failure of the Kubernetes' Cluster Version Upgrade Process.....	56
5.2.5	Assessing the Achievable Service Availability, during Kubernetes' Cluster Version Upgrade, and in Presence of Failure during Upgrade (H ₁)	58
5.3	Overall Analysis and Potential Improvements	62
5.4	Conclusion	65
6	Kubernetes Application Upgrade.....	68
6.1	Current Practice of Upgrade	68
6.1.1	Upgrade Strategies for Stateless Application	68
6.1.2	Upgrade Strategies for Stateful Application.....	70
6.2	Evaluation of Application Upgrade Strategies	72
6.2.1	RQ1: Evaluate the Impact of Kubernetes' Application Upgrade on the Application Services	73
6.2.2	RQ2-1: Evaluate the Impact of Kubernetes' Application Upgrade on the Recovery from a Pod Failure and on Application Services	78
6.2.3	RQ2-2: Evaluate the Impact on the Pod (Application) Version post Recovery from Failure	83
6.2.4	RQ3-1: Evaluate the Impact of the Kubernetes' Application Upgrade Process Failure on the Availability of the Application	84
6.2.5	RQ3-2: Evaluate the Remediation Measures Taken by the Respective Controller (Deployment and StatefulSet Controller) when Kubernetes' Application Upgrade Process Fails	87

6.2.6	Assessing the Achievable Service Availability during Kubernetes' Application Upgrade, and in Presence of Failure during Upgrade (H ₂)	89
6.3	Overall Analysis and Potential Improvements	92
6.3.1	Stateless Application	92
6.3.2	Stateful Application.....	94
6.4	Conclusion	95
7	Container Runtime Upgrade	97
7.1	Current Practice of Upgrade	97
7.1.1	Docker	99
7.1.2	CRI-O	100
7.2	Evaluation of Container Runtime Upgrade.....	102
7.2.1	RQ1: Evaluating the Impact of Container Runtime Upgrade on Service Availability of Hosted Application Instances	104
7.2.2	RQ2: What is the Impact of Container Runtime Upgrade on Recovery from Application Container Failure and on Application Services?.....	106
7.2.3	RQ3-1: Evaluating the Impact of Container Runtime Upgrade Process Failure on Application Services	110
7.2.4	RQ3-2: Evaluating the Recovery Actions Taken by Container Runtime Tools when their Upgrade Process Fails	114
7.2.5	Assessing the Achievable Service Availability, during Container Runtime Upgrade, and in Presence of Failure during Upgrade (H ₃)	115
7.3	Overall Analysis and Potential Improvements	117
7.4	Conclusion	120

8	Conclusion	122
	Bibliography	126

List of Figures

Figure 2-1: An example of a Kubernetes cluster	8
Figure 2-2: Kubernetes storage creation and usage	12
Figure 2-3: Context of upgrade in Kubernetes	14
Figure 3-1: Evaluation Cluster Architecture.....	18
Figure 3-2: An example of Stateful Application deployed with the name “vod” managed by State Controller	19
Figure 3-3: Metrics for stateless NGINX application.....	22
Figure 3-4: Metrics for stateful video streaming application.....	22
Figure 4-1: High-level architecture of Auto-Metric Collector	25
Figure 4-2: Integration of Auto-metric collector with Kubernetes and FEK stack	27
Figure 5-1: Master node upgrade process using kubeadm.....	32
Figure 5-2: Worker node upgrade process using kubeadm	33
Figure 5-3: An example of kOps managed single master Kubernetes cluster on AWS	34
Figure 5-4: Upgrade process flow in a kOps-created Kubernetes cluster	36
Figure 5-5: Single-master Kubernetes cluster: Service availability achieved when Kubernetes’ cluster version upgrades for a year, managed by kOps, and kubeadm tool	60
Figure 5-6: HA-master Kubernetes cluster: Service availability achieved when Kubernetes’ cluster version upgrade for a year, managed by kOps, and kubeadm tool	61
Figure 5-7: Post the master node upgrade process failure, the figure showing the status of the current config folder	64
Figure 6-1: Illustration of <i>Recreate</i> strategy for stateless application.....	69
Figure 6-2: Illustration of <i>RollingUpdate</i> strategy for stateless application.....	70
Figure 6-3: Illustration of <i>OnDelete</i> upgrade strategy for stateful application.....	71
Figure 6-4: Illustration of <i>RollingUpdate</i> upgrade strategy for stateful application	71

Figure 6-5: Snippet of Progressing status of "myapp-deployment" Deployment when progressing status fails.....	88
Figure 6-6: Service availability achieved when application deployed in a Kubernetes cluster upgrades for a year (with and without failure).....	91
Figure 6-7 : Snippet showing the value of the Conditions field of the Deployment	93
Figure 6-8: Snippet showing progressing status of StatefulSet “vod” stuck during the upgrade	94
Figure 7-1: Illustration of the placement of CRI in a Kubernetes cluster.....	98
Figure 7-2: Interaction of kubelet with the containers running on that node via CRI.....	98
Figure 7-3: Illustration showing Docker as a container runtime in Kubernetes	99
Figure 7-4: Docker upgrade process flow for nodes in a Kubernetes cluster	100
Figure 7-5: Illustration showing CRI-O as a container runtime in Kubernetes.....	101
Figure 7-6: Illustration showing the common component of CRI-O.....	101
Figure 7-7: CRI-O upgrade process flow for nodes in a Kubernetes cluster.....	102
Figure 7-8: Service availability achieved when container runtime integrated with Kubernetes cluster upgrades for a year (with and without failure).....	117

List of Tables

Table 4-1: Evaluation of popular open-source monitoring tools	24
Table 5-1: kOps: service outage and service degradation during worker node upgrade	40
Table 5-2: kubeadm: stateless application pod failure during master node upgrade	42
Table 5-3: kubeadm: stateful application pod failure during master node upgrade	43
Table 5-4: kOps: stateless application pod failure during master node upgrade	44
Table 5-5: kOps: stateful application pod failure during master node upgrade.....	44
Table 5-6: kubeadm: stateless application pod failure during worker node upgrade	45
Table 5-7: kubeadm: stateful application pod failure during worker node upgrade.....	46
Table 5-8: kOps: stateless application pod failure during worker node upgrade	48
Table 5-9: kOps: stateful application pod failure during worker node upgrade	48
Table 5-10: kubeadm: stateless application – worker node failure during master node upgrade	50
Table 5-11: kubeadm: stateful application – worker node failure during master node upgrade	50
Table 5-12: kOps: stateless application – worker node failure during master node upgrade ..	51
Table 5-13: kOps- stateful application – worker node failure during master node upgrade ...	52
Table 5-14: kubeadm: Stateless application – worker node failure during worker node upgrade.....	53
Table 5-15: kubeadm: Stateful application – worker node failure during worker node upgrade	54
Table 5-16: kOps: stateless application – worker node failure during worker node upgrade .	54
Table 5-17: kOps: stateful application – worker node failure during worker node upgrade ...	56
Table 6-1: Stateless application: Impact on the application services during its upgrade	75
Table 6-2: Stateful application: Impact on the application services during its upgrade	78

Table 6-3: Stateless application pod failure during application upgrade.....	80
Table 6-4: Stateful application pod failure during application upgrade	82
Table 6-5: Stateless Application: Impact on application availability due to upgrade process failure	85
Table 6-6: Stateful Application: Impact on application availability due to upgrade process failure	87
Table 7-1: Impact on application services during Docker upgrade	105
Table 7-2: Stateless application container failure during Docker upgrade.....	108
Table 7-3: Stateful application container failure during Docker upgrade	108
Table 7-4: Stateless application container failure during CRI-O upgrade.....	110
Table 7-5: Stateful application container failure during CRI-O upgrade	110
Table 7-6: Impact on the application services due to Docker upgrade process failure (internal failure).....	113
Table 7-7: Impact on the application services due to CRI-O upgrade process failure (internal failure).....	114

List of Acronyms

API	Application Programming Interface
AWS	Amazon Web Service
CNCF	Cloud Native Computing Foundation
CNI	Container Networking Interface
CRI	Container Runtime Interface
FEK	Filebeat Elasticsearch Kibana
GKE	Google Kubernetes Engine
HA	High Available
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
kOps	Kubernetes Operation
NGINX	Engine-X
NTP	Network Time Protocol
OCI	Open Container Initiative
OS	Operating System
PV	Persistent Volume
PVC	Persistent Volume Claim
REST	Representational State Transfer
RQ	Research Question
SC	State Controller
VLC	VideoLAN Client
VM	Virtual Machine

Chapter 1

Introduction

This chapter introduces the motivations and context for this thesis followed by its contributions and organization.

1.1 Thesis Motivations

Many systems must provide continuous and uninterrupted services, like continuous operation systems, for example, air traffic control, banking systems, and telecommunications systems. Service availability is a non-functional characteristic defined as the portion of time the service is accessible in a period [1]. High availability is achieved when the system is available at least 99.999% of the time. Therefore, the total downtime allowed in one year for highly available systems is around 5 minutes [2]. Service availability is not only dependent on the systems providing them; but also, on the environment that hosts and facilitates the orchestration of such systems, like Kubernetes [3].

Kubernetes is an open-source platform for managing containerized workloads and services. In Kubernetes, an application is encapsulated as a container image by a container runtime tool such as Docker. These containerized applications are deployed in Kubernetes as pods. Kubernetes' self-healing property for the deployed application ensures that the clusters always function at the desired state. These healing capabilities of Kubernetes include restarting failed containers to maintain the number of desired pods for the deployed application; rescheduling the containers if the hosts fail; making sure the containers are marked as available only when

it is healthy; killing the containers that are not healthy. Thus, the self-healing mechanism for the deployed application is one of the most crucial functionalities to ensure their service availability.

Upgrades are an essential part of the software lifecycle and maintaining service availability during upgrades can be tricky in a High Available (HA) distributed environment. For example, during upgrades in a HA distributed system, it is problematic to provide application services while maintaining a seamless user experience. As Kubernetes is a popular orchestrator of containerized applications: The question is, to what extent is service availability supported/guaranteed during an upgrade in the Kubernetes cluster?

The thesis aims to answer this question. The goal is to evaluate the state of the practice of upgrades in a Kubernetes cluster, quantify their impact, identify the shortcomings, and propose improvements.

1.2 Thesis Contributions

In this thesis, we investigate three upgrade levels in the context of the Kubernetes cluster: the upgrade of the Kubernetes cluster version, the upgrade of the application hosted in the Kubernetes cluster, and the upgrade of the container runtime tool. We qualitatively evaluate them from service availability perspective, conduct various experiments to quantify the impact, analyze and propose improvements. The main contributions of this thesis are summarized as follows:

- Qualitatively evaluate the current practices of an upgrade for the identified levels in the Kubernetes cluster.
- Perform different experiments to quantify the impact of the upgrade in the following way:

- Evaluate the impact of the upgrade on the service availability of the deployed application instance.
- Evaluate the impact of the upgrade in the presence of failure.
- Evaluate the impact of upgrade process failure.
- Assess achievable service availability for the deployed application, during upgrade and in the presence of failure during upgrade.
- For quantitative evaluation, we devise a tool to automate the process of metric calculation. In this contribution, we:
 - Address the complication of manual metrics collection for experiments of upgrades and evaluate the existing tool.
 - Implement an “Auto-Metric Collector” tool that runs outside the Kubernetes cluster and observes any state change of the user-selected Kubernetes object.
 - In the event of a change in a Kubernetes object, it would collect the defined events and calculate the evaluation metrics.
- Finally, we analyze the results of the performed experiments, identify shortcomings, and propose potential improvements.

1.3 Thesis Organization

This thesis is arranged into eight chapters. In Chapter 2, we provide background information about Kubernetes: its objects, and an overview of its cluster architecture by discussing components involved in hosting and managing the deployed application instances. This chapter also discusses the context of upgrades in a Kubernetes cluster. In Chapter 3, we introduce our evaluation approach, discuss the considered cluster setup and the evaluation metrics in the performed experiments. In Chapter 4, we present the “Auto-Metric Collector” tool, devised, and implemented to automate the calculation of evaluation metrics. In this chapter, we discuss the

architecture of the tool, its application, assumptions, and limitations. In chapters 5, 6 and 7, we discuss the evaluation of the different levels of upgrades in the Kubernetes cluster, identify shortcomings, perform analysis and propose potential solutions. In Chapter 8, we conclude this thesis by summarizing the contributions and discussing possible future work.

Chapter 2

Background

In this chapter, we discuss the concept of service availability in Section 2.1; then we provide an overview of containerization and containers in Section 2.2. In Section 2.3, we introduce Kubernetes, followed by the summary and discussion of its architecture in Section 2.4, and its failure handling in Section 2.5. Finally, in Section 2.6, we introduce and briefly discuss the different levels of upgrades investigated in a Kubernetes cluster.

2.1 Service Availability

In [1], availability is defined as “the degree to which a system is functioning and is accessible to deliver its services during a given time interval.” In [4], the availability of a system can be viewed through the availability of its services. So, service availability can be defined as:

Equation 1

$$\text{Service Availability} = \frac{\textit{ServiceUptime}}{(\textit{ServiceUptime} + \textit{ServiceOutage})}$$

where service uptime is the duration during which the system delivers the given service, while service outage (also referred to as downtime) is the period during which the service is not delivered [1].

High availability is achieved when the system is available at least 99.999% of the time. Therefore, the total downtime allowed in one year for highly available systems is around 5 minutes as described in [2].

2.1.1 Service Availability during Upgrades

System must be upgraded based on their upgrade release cycle. Since services of the system may be impacted during upgrade - especially in case of failure during upgrade, the total service outage caused during system upgrade can be calculated as follows:

$$\text{ServiceOutage} = N_U \times O_U$$

where N_U is the number of system upgrades in a year, and O_U is the service outage for a complete system upgrade. Next, we can use service outage to calculate the duration during which the system delivers the given service i.e., service uptime, as follows:

$$\text{ServiceUptime} = \text{TotalTime} - \text{ServiceOutage}$$

where TotalTime is the total agreed service time in a year. Finally, we substitute the values of service outage and service uptime in Equation 1 to calculate service availability during upgrade. The equation becomes as follows:

Equation 2

$$\text{Service Availability} = \frac{\text{TotalTime} - (N_U \times O_U)}{\text{TotalTime}}$$

The service availability in the Equation 2 represents the availability of the services offered by the system when it is upgraded in a year.

2.2 Containers

Containerization technology encapsulates the application's code and dependencies and enables fine-grained resource control and isolation [5]. A container image is a lightweight executable software package that includes application code, associated dependencies, and libraries. A container runtime engine, like Docker [6], is used to create a container image. Container images become containers when they run on the container runtime engine.

Multiple containers can run on a single host - a virtual machine that runs an Operating System (OS); however, containers remain isolated; the applications running inside these containers do not get impacted by other containers running on that host.

2.3 Introduction to Kubernetes

With the dawn of the Container Era, in a containerised environment, it is essential to have a system that manages the containerized workload. Since, if in a production system a container goes down, another one should be re-started. Automated orchestration of such a use case is crucial to maintain minimum downtime and hence the availability of services. Kubernetes is an orchestrator platform that manages containerized workloads and services. It facilitates both declarative functionalities and workload automation. It was introduced by Google in June 2014 at a Google Developer Forum [7]. Cloud Native Computing Foundation [8] currently maintains its development and support.

Kubernetes orchestrates container-centric infrastructure and provides self-healing mechanisms for the deployed application, service discovery, and load balancing. Thus, it acts as a cluster manager and handles the deployment, management, upgrading, monitoring, and scaling of containerized applications in a distributed environment.

2.4 Kubernetes Cluster Architecture

A group of nodes forms a Kubernetes cluster. Such nodes are either physical or virtual machines on which we install Kubernetes. While the node(s) which has the Kubernetes components installed for managing the orchestration of the cluster is referred as the master node (or control-plane node), and the node(s) that host the deployed application instances is called as a worker node. Since Kubernetes follows a master-slave architecture, its cluster has at least one master node and at least one worker node. Figure 2-1 shows a high-level architecture of a single master and two worker nodes Kubernetes cluster.

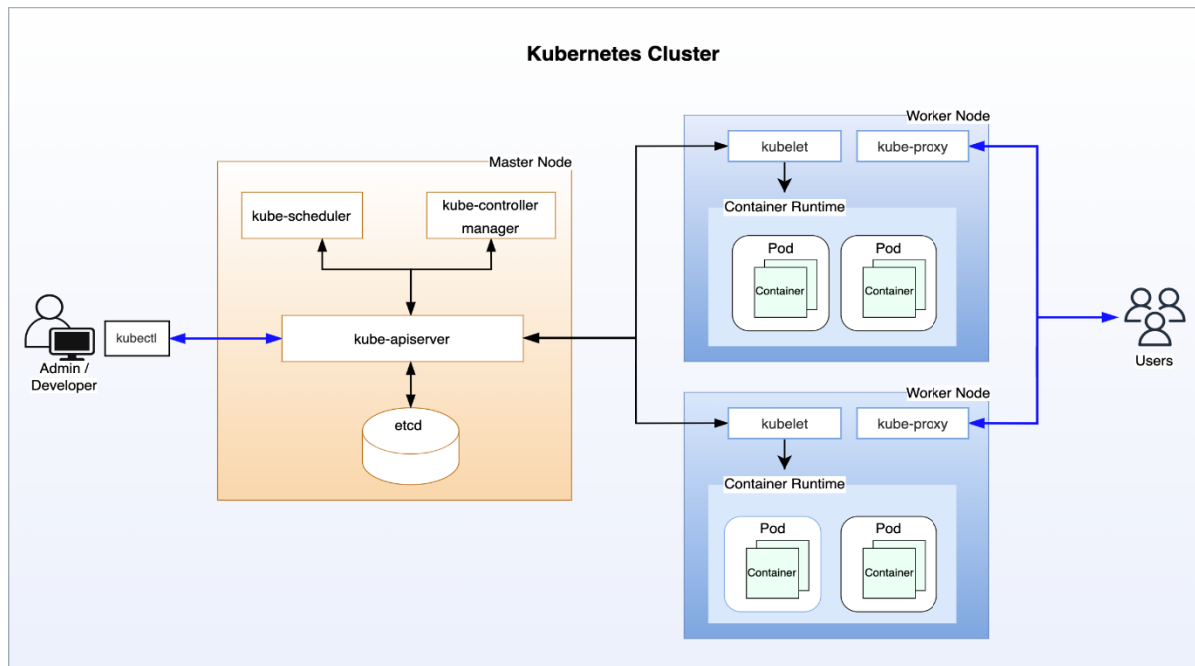


Figure 2-1: An example of a Kubernetes cluster

A user (usually a system administrator or a developer) can communicate with the Kubernetes cluster using a command-line tool called kubectl. The master node component receives the commands sent via kubectl.

In a Kubernetes cluster, the master node includes a collection of components that runs as static pods and are responsible for the orchestration of the cluster. These components are the kube-apiserver, kube-controller-manager, and kube-scheduler. Master node components take global decisions about the cluster through these processes while storing and fetching the cluster's state from another component called ETCD- a highly distributed key-value datastore. The functions of these master node components are discussed in detail hereafter:

- **kube-apiserver:** The kube-apiserver component forms the heart of the orchestration operations. It acts as the front end for all Kubernetes components, management devices, and command-line interfaces. It acts as an interface to create, update, or

configure Kubernetes clusters. The kube-apiserver exposes the HTTP API that allows any component interacting with it to query and manipulate the state of Kubernetes objects.

- **kube-controller-manager:** In Kubernetes, controllers are the brain behind the orchestration operations. Controllers are the control loops that watch the state of the cluster and try to move the current state closer to the desired state. Logically there are different controllers in a Kubernetes cluster, but for simplicity, they are combined into a single binary (running as a single process). For example, the Node controller is responsible for managing the node status; the Endpoints controller takes care of the endpoint objects. While the Deployment Controller manages the orchestration of a Stateless application, the StatefulSet Controller manages the Stateful application and its associated storage.
- **kube-scheduler:** The kube-scheduler component of a master node is responsible for scheduling the deployment of the pod(s) on an available worker node. As the kube-scheduler distributes the workload, it ensures that the pre-defined constraints mentioned for the application in its pod specification are met.
- **ETCD:** It is a key-value database that stores the data associated with every Kubernetes object (such as a pod or node). It contains information such as the current state of the cluster, its configuration, and the state of Kubernetes objects. It implements locks within the cluster to avoid conflicts to maintain atomicity during concurrent operations. It uses Raft [9] consensus to fulfill its duties.

The worker node in a Kubernetes cluster hosts containerized application instances. To deploy an application on Kubernetes, a deployment configuration YAML file is created, which

provides information regarding the management of the application instances on a cluster. For example, the information related to container configuration, storage, and other Kubernetes resources required to run the application is specified. Once the deployment configuration YAML is provided to Kubernetes, it creates pods to host the application instance defined in the specification file.

Pods represent the smallest deployable units in a Kubernetes cluster. A pod can include one or more application containers; that share resources such as storage, networking space, etc. The containers in a pod share an IP address and a namespace, are always co-located and co-scheduled, and run in a shared context on the same worker node. To spin and manage these containers, a container runtime tool, like Docker, CRI-O [10] etc., must be installed on each node of the Kubernetes cluster.

Each node in the cluster has a monitoring agent called kubelet. It registers itself to the node on which it runs and gets the information about the containers running on it (from the master node components). This information is called PodSpec- a YAML or JSON file that describes the pod(s). Kubelet ensures that the container(s) defined in the PodSpec are running and healthy. The Kubelet component of the worker node is also responsible for sending status reports to the master node at a regular period of 10 seconds (by default). This periodic status report is also referred to as heartbeat, where kubelet informs the master node about the status of the worker node on which it is running and the containers running on that worker node.

Since pods can be created and destroyed dynamically due to various reasons, such as during scaling operation of the deployment replicas, due to a rolling update, or due to failure, etc., pod's IP addresses might change over time, thus causing communication disruption in

cases where the IP address was the point of communication. A Kubernetes Service is an abstraction that defines a set of pods as an endpoint and a policy to access them. Kubernetes can update the endpoint when a set of pods in service changes.

The networking on each node is managed by a component called Kube-proxy; it also applies networking rules and allows running containers to be accessible to the external world. When a Service is guaranteed through ClusterIP or NodePort, kube-proxy forwards it to the appropriate pod. The kube-proxy can run in three proxy modes - *userspace*, *iptables*, and *ipvs*; the networking differs depending on the selected proxy mode. The default proxy mode is *iptables*.

Storage is another significant aspect that should be considered while managing containerized applications. Two storage abstractions are present with Kubernetes: Volumes and Persistent Volumes (PV). While the former is used for storing temporary data in a pod, the latter is for storing data regardless of the pod's lifecycle. For pods to start using these volumes, they need to be claimed (via a persistent volume claim), and the claim is referenced in the specification for a pod. A Persistent Volume Claim (PVC) describes the amount and characteristics of the storage required by the pod, finds any matching persistent volumes, and claims those. Storage Classes depict default volume information. Figure 2-2 shows the creation of Persistent Volume and its claim.

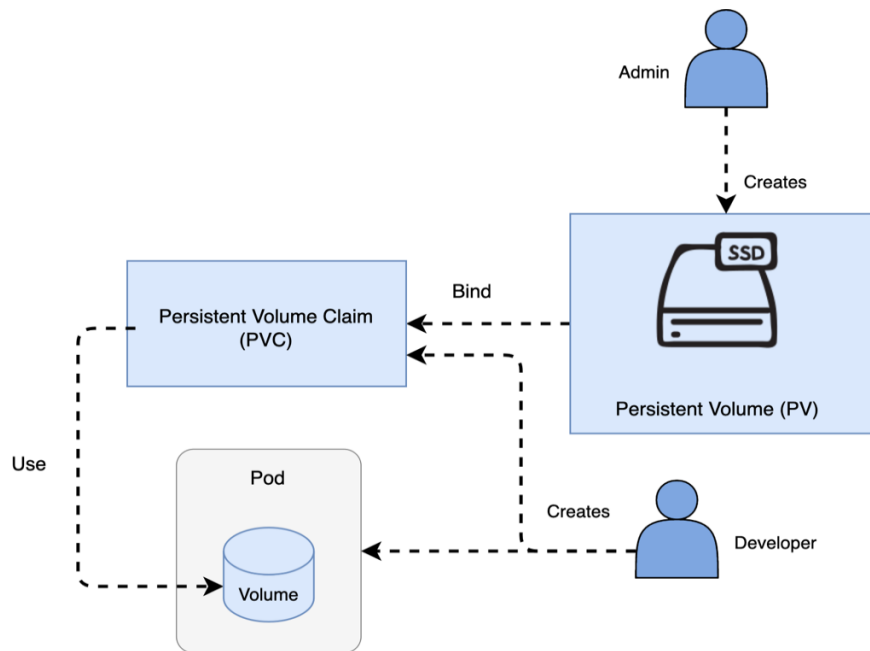


Figure 2-2: Kubernetes storage creation and usage

2.5 Failure Handling with Kubernetes

Kubernetes provides the deployed applications with a self-healing capability as the most significant feature for maintaining availability. It does that by continuously monitoring and repairing its object. In this sub-section, we present an overview of the failure-handling measures carried out by Kubernetes.

2.5.1 Pod Failure

When any pod is deployed, the container runtime tool spins the application container(s) specified in the pod template. Also, an additional parent container may be created depending on the container runtime used. For example, if the container runtime is Docker, a parent container called a pod container is also created for each pod.

Kubernetes monitors the containers on a worker node by communicating with the kubelet component on that worker node. This kubelet component continuously communicates with the container runtime tool to fetch the condition of the containers running on the node. If

the application container crashes, kubelet reacts according to the defined *restartPolicy*, whose default value is *Always*, meaning that any failed container is always restarted.

As stated earlier, if the container runtime is Docker, the pod container is the parent container, if the pod container crashes, the associated application container also fails, and the pod loses its *Ready* status. When Kubernetes is informed of this failure by kubelet, it removes the pod from the endpoint list to avoid any service requests to the failed pod. Next, it creates a new pod, and when the new pod's status becomes *Ready*, it is assigned an IP address by the CNI (Container Network Interface) [11]. These changes are reflected in the iptables by the kube-proxy component of the master node. The pod starts getting service requests when its IP address is in the iptables, and thus failed pod is recovered.

2.5.2 Node Failure

In a Kubernetes cluster, the kubelet component sends the status of its worker node to the master node at a regular interval (by default every 10 seconds), also referred to as heartbeats. If Kubernetes loses connection with the worker node for some reason, such as the node reboot, the master node stops receiving heartbeats from this worker node. On the fourth missed heartbeat, the master node marks the worker node's state as *NodeNotReady*. The pods on that node are then marked for termination, and the pod's status shows *Completed*. After waiting for the duration specified in the *pod-eviction-timeout* [12] (by default, 5 minutes), Kubernetes will initiate the eviction of the pods on the failed worker node and re-schedule them on available healthy nodes.

If the worker node becomes available again, its services (like kubelet and Docker) are restarted, and its kubelet starts sending heartbeats to the master node. When the master node receives these heartbeats, it marks the node's state as *NodeReady*. Kubelet on the recovered node then communicates with the master node to fetch the information about the pods deployed

on that worker node. Now, either of two things can happen: if the worker node has become available before the elapse of the pod-eviction-timeout timer, kubelet restarts the pods on the recovered worker node; else, if the node has recovered after the pod-eviction-timeout, then the pods have already been restarted on other nodes, and so the kubelet communicates with the container runtime to delete the information about those containers from the recovered worker node as a garbage collection measure.

2.6 Upgrades in Kubernetes Clusters

Upgrades are an essential part of the software lifecycle. There can be various reasons a system might require an upgrade, such as bug fixes, new functionality, an upgrade of the underlying technology, etc. An upgrade can cause service disruptions and is an essential aspect to be considered for High Availability.

As Kubernetes operates on a container level, everything at the container level; and beneath; can be upgraded. Figure 2-3 shows the possibilities (context) of upgrades in a Kubernetes cluster. The upgrade of hardware and operating system is not covered in the scope of the work.

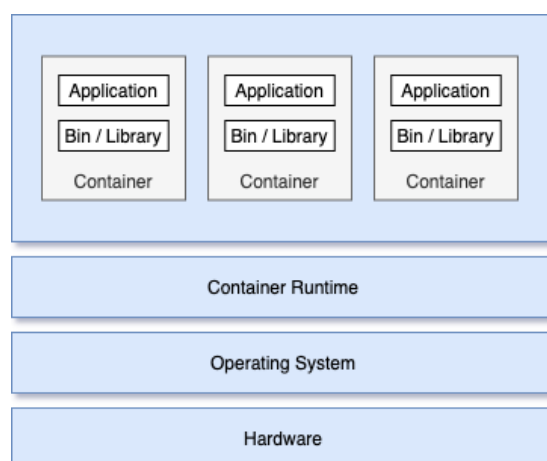


Figure 2-3: Context of upgrade in Kubernetes

Upgrade in a Kubernetes cluster can be divided into the following three levels:

2.6.1 Kubernetes' Cluster Version Upgrade

A Kubernetes cluster is created by various tools available to the practitioner, like kubeadm [13], kOps [14], and GKE [15]. A Kubernetes cluster version upgrade involves upgrading the Kubernetes version of the master nodes (i.e., the master node components), followed by the Kubernetes version upgrade of the worker node. As the master node is responsible for the management of the cluster, and worker nodes for hosting the application instances, so upgrading the Kubernetes version of these nodes would impact the cluster orchestration and the application deployed in its cluster.

2.6.2 Kubernetes' Application Upgrade

Kubernetes manages the containerized applications deployed in its cluster; these applications can either be Stateful or Stateless. Kubernetes has native upgrade strategies defined for an application that depends on the kind of application- Deployment for Stateless application or StatefulSet for Stateful application as mentioned in its deployment configuration. A Kubernetes application upgrade involves upgrading the deployed application's version using the mentioned native strategies.

2.6.3 Container Runtime Upgrade

The container runtime tool is installed on each node of a Kubernetes cluster, and it is responsible for creating and managing the containers. Kubelet communicates with the container runtime tool to get the status of the containers running on a worker node for the creation and deletion of the pod and other pod's management-related operations, so the upgrade of a container runtime tool can lead to possible disruptions in the services provided by these tools. A Kubernetes container runtime upgrade involves manually upgrading the container runtime tool's version on each node of a Kubernetes cluster.

Chapter 3

Evaluation Method and Setup

This chapter introduces the approach of evaluating upgrades in a Kubernetes cluster. In Section 3.1, we discuss the overall process of the evaluation; then, we describe the cluster architecture and its settings in Section 3.2. Section 3.3 introduces different evaluation scenarios and associated experiments considered in this thesis. The metrics are presented in Section 3.4.

3.1 Evaluation Method

In this thesis, we performed evaluations for all the identified levels of upgrades in a Kubernetes cluster, as mentioned in Section 2.6. These levels of an upgrade are evaluated with respect to service availability, and their impact is quantified through experiments. Following are the research questions we are aiming to answer:

RQ1: How does an upgrade impact the service availability of a hosted application?

RQ2: How does an upgrade impact the recovery from a failure?

RQ3: What is the impact of a failure of the upgrade process, and what recovery measures are taken?

Depending on the upgrade level-specific characteristics, these questions may be refined further in the subsequent chapters.

Kubernetes deploys a containerized application instance as a pod. A pod goes through different phases in its lifecycle caused by various factors, such as scaling events, pod failure,

application container failure etc. Kubernetes (via Kubelet) monitors the state of the pod and publishes relevant events if it detects any change in the state of the pod. Thus, it is crucial to understand the pod's lifecycle through these events to evaluate the impact on service availability. These events and their relations with each other are further explained in Section 3.4.

For the evaluation, we conducted experiments covering several scenarios and measured the defined availability metrics (through these events) to address the research questions. Each scenario is repeated ten times, and the average values of the measurements are shown in the tables of Chapters 5 through Chapter 7. The unit of measurement for all the experiments is in seconds, with an accuracy of milliseconds. The process of collection of metrics is automated by the “Auto-metric collector” tool. The architecture and functioning of the tool are discussed in detail in Chapter 4. The experiment setup and the availability metrics are explained in the following sections.

3.2 Evaluation Cluster Setup

In this section, we discuss the cluster architecture and its configuration for the performed experiments.

3.2.1 Cluster Architecture

For the experiments, we considered two cluster architectures, as shown in Figure 3-1, a single-master Kubernetes cluster with two worker nodes and a Highly Available (HA) Kubernetes cluster with three masters and two worker nodes. In a HA Kubernetes cluster setup, the number of master nodes runs in an odd number, with three being the minimum number. Thus, it is necessary to have at least three different machines (running as master nodes) to configure the HA Kubernetes cluster.

Since the nodes of clusters must be on different machines for experimenting with high availability, we used Virtual Machines (VM) to imitate multiple physical machines. Kubernetes version 1.18.1 was installed on all these nodes that run Ubuntu 18.06 as Operating System (OS). Docker version 19.06 was used as a container runtime to spin the containers. All the experiments are performed under the default configuration of Kubernetes.

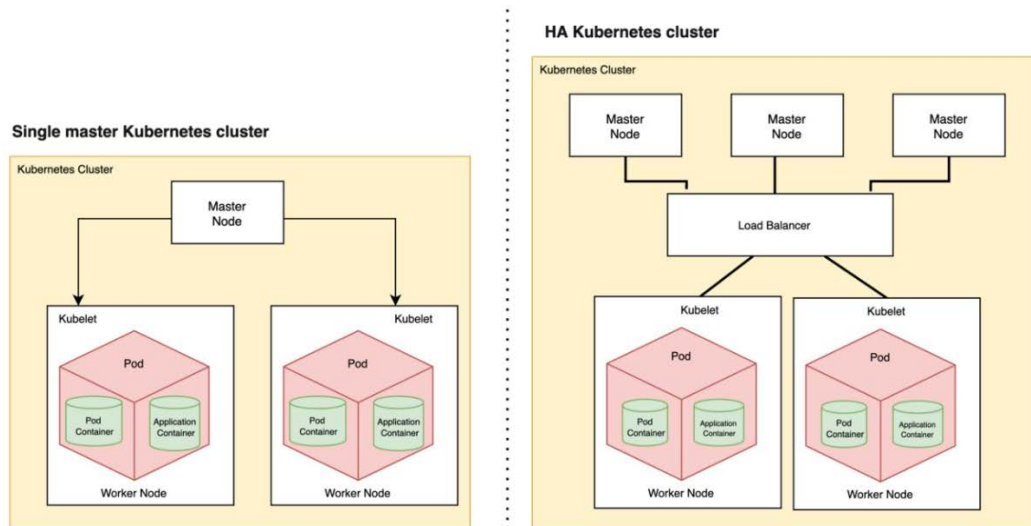


Figure 3-1: Evaluation Cluster Architecture

These architectures may be refined in the subsequent chapters based on the scenarios considered for different upgrade levels.

3.2.2 Application Deployment

For all the experiments, two pods are deployed in the cluster, with one pod running on each of the two worker nodes.

We considered stateless and stateful applications: a simple NGINX webserver application was selected as the stateless application, while a VLC video-streaming application was chosen as the stateful application.

The State Controller managed the availability of the stateful application described in [16] and presented in Figure 3-2, it runs on a separate VM in our cluster setting. The pods of

the stateful application are deployed in pairs, and the State Controller assigns them active and standby labels. While the active pod (pod with the active label) actively provides the streaming service, the standby pod (pod with the standby label) replicates the state of the streaming service provided by the active pod. If the active pod fails, the State Controller performs a failover to the standby pod, and this way, it manages the service availability.

In our experiment setting, for both single master and HA master cluster architecture, the State Controller runs on a separate VM and communicates with the master node to perform failover.

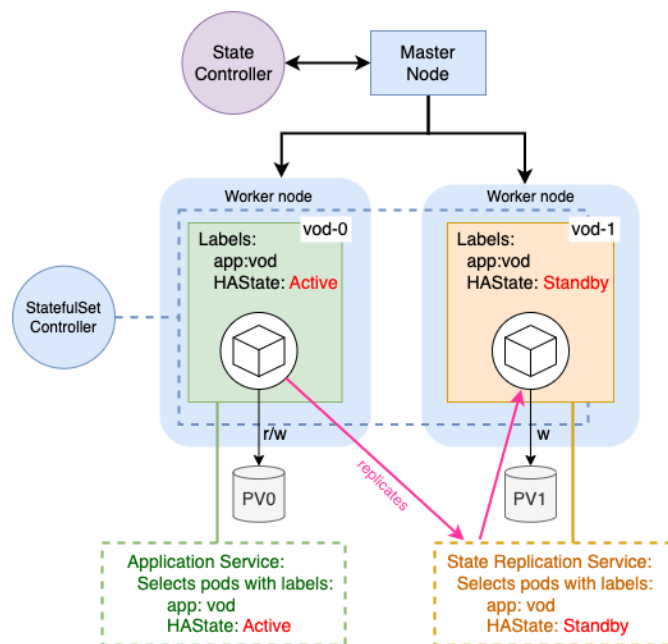


Figure 3-2: An example of Stateful Application deployed with the name “vod” managed by State Controller

3.3 Evaluation Scenarios

This section describes the scenarios considered and the experiments performed to answer the defined Research Questions mentioned in Section 3.1.

3.3.1 Upgrade Scenario

Pods may get terminated and re-created during upgrades in a Kubernetes cluster. For example, in a kOps managed Kubernetes cluster during the Kubernetes version upgrade of a worker node, the pod running on it is gracefully evicted and scheduled on an available worker node. RQ1 focuses on the impact on service availability during an upgrade; to answer this, we perform the upgrade and observe/measure its impact on the deployed application instances.

3.3.2 Failure Scenarios

Kubernetes provides a self-healing capability for the deployed application as a most significant feature for maintaining their high availability. It continuously monitors and repairs its object (such as a pod). Kubernetes's kube-apiserver communicates with the Kubelet component running on each worker node to get the status of the worker node and the containers running on it. The following experiments have been performed to simulate the failure scenarios; for evaluating the service availability guaranteed by Kubernetes through its recovery actions.

Pod Failure: Pod container runs as a process in the OS and may crash. In this scenario, the failure is simulated by killing the pod process from the OS.

Application Container Failure: Since each application container runs as a process on the OS, we simulate application container failure by killing its associated process.

Node Failure: In this scenario, a node hosting a pod is failed. This scenario is simulated by Linux's *reboot* command.

Upgrade Process Failure: In this scenario, the ongoing upgrade process is failed. The simulation of this failure differs on the type of upgrade and is refined in Chapters 5, 6, and 7 respectively.

3.4 Metrics for Evaluation

In this section, we discuss the various events identified during these experiments; and use their relationships to define the respective metrics presented in Figure 3-3 for stateless application and Figure 3-4 for stateful application.

During an ongoing upgrade, we evaluate the impact on application services in terms of service outage and service degradation, shown in Figure 3-3 for stateless application and Figure 3-4 for stateful application. To evaluate this impact, we define these metrics below.

Service Outage: The duration for which the application was not providing the service, i.e., the number of pods actively providing the application service, was zero.

Service Degradation: The duration for which the actual number of pods actively providing application service was less than the desired number of pods.

In the event of failure, we measure the failure recovery in terms of failed unit outage time. To evaluate this, we define the metrics as shown in Figure 3-3 for stateless application and Figure 3-4 for stateful application, and their relation is summarized below:

Detection Time: The time between the failure event we introduced and when Kubernetes detects the failure event.

Repair Time: The time between the detection of failure event and when the pod failed due to the failure event is repaired.

Assignment Time: The time between the re-creation of the pod and when the service provided by the failed pod is available again.

Failed unit outage Time: The duration for which the failed unit was not providing service. It is the sum of detection time, repair time and assignment time.

The Figure 3-3 shows the way metrics are measured for a stateless NGINX application.

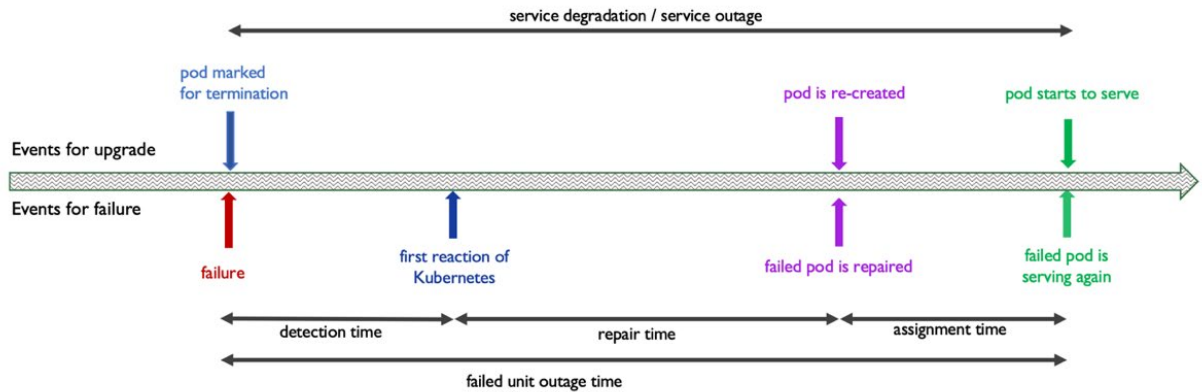


Figure 3-3: Metrics for stateless NGINX application

The following Figure 3-4 shows the way metrics are measured for a stateful video streaming application whose failover is managed by State Controller (SC).

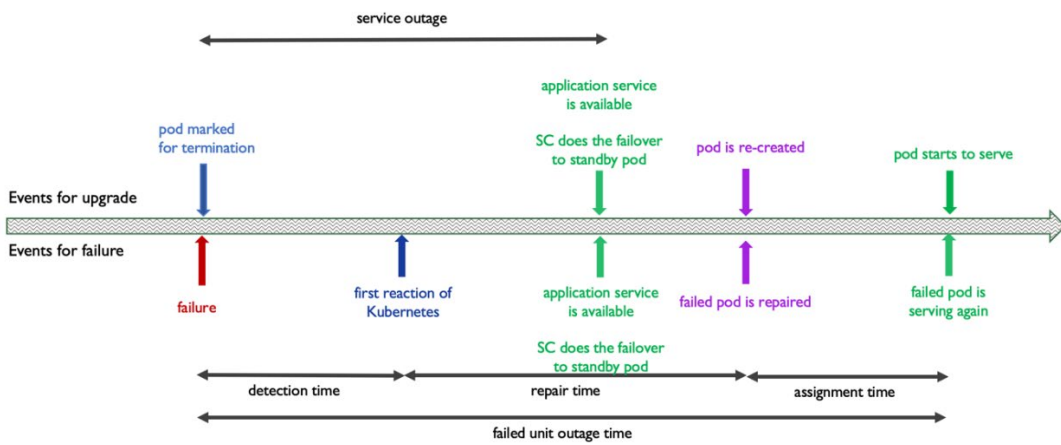


Figure 3-4: Metrics for stateful video streaming application

Chapter 4

Auto-Metric Collector – a Tool to Automate the Process of Metric Collection

This chapter introduces the Auto-Metric collector tool developed to monitor the pre-defined events, collect their timestamps, and calculate metrics automatically. In Section 4.1, we briefly discuss the need for a tool for our research, followed by an evaluation of three of the existing popular open-source tools in Section 4.2. In Section 4.3, we explain the high-level architecture of the Auto-Metric collector, its integration with Kubernetes and the Filebeat-Elasticsearch-Kibana stack. We finally present our conclusion in Section 4.4, where we discuss the assumptions and limitations associated with the tool.

4.1 Problem Statement

Event monitoring, timestamp collection and metric calculation are not only tedious, but it requires significant manual efforts and is also prone to human errors. In an estimate, each of our manual experiments for a scenario takes around 10-11 minutes to capture data and calculate metrics; and another 10-12 minutes per set for data analysis and visualization. So, for every 100 scenarios, more than 18 hours of continuous work is needed to get the required metrics calculated and analyzed.

Though various open-source tools are available in the market to monitor the resources in a Kubernetes cluster, the question is: would those tools cater to the needs of our experiments with the much-needed granularity of milliseconds precision in all scenarios?

4.2 Evaluation of Existing Tools

To answer the previous question, we evaluated Prometheus [17], Kubernetes Dashboard [18] and cAdvisor [19], three of the popular open-source tools well integrated with Kubernetes. Following are the evaluation criteria we considered:

- Granularity: The metric data needed for our experiment should be in milliseconds precision.
- Applicability: The ability of the tools to collect events specific to our work.
- Controllability: The ability to control the metric collection at a specific time.

Table 4-1 summarizes the monitoring tools' evaluation against the defined criteria.

Table 4-1: Evaluation of popular open-source monitoring tools

Tools \ Evaluation criteria	Prometheus	Kubernetes Dashboard	cAdvisor
Granularity			
Applicability			
Controllability	✓		✓

With the above evaluation, it is evident that existing tools do not meet our needs. The common issue with open-source tools is the lack of fine-grain monitoring. Since Kubernetes stores the timestamp of the events to seconds precision, most tools available in the market expose these events through metrics. Furthermore, these tools are mostly suited for resource and performance monitoring and utilize the metrics exposed by Kubernetes; consequently, collecting specific events and metrics is not possible. Thus, designing a new tool is essential for the course of the research in terms of automation and further developments.

4.3 Collecting Metrics Via Auto-Metric Collector

Auto-Metric collector is a tool designed to reduce the manual effort in the metric (described in Section 3.4) calculation. Our tool is written in Golang and utilizes the client-go library to monitor the events happening in the Kubernetes cluster. In this section, we present a high-level architecture of the tool, explain the role of each of its components and then discuss the tool's integration with Kubernetes for monitoring the Kubernetes objects and Filebeat-Elasticsearch-Kibana (FEK) stack for data visualization.

4.3.1 Architecture

Figure 4-1 illustrates the high-level architecture diagram of the Auto-Metric collector tool. The different components involved in the tool's functioning are shown in the architecture, and their roles are explained later in this section.

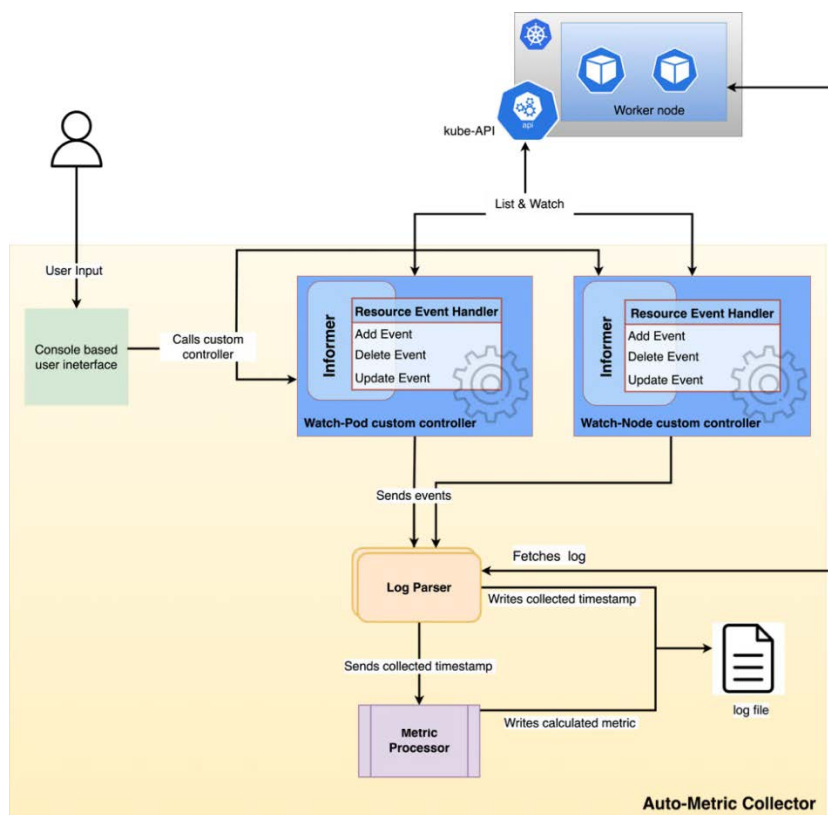


Figure 4-1: High-level architecture of Auto-Metric Collector

The roles of each component of the Auto-Metric collector are defined as follows:

- **Console based user-interface:** It is used to get the user input which determines the following:
 - The Kubernetes tool, i.e., kubeadm or kOps, is used to create a Kubernetes cluster and sets up the cluster configuration to be used by the Log Parser.
 - The Kubernetes object (pod/node) needs to be monitored.
- **Custom Controller:** The tool monitors the events associated with the user-selected Kubernetes object. This monitoring is possible via the custom controller. A custom controller is a controller that acts upon the native Kubernetes resources and is used to add new features. The custom controller is implemented using the client-go library and utilizes its features for monitoring the state change of Kubernetes objects. It has an Informer component which lists and watches different events associated with the object specified in the customer controller. When the Informer identifies a change in the state for its selected Kubernetes object, depending on the type of change, the Add/Update/Delete Event functions of the Resource Event Handler are triggered, which further sends the state change information to the Log Parser component.

Since a custom controller is dedicated to a specific Kubernetes object, so according to the need of our experiments, we have two custom controllers to monitor the pod and node.
- **Log Parser:** Since the event timestamps of Kubernetes is in seconds precision, we have created a Log Parser component that the custom controller triggers, as shown in Figure 4-1, and performs the following functions to collect event timestamps stored in milliseconds precision from the components that reports those events:
 - The Docker/CRI-O and kubelet logs are collected via a script on each worker node.

- The log is then parsed to capture the reported event's timestamp, which is written in the log file, and this information is also forwarded to the Metric Processor component.
- Apart from collecting the events reported by Kubernetes, the Log Parser is also responsible for collecting the timestamp of the failure. Since we manually inject failure via a script, the script stores the timestamp in a file on a worker node (where failure is injected), which is collected by the Log Parser.
- **Metric Processor:** The role of the metric processor is to calculate the defined metrics (as mentioned in Section 3.4) using the event timestamps received from Log Parser. It finally writes the calculated metrics data into the log file.

4.3.2 Operations

Figure 4-2 shows the integration of our Auto-Metric collector tool with the Kubernetes cluster and with the FEK stack. These are discussed further in the two sub-sections.

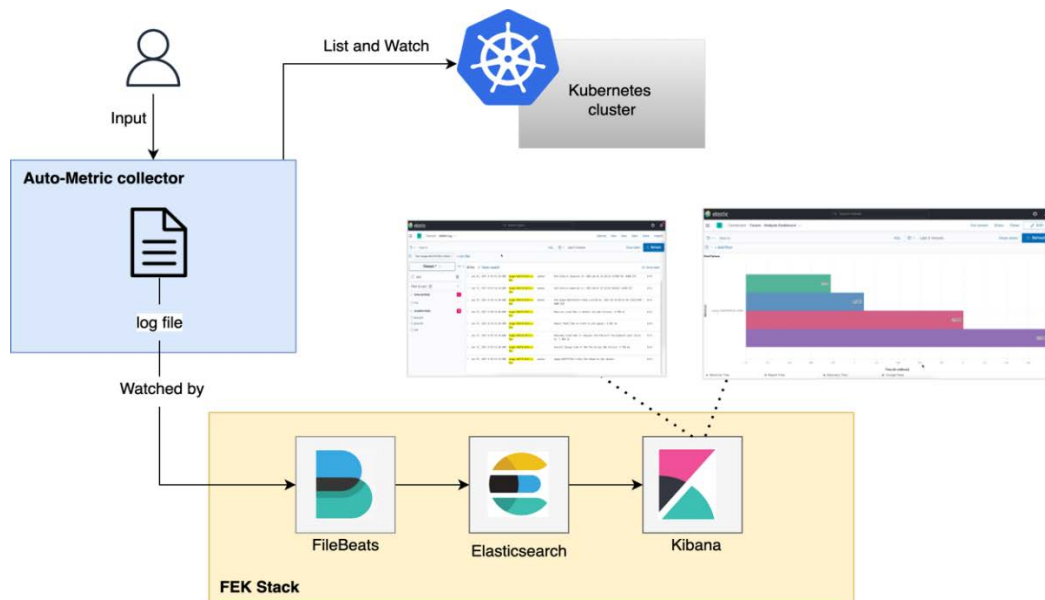


Figure 4-2: Integration of Auto-metric collector with Kubernetes and FEK stack

4.3.2.1 Integration with Kubernetes for Event Monitoring and Timestamp Collection

To allow the integration of the Auto-Metric collector with the Kubernetes cluster, we used the client-go library of Kubernetes which interacts with the REST interface exposed by the Kubernetes kube-apiserver component, as shown in Figure 4-2, to access the Kubernetes objects (pods/nodes). In our implementation, we retrieve the configuration of the Kubernetes cluster and create a client that communicates with the kube-apiserver to monitor the events related to the Kubernetes object.

4.3.2.2 Integration with FEK Stack for Data Visualization

The log file generated by the tool acts as an input to the FEK stack, as shown in Figure 4-1. The following three components of the FEK stack work together for data visualization in the form of a Dashboard:

- **Filebeat:** It is a lightweight shipper and acts as the starting point for the FEK stack. It monitors the log file generated by the Auto-Metric collector tool for any new content, aggregates the events and forwards the aggregated data to Elasticsearch for indexing [20].
- **Elasticsearch:** It acts as a distributed document store to index the data received from Filebeat. It provides near real-time search and analytics for all types of data; structured or unstructured text, numerical data, or geospatial data [21]. This indexed data is forwarded for data visualization by tools such as Kibana.
- **Kibana:** It lets you filter relevant data and then use that data to build a dashboard comprising logs and graphs.

4.4 Conclusion

The implemented Auto-metric collector tool caters to the need for fine-grain evaluation required in our research while reducing the human effort. For every 100 scenarios with existing

capability, the Auto-Metric Collector tool will save approximately 16-18 hours of human effort. This automation of the metric collection also avoids discrepancies due to human error. However, the following points are assumed for the functioning of the tool:

- The tool should have admin rights and access to all the nodes in the Kubernetes cluster. It requires interaction with the cluster for the collection of event timestamps through kubelet logs and Docker logs. This operation will not be possible if the tool cannot access the node.
- Time is a significant factor in the evaluation. All the nodes in the Kubernetes cluster must follow the same time zone. To achieve a time-synced Kubernetes cluster, NTP (Network Time Protocol) [22] is configured on each node of the cluster. This is a popular protocol widely used in the industry by practitioners.

The following are the limitations in terms of the functionality offered by the tool:

- Currently, the tool can only monitor one single user-selected Kubernetes object at-a-time for calculating metrics. However, it can be further extended to monitor multiple Kubernetes objects simultaneously.
- Currently, the metrics collected by the tool are specific to our experiments. However, it can be further extended for customized metrics.

Chapter 5

Kubernetes Cluster Version Upgrade

In this chapter, we quantitatively evaluate the Kubernetes cluster version upgrade managed by kOps and kubeadm tools provided by CNCF. In Section 5.1, we explain the current practice of upgrade of these tools. In Section 5.2, we discuss the research questions we aim to answer before discussing the results and analysis of the performed experiments and assessing the achievable service availability during upgrade. In Section 5.3, we provide potential improvements to some of the identified issues and conclude our evaluations in Section 5.4.

5.1 Current Practice of Upgrade

The Kubernetes cluster version indicates the major, minor, and patch versions [23]. A minor version of Kubernetes is released every three months and maintained for nine months. During these nine months, patch versions are released every 1-2 weeks. These different versions are released to fix bugs, deploy security patches, and/or add new functionality to improve the overall experience. Thus, deploying these enhancements requires regular Kubernetes cluster version upgrades.

Various tools like kubeadm and kOps (Kubernetes Operation) are available to the practitioner to deploy a Kubernetes cluster. These tools also help in provisioning the Kubernetes cluster. In addition, for upgrades, a recommended upgrade flow [24] has been described in the Kubernetes documentation. We have evaluated and analyzed the upgrade process flow of these tools and have presented them in Figure 5-1, Figure 5-2 and Figure 5-4. In these figures, the blue indicates manual operations to be performed by an administrator, while yellow shows the

automated procedures triggered in response to the manual operations. Only actions significant to our work are shown in detail.

5.1.1 Kubeadm

Kubeadm is a tool to create a minimum viable Kubernetes cluster [25]. It can also be used to upgrade (or downgrade) such a cluster. When a Kubernetes cluster is created using kubeadm, it constructs a cluster configuration file referred to as *kubeadm-config* to be used by other management components. In addition, on one of the master nodes (usually the first created), kubeadm also writes a configuration file called *admin.conf* for administrative purposes. These files are referred to during the Kubernetes cluster version upgrade process.

The Kubernetes cluster version upgrade process using kubeadm is a manual process and consists of the following steps.

The upgrade process starts by upgrading the master node bearing the *admin.conf* file (usually the master node where the Kubernetes cluster is initialized). As shown in Figure 5-1, we first upgrade the kubeadm tool on this master node, which is necessary as kubeadm enforces the version skew policy. The upgrade process continues further with upgrading the master node components. As part of this step, kubeadm performs a health check of the cluster and then pulls the updated manifest file (for the desired Kubernetes cluster version) of the master node components (running as static pods on the master node). Then, a master node component is selected for an upgrade, where its new manifest file is moved to the current config folder while the old manifest file is backed-up. Next, the kubelet component restarts the static pods of the master node component to reflect the upgraded version. This process is repeated for all master node components, one at a time. Then the master node is drained of any existing workload, which also cordons the master node. This step is followed by updating the configuration of the kubectl client and kubelet agent. Subsequently, the master node is un-cordoned to start accepting the

orchestration workload again. Once the master node with the *admin.conf* file is upgraded successfully; it updates the associated *kubeadm-config* file with the new Kubernetes cluster version. The subsequent master nodes, in the case of an HA-master Kubernetes cluster, refer to the *kubeadm-config* file and are upgraded to the Kubernetes cluster version mentioned in the file in a similar manner except for the update of the *kubeadm-config* file, which does not apply to them. We have summarized the flow of upgrading the master nodes using kubeadm in Figure 5-1.

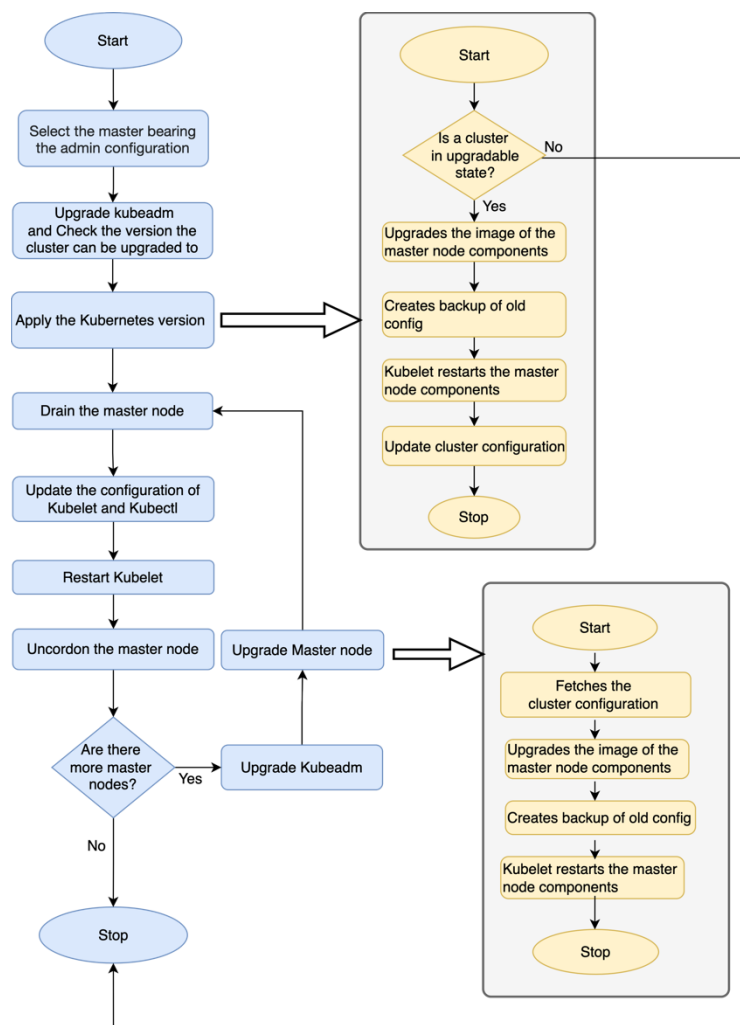


Figure 5-1: Master node upgrade process using kubeadm

Once the master node(s) is(are) upgraded, the upgrade of worker node(s) can follow.

Figure 5-2 presents the upgrade process flow of worker nodes. The worker node upgrade also

starts by upgrading the version of the kubeadm tool. Next, in the worker node upgrade step, which is specific to the upgrade of the node, the upgraded *kubeadm-config* file is fetched so that the kubelet configuration can be updated with the correct Kubernetes cluster version. Then, we drain the worker node, which evicts the pods running on that worker node. The step to drain the worker node prepares the node for its maintenance by gracefully evicting the existing workload and cordoning the node to make it non-schedulable. It is a precautionary and optional step to avoid service disruption when the component monitoring its state, i.e., kubelet, will be upgraded next. Then, we update the kubectl and kubelet configurations with the updated configuration fetched at the upgrade step and restart the kubelet component to reflect the change. If the step to drain the worker node(optional) is executed, it also cordons it, so as the next step, we need to un-cordon the worker node so that it starts to receive new workload requests. The remaining worker node(s) must be manually upgraded, following the same steps.

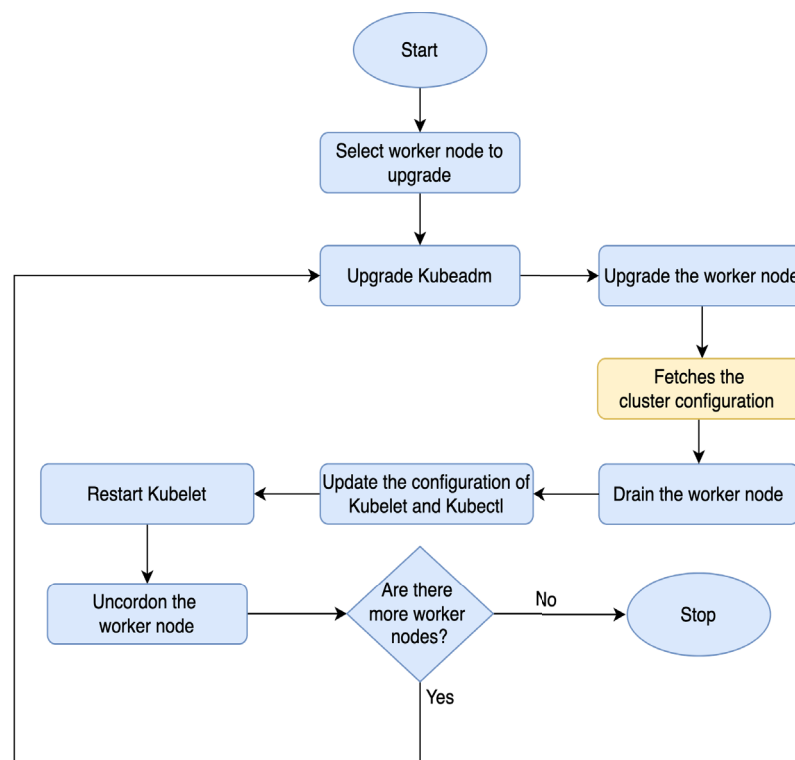


Figure 5-2: Worker node upgrade process using kubeadm

5.1.2 kOps

kOps is a tool that automates the provisioning of a Kubernetes cluster [14]. It provides full support for Amazon Web Services (AWS) and uses cloud provider features. For example, in AWS, a group of nodes with a common purpose is bound together as an entity called an instance group [26]. Each instance group has a role specified by the administrator at the time of creating the Kubernetes cluster. The master nodes are in the instance group with the role “*Master*” while the worker nodes are part of the instance group with the role “*Node*”. These roles play an essential part in deciding the order of upgrade when the Kubernetes cluster upgrade is initiated, where kOps utilize the instance group capability of AWS to upgrade the Kubernetes cluster in a rolling update fashion.

kOps also allows previewing the changes that a specific operation would result in a Kubernetes cluster. If a kOps command is used without the `--yes` option, then kOps display the preview of the changes for that specific command rather than executing it. It is recommended to preview the changes made by a particular command before updating the state of the cluster.

In a kOps-managed Kubernetes cluster on AWS (see Figure 5-3), the master nodes and the worker nodes are encapsulated in their respective instance groups, which may be distributed across different AWS zones.

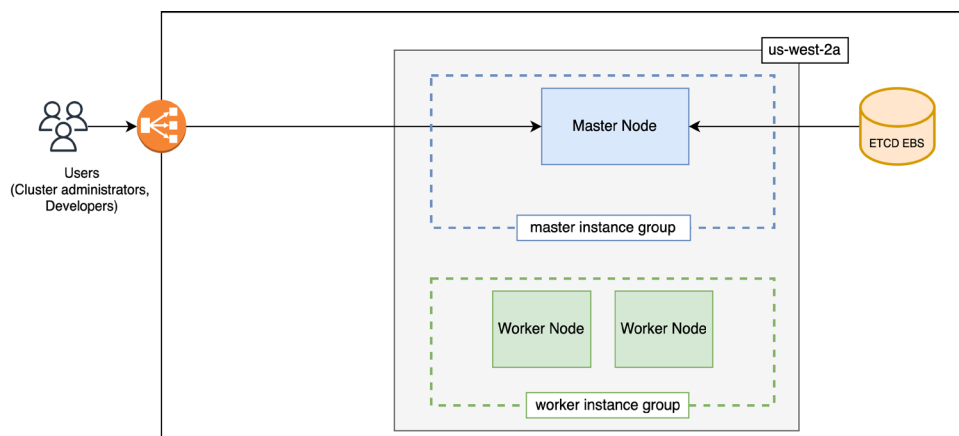


Figure 5-3: An example of kOps managed single master Kubernetes cluster on AWS

Figure 5-4 shows the Kubernetes version upgrade process flow for a kOps managed Kubernetes cluster. Before starting the cluster upgrade, it is recommended to update the kOps tool to ensure compatibility with the intended Kubernetes cluster version and enforce the version skew policy properly. To upgrade the Kubernetes cluster version, the local cluster specification file is first updated with the desired Kubernetes cluster version. This update can either be performed manually or automatically by kOps. These changes are also propagated to the cluster specification file on AWS, which instance groups refer to during an upgrade. The configuration changes are applied to the instance groups of the cluster in a rolling update fashion, i.e., the order in which the instance groups upgrade depends on their roles. As shown in Figure 5-4, the upgrade starts with the master instance groups (instance group with the role of “Master”).

During the rolling update of the cluster, first, one of the master instance groups is selected for the upgrade, and its master node is upgraded. By default, kOps keep one master per master instance group. This upgrade involves the drain and termination of the selected master node in the current version and then creating a new master node with the desired Kubernetes version. Then, on this newly created master, all the pods specific to the master node are created, and once these pods become ready, that master node becomes available. Then, another master instance group (if any) is selected for an upgrade, and the same upgrade process is repeated to upgrade its master node.

Once all the master instance groups are upgraded, the upgrade of the worker instance groups follows. First, a worker instance group is selected, then the node(s) in that instance group is upgraded. Unlike the master node upgrade process, during the worker node upgrade, the worker node selected for upgrade is initially detached. Next, a new worker node with the desired Kubernetes version is created; however, at this time, this worker node will not be hosting pods. Then, the worker node selected for an upgrade (running on the current Kubernetes

version) is drained and terminated, which leads to the graceful termination of the pods running on the old worker node and the creation of the pods on the newly created worker node. Once all the worker nodes in an instance group are upgraded, another worker instance group (if any) is selected for an upgrade, and its worker nodes are upgraded similarly. Once all the worker instance groups are upgraded, the rolling update process completes.

In addition, kOps perform cluster validation during the upgrade to ensure that the state of the cluster is unaffected by the node update.

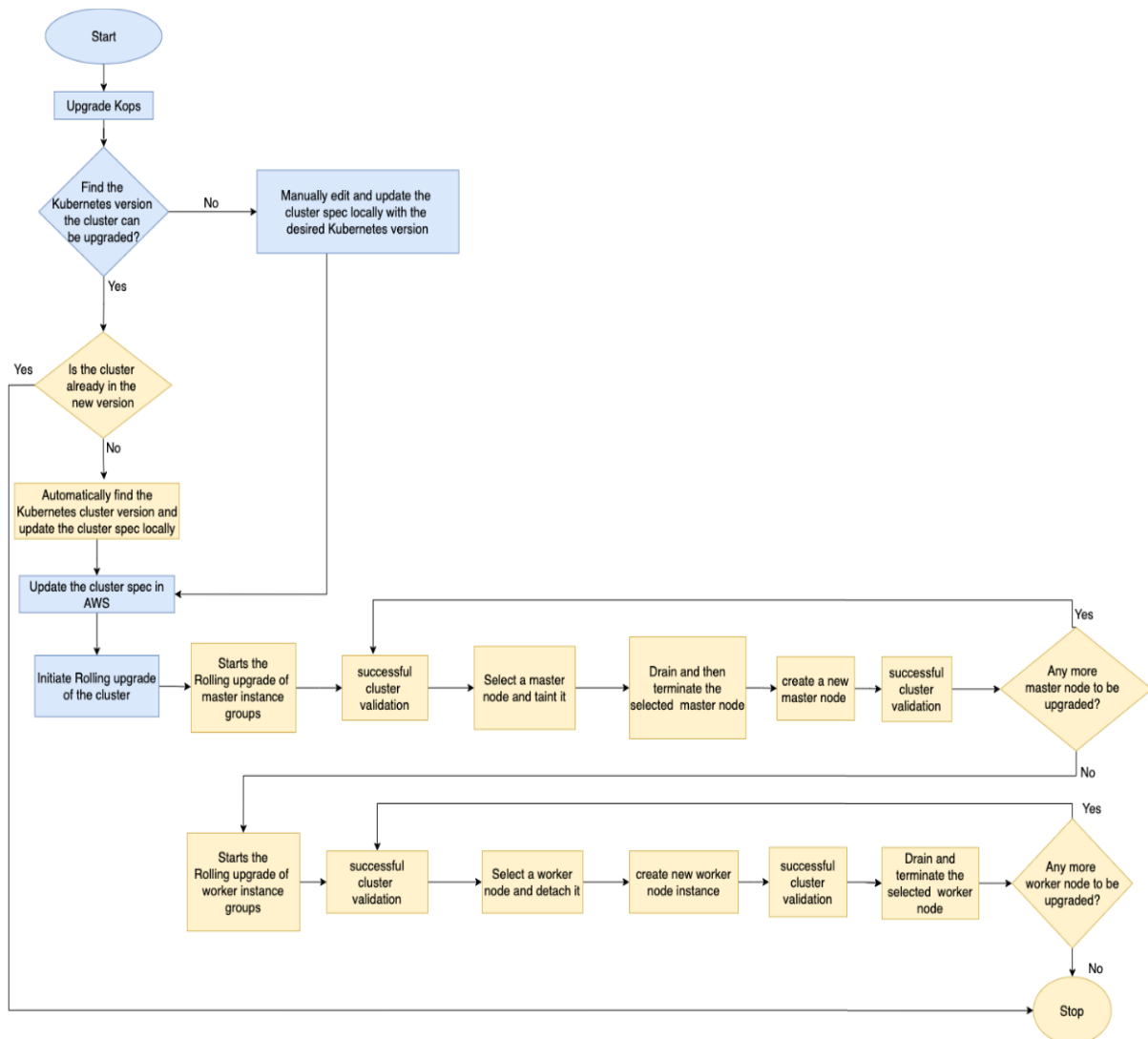


Figure 5-4: Upgrade process flow in a kOps-created Kubernetes cluster

5.2 Evaluation of Kubernetes' Cluster Version Upgrade

Kubernetes community claims that the services availability of the deployed application is maintained by Kubernetes through its self-healing functionality. However, upgrading the Kubernetes cluster version impacts the different nodes in the cluster, through which it may also impact the cluster orchestration and the service availability of the deployed applications. So, we state the following hypothesis:

H₁: High Availability of the application deployed in a Kubernetes cluster is impacted during Kubernetes' cluster version upgrade, especially in the presence of failure.

So, to test H₁, we evaluated the impact of upgrading the Kubernetes version of kubeadm and kOps created Kubernetes cluster, from the perspective of the service availability of applications deployed in the cluster. We also considered failures during the upgrade; failures of the pod, worker node and the failure of the upgrade process itself. Finally, we have defined the following research questions for the evaluation:

RQ1: What is the impact of a Kubernetes cluster version upgrade on the application services?

RQ2: What is the impact of Kubernetes cluster version upgrade in the presence of failure on the application services and recovery

1. from a pod failure?
2. from a worker node failure?

RQ3: What is the impact of a failure of the Kubernetes cluster version upgrade process, and what are the recovery measures taken?

To answer these research questions, we had a cluster setup whose architecture details are discussed in Section 3.2 and illustrated in Figure 3-1. Since this chapter focuses on evaluating Kubernetes cluster version upgrade, our setup has a Kubernetes cluster, created, and managed by kubeadm and kOps tool, in a single master and HA master Kubernetes cluster. Docker

19.06 was integrated with these clusters to spin the containers. We conducted experiments whose scenarios are discussed in Section 3.3, and metrics are presented in Section 3.4. The following sub-section presents the results and analysis of the performed experiments when the Kubernetes was upgraded from version 1.18.0 to version 1.20.0. All our experiments are performed with the default configuration of Kubernetes. The process of collection of metrics was automated by the Auto-Metric collector tool, which is explained in Chapter 4.

5.2.1 RQ1: Evaluate the Impact on Application Services during Kubernetes' Cluster Version Upgrade

In this sub-section, we explain the different experiments performed to answer RQ1, then we discuss the results of these experiments and present our analysis. As discussed earlier, in a Kubernetes cluster, first, a master node(s) is upgraded followed by the upgrade of worker node(s), so, for both kubeadm and kOps, we present the results and analysis of the performed experiment in the same order.

5.2.1.1 Experiments

The experiment aimed to evaluate the service degradation/outage caused by the Kubernetes cluster version upgrade. We conducted the experiments on both; the single-master Kubernetes cluster and the HA-master Kubernetes cluster, created and managed by kubeadm and kOps. The events related to the state changes of the two application pods were monitored during an upgrade to calculate the impact on application services availability illustrated in Section 3.4.

For stateful application, we upgrade the worker node bearing the pod with the active label (the pod actively providing the application service) to measure the maximum impact on the application service.

5.2.1.2 Evaluating the Impact during the Master Node Upgrade

For both kubeadm and kOps, during the master node upgrade, in a single-master and the HA-master Kubernetes clusters, we do not observe any change in the state of the application pods. As pods reside on the worker nodes, their state does not have any impact during the upgrade of master nodes.

5.2.1.3 Evaluating the Impact during Worker Node Upgrade of Kubeadm Managed Kubernetes Cluster

In the case of kubeadm, as mentioned in the worker upgrade flow in Figure 5-2 and sub-section 5.1.1, in a worker node upgrade step, it fetches the latest configuration from the *kubeadm-config*. In this step, we observed no change in the state of the pods, thus no service outage/degradation. The next step involves draining the worker node, which ensures that the worker node has no running pods during its maintenance. Though the step to drain causes service outage for the stateful application and service degradation for the stateless application, this step is a node maintenance step. It is optional, so we do not consider the application pod's state change in this step.

5.2.1.4 Evaluating the Impact during Worker Node Upgrade of kOps Managed Kubernetes Cluster

In the case of kOps, during a worker node upgrade for a stateless application, we observed a service degradation of 2.072 seconds. Based on our analysis, this happens because of the worker node upgrade flow of kOps mentioned in sub-section 5.1.2, where the new worker node with desired Kubernetes version is created before the old worker node is terminated. Once the new worker node is created, Kubernetes terminates the pods running on the old worker node and creates them on the new worker node. Due to the nature of the stateless application, this does not cause any service outage, as the pod running on other worker continue to provide

the application service, but this causes service degradation. The time Kubernetes takes to create a new pod on the newly created worker node constitutes the duration for which the actual number of pods is less than the desired number of pods, i.e., service degradation of 2.072 seconds, shown in Table 5-1.

For the stateful application, the application experiences a service outage of 0.681 seconds during worker node upgrade (hosting active pod). As analyzed, this happens because of the mentioned worker node upgrade process of kOps. When the new worker node with desired Kubernetes version is created, Kubernetes initiates the termination of the old worker node, during which the pods running on it are gracefully terminated and created on the new worker node. Since the worker node selected for the upgrade hosts an active pod, the pod termination causes the VLC application to stop streaming video. Simultaneously, the State controller receives the information of active pod termination and performs failover to the standby pod running on another worker node. The observed service outage is the time the State Controller takes to complete the failover to the standby pod; the measurements are reflected in Table 5-1.

Furthermore, in our experimental setting, since we considered two application pods, so, for stateful application, we had one pair of the active-standby pod. This means only one pod actively provides the services while the other is a standby pod, so service degradation is not applicable for stateful application.

Table 5-1: kOps: service outage and service degradation during worker node upgrade

Scenario	Kind of application	Metrics (unit: seconds)	
		Service outage	Service degradation
Worker node upgrade	Stateless application	0	2.072
	Stateful application	0.681	N/A

5.2.2 RQ2-1: Evaluate the Impact of Kubernetes' Cluster Version Upgrade in the Presence of Pod Failure, on its Failure Recovery Actions and on Application Services

In this sub-section, we explain the different experiments performed to evaluate the impact of upgrade process of kubeadm and kOps managed Kubernetes cluster. We first present the results and analysis of the impact during master node upgrade and then the worker node upgrade.

5.2.2.1 Experiments

The experiment aimed to evaluate the impact of the upgrade of the master/worker node on the recovery from the pod failure and on the services of hosted applications. In this experiment, pod failure was simulated by killing the pod process running on the OS.

For better comparison and understanding of these impacts, initially, pod failure was injected in the absence of an upgrade, which gave a clear picture of the time needed to restore the failed pod. Then, a pod failure was injected during the master/worker node upgrade to evaluate the impact of the upgrade on the failure recovery.

For the stateful application, the pod with the active label (the pod actively providing the service) was failed to evaluate the maximum impact of the upgrade on the application services.

5.2.2.2 Evaluating Master Node Upgrade of Kubeadm Managed Kubernetes Cluster Impact on Failure Recovery and Service Degradation

In the case of kubeadm, for both stateless and stateful application, the recovery operations for pod failure (as mentioned in Section 2.5.1) are impacted for the single-master cluster. Kubelet, together with the master node, performs the repair operations for the failed unit, and

since the master node is being upgraded, recovery from pod failure is not possible. It leads to an increase in the repair time, causing an increased failed unit outage time of 7.070 seconds for the stateless application, as shown in Table 5-2, and 12.874 seconds for the stateful application, as shown in Table 5-3.

For stateless application, the observed failed unit outage time also represents the duration for which the actual number of pods providing application service is less than the desired number of pods, i.e., service degradation. Hence, the measurements are the same and represented in Table 5-2.

Table 5-2: kubeadm: stateless application pod failure during master node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time / Service degradation	Service outage
Without upgrade	Single Master	0.849	0.403	1.536	2.788	0
With upgrade		0.847	4.613	1.610	7.070	0
Without upgrade	HA Master	0.849	0.403	1.536	2.788	0
With upgrade		0.833	0.449	1.516	2.798	0

Impact on Service Outage

For a stateless application, we do not observe any service outage because the application service remains available, as the pod hosted by another worker node is available to provide it.

For stateful application, in the single-master cluster, we observe an increase in the service outage time. As the State controller depends on its interaction with the master node to initiate and completes the failover operation, these communications are not possible during the master node upgrade, leading to an increased service outage time of 8.497 seconds, as shown in Table 5-3. Once the master node is upgraded and becomes available, the State controller

completes the failover to the available standby pod and application service becomes available again.

Table 5-3: kubeadm: stateful application pod failure during master node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time	Service outage
Without upgrade	Single Master	0.943	2.583	1.593	5.119	1.612
With upgrade		0.683	10.591	1.600	12.874	8.497
Without upgrade	HA Master	0.943	2.583	1.593	5.119	1.612
With upgrade		0.927	2.581	1.592	5.100	1.574

5.2.2.3 Evaluating Master Node Upgrade of Kops Managed Kubernetes Cluster

Impact on Failure Recovery and Service Degradation

For the single-master cluster, we see a drastic increase in the failed unit outage time for stateless and stateful application, as shown in red in Table 5-4 and Table 5-5. This happens because of the master node upgrade process of kOps, (as mentioned in sub-section 5.1.2), which involves the termination of the master node in the current Kubernetes version and then the creation of a new master node in the desired Kubernetes version. So, only once all the components of the newly created master are ready, the master node becomes available to complete the failure recovery of the pod. The duration for which the master node is unavailable due to an upgrade causes an increase in the repair time, causing an increased failed unit outage time.

Also, for stateless application, the time duration for which the failed pod doesn't provide the service, the application's service is said to be degraded, as highlighted in red in Table 5-4.

Table 5-4: kOps: stateless application pod failure during master node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time/ Service degradation	Service outage
Without upgrade	Single Master	0.562	0.642	1.532	2.736	0
With upgrade		0.892	340.005	1.593	342.490	0
Without upgrade	HA Master	0.562	0.642	1.532	2.736	0
With upgrade		0.623	0.710	1.389	2.722	0

Impact on Service Outage

For a stateless application, as at least one pod is available (running on another worker node) to provide the application service, we do not observe any service outage. However, for stateful application during the master node upgrade, when an active pod is failed, we see a drastic increase in the service outage time. As analyzed, this happens because the State controller depends on its interaction with the master node to perform the failover operation. Due to the unavailability of the master node during its upgrade, these communications are not possible. Only once the master node is available, the State controller is able to complete the failover. It causes an increased service outage time of 346.661 seconds, as highlighted in red in Table 5-5.

Table 5-5: kOps: stateful application pod failure during master node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time	Service outage
Without Upgrade	Single Master	0.793	2.669	1.596	5.058	1.434
With Upgrade		0.789	351.984	1.708	354.481	346.661
Without Upgrade	HA Master	0.793	2.669	1.596	5.058	1.434
With Upgrade		0.713	2.615	1.605	4.933	1.353

In HA-master Kubernetes cluster, for kubeadm and kOps, we observe no additional impact on the failed unit outage time, service degradation, and service outage, as shown in

Table 5-2 to Table 5-5. It is due to the redundancy of the master nodes; while one of the master nodes is upgrading, another master node will be available to perform the recovery operation.

5.2.2.4 Evaluating Worker Node Upgrade of Kubeadm Managed Kubernetes Cluster

Impact on Failure Recovery and Service Degradation

For both stateless and stateful application, when we failed a pod running on a worker node while another worker node is being upgraded, we observed no additional impact on the failure recovery. It is because the kubelet component of the worker node whose pod is injected with failure is able to perform the failure recovery operation (as mentioned in sub-section 2.5.1) along with the master node. The measurement of this analysis is shown in Table 5-6 for stateless application and Table 5-7 for stateful application. Also, for stateless application, the time duration for which the failed pod doesn't provide the service, the application's service is said to be degraded, so service degradation equals failed unit outage time. The analysis of this measurement is the same for the HA-master Kubernetes cluster because of the redundancy of master nodes that does not create any difference during worker node upgrade.

Table 5-6: kubeadm: stateless application pod failure during worker node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				Service outage
		Detection time	Repair time	Assignment time	Failed unit outage time / Service degradation	
Without upgrade	Single Master	0.849	0.403	1.536	2.788	0
With upgrade		0.863	0.412	1.524	2.799	0
Without upgrade	HA Master	0.849	0.403	1.536	2.788	0
With upgrade		0.741	0.455	1.503	2.699	0

Impact on Service Outage

For a stateless application, we do not observe any service outage. During the worker node upgrade step, the pod running on the upgrading worker node is not impacted and keeps providing the application service. So, at this time, when another pod is injected with failure, at least one pod remains available (on the worker node being upgraded) to provide application service.

For stateful application, when an active pod running on a worker node is injected with the failure while the worker node hosting the standby pod is being upgraded, there is no additional impact on the service outage, as shown in Table 5-7. Based on our analysis, this happens because of the worker node upgrade process of kubeadm. As mentioned in sub-section 5.1.1 and Figure 5-2, the worker node upgrades step in kubeadm involves fetching the latest *kubeadm-config*. Since this step is not disruptive, the standby pod running on this worker node remains available during this upgrade step. So, during the upgrade step, when the active pod is injected with failure, the State controller performs the failover to the available standby pod. Thus, we see no additional impact caused by the ongoing upgrade. The results and analysis are similar for the HA-master Kubernetes cluster.

Table 5-7: kubeadm: stateful application pod failure during worker node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time	Service outage
Without upgrade	Single Master	0.943	2.583	1.593	5.119	1.612
With upgrade		0.711	2.625	1.604	4.940	1.382
Without upgrade	HA Master	0.943	2.583	1.593	5.119	1.612
With upgrade		0.749	2.599	1.549	4.897	1.420

5.2.2.5 Evaluating Worker Node Upgrade of Kops Managed Kubernetes Cluster

Impact on Failure Recovery and Service Degradation

For both stateless and stateful application in a single and HA-master cluster, there is no additional impact on the failure recovery, as shown in Table 5-8 and Table 5-9, respectively. Since the kubelet component responsible for reporting the failure is available on the worker node whose pod is injected with failure, it communicates with the master node and performs the recovery operation of the failed pod.

Also, for stateless application, we observed an increased service degradation of 5.141 seconds and 5.089 seconds, for Single and HA-Master cluster respectively, as highlighted in

red in Table 5-8. As analyzed, the observed service degradation represents the total time taken to recover the failed pod and the service degradation caused due to graceful termination of the pod on the upgrading worker node.

Impact on Service Outage

For a stateless application, we do not observe any service outage. As analyzed, when we inject pod failure on one of the worker nodes while another worker node is being upgraded, the graceful termination of a pod from the worker node that is being upgraded does not overlap with the failed pod's failure recovery duration. So, at least one pod is available to provide application service.

For stateful application, we observed an increase in the service outage time, as highlighted in red in Table 5-9. It happens because of the upgrade process flow of the worker node of kOps, as shown in Figure 5-4, where a new worker node with an updated Kubernetes version is created, and the old worker node is terminated. So, in our experiment, when the active pod is injected with failure while the worker node hosting the standby pod is being upgraded, the active pod is failed twice, causing the State controller to perform two failovers. The first failover is triggered due to active pod failure injection, where the State controller makes the standby pod available on the upgrading worker node as active. The second failover is triggered after some time when the old worker node is marked for termination as a part of the upgrade process, evicting the currently active pod on that worker node and making the State controller to perform the failover to the current standby pod (that failed earlier). It causes an increased service outage time of 1.995 seconds in a single-master Kubernetes cluster and 2.011 seconds in the HA-master Kubernetes cluster, as shown in Table 5-9.

Table 5-8: kOps: stateless application pod failure during worker node upgrade

Scenarios	Architecture	Metrics (unit: seconds)					
		Detection time	Repair time	Assignment time	Failed unit outage time	Service degradation	Service outage
Without upgrade	Single Master	0.562	0.642	1.532	2.736	2.736	0
With upgrade		0.609	0.658	1.617	2.884	5.141	0
Without upgrade	HA Master	0.562	0.642	1.532	2.736	2.736	0
With upgrade		0.589	0.661	1.527	2.777	5.089	0

Table 5-9: kOps: stateful application pod failure during worker node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time	Service outage
Without upgrade	Single Master	0.793	2.669	1.596	5.058	1.434
With upgrade		0.713	2.674	1.664	5.051	1.995
Without upgrade	HA Master	0.793	2.669	1.596	5.058	1.434
With upgrade		0.704	2.653	1.610	4.967	2.011

5.2.3 RQ2-2: Evaluate the Impact of Kubernetes’ Cluster Version Upgrade, on its Worker Node Failure Recovery Actions and on Application Services

In this sub-section we explain the different experiments performed to evaluate the upgrade process. For both kubeadm and kOps, we first present the results and analysis of the impact, during master node upgrade and then the impact during worker node upgrade.

5.2.3.1 Experiments

The experiment aimed to evaluate the impact of the Kubernetes cluster version upgrade on the recovery from a worker node failure. The worker node failure was simulated by issuing the Linux’s *reboot* command on the worker node. For a better comparison and to understand the impact of the upgrade on the recovery, initially, worker node failure was injected in the absence of the upgrade, which gave a clear picture of the time needed to restore the pod in the failed worker node. Then, a worker node failure was injected during the master/worker node upgrade to evaluate the impact of the upgrade on the recovery of the failed unit.

For stateful application, the worker node bearing the active pod was injected with failure with an aim to evaluate the maximum impact of the upgrade on the service outage.

5.2.3.2 Evaluating Master Node Upgrade of Kubeadm Managed Kubernetes Cluster

Impact on Failure Recovery and Service Degradation

For both stateful and stateless applications, in the event of a worker node failure during a master node upgrade in a single-master Kubernetes cluster, we observed that there is no additional impact on the failure recovery, as shown in Table 5-10 and Table 5-11 for stateless and stateful applications respectively. As analyzed, by the time the failed worker node reboots, the upgrade of the master node is completed. The master node reacts to the failure of the worker node by marking the worker node's state as *NodeNotReady*. Once the failed worker node is rebooted, the kubelet component of the worker node can communicate with the master node components to fetch the list of the containers associated with that node and completes the recovery operations of the pod failed on the worker node that was injected with failure. Also, as service degradation equals failed unit outage time for stateless application, there is no additional impact on service degradation, as shown in Table 5-10.

These measurements are similar for a stateless and stateful application in an HA-master Kubernetes cluster.

Impact on Service Outage

For a stateless application, during the master node upgrade step, while one of the worker node is failed, the pod running on another worker node is not impacted, so, there is at least one pod available to provide the application service. Thus, the application service remains available, and we do not observe any service outage.

For stateful application, when the worker node bearing active pod fails during the master node upgrade, we do not observe any additional impact on the service outage when compared to the without upgrade scenario, as shown in Table 5-11. Based on our analysis, the upgrade of the master node is completed by the time the failed worker node reboots. The master node reacts to the failure of the worker node by marking the worker node's state as *NodeNotReady*, and the State controller initiates the service recovery of the failed pod caused due to worker node failure, by performing a failover to the available standby pod, running on another worker node. These measurements are similar for a stateless and stateful application in an HA-master Kubernetes cluster.

Table 5-10: kubeadm: stateless application – worker node failure during master node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time / Service degradation	Service outage
Without upgrade	Single Master	34.401	80.731	1.614	116.746	0
With upgrade		35.512	81.021	1.631	118.153	0
Without upgrade	HA Master	34.401	80.731	1.614	116.746	0
With upgrade		36.867	80.284	1.571	118.722	0

Table 5-11: kubeadm: stateful application – worker node failure during master node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time	Service outage
Without upgrade	Single Master	36.085	82.917	1.591	120.593	36.744
With upgrade		37.151	82.141	1.672	120.964	37.802
Without upgrade	HA Master	36.085	82.917	1.591	120.593	36.744
With upgrade		35.819	81.502	1.493	118.884	36.488

5.2.3.3 Evaluating Master Node Upgrade of kOps Managed Kubernetes Cluster

Impact on Failure Recovery and Service Degradation

For stateless and stateful applications, for a single master cluster, in the event of a worker node failure during the master node upgrade, we observed an overall increase in the failure recovery, as shown in Table 5-12 and Table 5-13. As analyzed, unlike kubeadm, kOps

first delete the selected master node in the current Kubernetes version and then creates a new master node in the desired Kubernetes version. Since, no master node is available for the duration when the old master node is terminated on account of the upgrade, till the newly created master becomes available (has all the master node components in a ready state), thus the worker node failure is not detected. It is only when the newly created master becomes available, it reacts to the worker node failure. So, this additional time taken during the master node upgrade increases the detection time. It is also observed that the repair time is decreased in comparison to the failure without an upgrade scenario. The decrease in repair time is because the worker node has already been rebooted and is ready for repair by the time the master node finishes its upgrade process and detects the failure. So, once the master node becomes available, the worker node's kubelet component communicates with the master node to fetch the list of the pods running on the node and completes the recovery process.

Also, the stateless application experiences service degradation for the duration the failed pod does not provide application service, so failure recovery equals service degradation, as shown in Table 5-12.

Table 5-12: kOps: stateless application – worker node failure during master node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time / Service degradation	Service outage
Without upgrade	Single Master	38.105	72.471	1.598	112.174	0
With upgrade		325.119	58.704	1.640	385.463	0
Without upgrade	HA Master	38.105	72.471	1.598	112.174	0
With upgrade		36.597	71.912	1.607	110.116	0

Impact on Service Outage

In a single master cluster, for stateless application, when one of the worker node hosting application pods is injected with failure during the master upgrade, we do not observe any

service outage. This is because a pod is available on another worker node, which will continue to provide application service.

For stateful application, we observe an increase in the service outage, as shown in Table 5-13. The State controller relies on its interaction with the master node to perform failover. Since the master node is unavailable during its upgrade, the State controller cannot perform failover for that duration. Once the new master node becomes available, it reacts to the failure. At this time, the State controller is informed about the failure, so it performs a failover to the available standby pod. It causes an increased service outage of 319.782 seconds, as highlighted in red in Table 5-13.

Table 5-13: kOps- stateful application – worker node failure during master node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time	Service outage
Without upgrade	Single Master	35.489	80.291	1.635	117.415	36.138
With upgrade		319.087	64.290	1.706	385.083	319.782
Without upgrade	HA Master	35.489	80.291	1.635	117.415	36.138
With upgrade		36.971	79.813	1.669	118.453	37.633

However, for the HA-master Kubernetes cluster in kOps for stateless and stateful application (see Table 5-12 and Table 5-13), no additional impact is observed on the failed unit outage, service degradation, and service outage time in any of the experiments. This is because another master node would be available to perform recovery operations.

5.2.3.4 Evaluating Worker Node Upgrade of Kubeadm Managed Kubernetes Cluster Impact on Failure Recovery and Service Degradation

For both stateless and stateful application, it is observed (see Table 5-14 and Table 5-15) that there is no additional impact on the failed unit outage time due to the worker node upgrade. As the kubelet component remains available on the worker node whose pod is injected

with the failure, it can detect the failure and communicate with the master node to perform the recovery operations. Thus, we do not observe any additional impact.

For stateless application, the application’s service is observed to be degraded for the duration it takes to recover the pod failed due to worker node failure.

Table 5-14: kubeadm: Stateless application – worker node failure during worker node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				Service outage
		Detection time	Repair time	Assignment time	Failed unit outage time / Service degradation	
Without upgrade	Single Master	34.401	80.731	1.614	116.746	0
With upgrade		35.512	79.520	1.682	116.714	0
Without upgrade	HA Master	34.401	80.731	1.614	116.746	0
With upgrade		35.190	79.419	1.596	116.205	0

Impact on Service Outage

For a stateless application, we do not observe any service outage. This happens because, during the worker node step in kubeadm, the pods running on it are not impacted, so at the same time if another worker node bearing the application pod is injected with failure, there is at least one pod available to provide application services. Thus, the application service will be available.

For stateful application, there is also no additional impact on the service outage time (see Table 5-15). As analyzed, the worker node upgrade process, as shown in Figure 5-2, is not disruptive, so during the worker node upgrade step, the standby pod hosted on it keeps running. At this time, when the worker node bearing the active pod is failed, the State controller can perform a failover to the available standby pod. These results are the same for the HA-master Kubernetes cluster.

Table 5-15: kubeadm: Stateful application – worker node failure during worker node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time	Service outage
Without upgrade	Single Master	36.085	82.917	1.591	120.593	36.744
With upgrade		34.572	82.419	1.623	119.623	35.251
Without upgrade	HA Master	36.085	82.917	1.591	120.593	36.744
With upgrade		35.623	79.865	1.558	119.046	36.284

5.2.3.5 Evaluating Worker Node Upgrade of kOps Managed Kubernetes Cluster

Impact on Failure Recovery and Service Degradation

For both stateless and stateful application, we do not observe any additional impact on the failed unit outage time, as the master node is available to detect the worker node failure and perform necessary recovery operations. So, post failure, when the worker node joins the cluster, it completes its recovery actions (mentioned in sub-section 2.5.2) by communicating with the master node. The calculated failed unit outage time is shown in Table 5-16 and Table 5-17 for stateless and stateful applications, respectively. These results are the same for the HA-master Kubernetes cluster.

Also, for stateless application, we observe the service degradation for the duration it takes to recover the pod failed due to worker node failure, as shown in Table 5-16.

Table 5-16: kOps: stateless application – worker node failure during worker node upgrade

Scenarios	Architecture	Metrics (unit: seconds)					
		Detection time	Repair time	Assignment time	Failed unit outage time	Service degradation	Service outage
Without upgrade	Single Master	38.105	72.471	1.598	112.174	112.174	0
With upgrade		37.750	71.998	1.627	111.375	111.375	2.071
Without upgrade	HA Master	38.105	72.471	1.598	112.174	112.174	0
With upgrade		36.168	72.043	1.586	110.247	110.247	2.067

Impact on Service Outage

For stateless application, we observe an increased service outage of 2.071 seconds and 2.067 seconds, for Single and HA master Kubernetes cluster, respectively, as highlighted in red in Table 5-16. As analyzed when we inject failure in one of the worker nodes while another node is being upgraded, there is no pod available to provide the application service. This happens because, while the pod failed on account of worker node failure is recovering, the pod running on another worker node is evicted because of the upgrade, causing a service outage. When the pod evicted on account of the upgrade is started before the failed pod is recovered, the application resumes its service.

For stateful application, we observed an increase in the service outage time of 37.019 seconds and 36.215 seconds for single and HA master Kubernetes clusters, respectively, as highlighted in red in Table 5-17. It happens because when the worker node bearing the active pod is failed while the worker node bearing the standby pod is upgrading, the active pod is failed twice, causing the State controller to perform the failover twice. The first failover is performed when the State controller is notified of the active pod failure through the master nodes, i.e., detection time of 35.701seconds for a single master and 34.904 seconds HA master cluster; it performs failover from the failed active pod to the available standby pod running on the worker node being upgraded. The second failover is performed due to an ongoing upgrade when the currently active pod running on the upgrading node is marked for graceful eviction. As the State controller gets notified about this event, it performs a failover to the current standby pod. Since the scenario caused the active pod to terminate twice, it caused an increased service outage time.

Table 5-17: kOps: stateful application – worker node failure during worker node upgrade

Scenarios	Architecture	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time	Service outage
Without upgrade	Single Master	35.489	80.291	1.635	117.415	36.138
With upgrade		35.701	79.826	1.586	117.113	37.019
Without upgrade	HA Master	35.489	80.291	1.635	117.415	36.138
With upgrade		34.904	80.211	1.681	116.796	36.215

5.2.4 RQ3: Evaluate the Impact of a Failure of the Kubernetes’ Cluster Version Upgrade Process

5.2.4.1 Experiments

The experiment aims to evaluate the impact on the state of the cluster, post upgrade process failure, and the remediation measures taken by the tools to restore the state of the cluster.

In kubeadm, the successful completion of the master node updates the *kubeadm-config* file, which is later referred to by all other nodes in the cluster during their upgrade. So, failing the master node upgrade process would result in the upgrade process failure of kubeadm created Kubernetes cluster. To simulate the failure, we abruptly reboot the master node while it is upgrading using the Linux *reboot* command.

In the kOps created Kubernetes cluster, cluster validation failure results in upgrade process failure of the cluster. As cluster validation happens for each master/worker node upgrade, so to fail the cluster validation, we inject node failure by continuously rebooting the upgrading node using the Linux *reboot* command.

The evaluation of the impact was divided into two parts. First, we identify if the upgrade process failure was detected by the tool managing the Kubernetes cluster. Second, if the tool

detects the failure, we evaluate the remediation measures taken by the tools to recover the cluster post the upgrade process failure.

5.2.4.2 Evaluating the Detection of Upgrade Process Failure by the Tools Managing the Kubernetes Cluster

Kubeadm does not provide any functionality for detecting a failure of the upgrade process, so the detection process is manual. However, in kOps, the Kubernetes cluster version upgrade process involves cluster validation, and if the cluster is not validated within the *cluster-validation-timeout* duration, the upgrade process is marked as failed. So, the time to detect the upgrade process failure is the default *cluster-validation-timeout* duration of 15 minutes.

5.2.4.3 Evaluating the Remediation Measures Taken by the Tools Managing the Kubernetes Cluster to Restore the Cluster's State post Kubernetes' Upgrade Process Failure

Kubeadm does not provide any remediation measures to restore the Kubernetes version. The Kubernetes version must be restored manually.

However, in the kOps created Kubernetes cluster, if the cluster fails to validate within the default duration of 15 minutes, kOps fails the cluster version upgrade process. After the upgrade process failure is identified, the AWS instance group creates a new master/worker node in the updated Kubernetes version. When the upgrade process fails during an ongoing upgrade of a node in a Kubernetes cluster, the instance groups in AWS are responsible for launching a new node by referring to the cluster specification file (that is updated in one of the very first steps of kOps upgrade process flow) as shown in Figure 5-4 and mentioned in subsection 5.1.2. The updated cluster specification becomes the point of reference for the cluster's state (in terms of Kubernetes cluster version) in the event of master/worker node failure. However, the remaining nodes that are not yet upgraded keep running in the old Kubernetes version.

5.2.5 Assessing the Achievable Service Availability, during Kubernetes' Cluster Version Upgrade, and in Presence of Failure during Upgrade (H₁)

In this sub-section, we discuss the results of our evaluation to test our hypothesis H₁. Kubernetes cluster version must be upgraded at-most 56 times in a year, this is deduced from the frequency of releases by Kubernetes community as mentioned in Section 5.1. So, we use the measurements of our evaluation to calculate service availability using Equation 2. The calculations represents the service availability achieved when the Kubernetes cluster version is upgraded (with and without failure) 56 times a year. Furthermore, each service availability calculation (for every evaluated scenario) considers the upgrade of all the nodes in the entire cluster and is specific to our experiment setting and the associated measurements.

5.2.5.1 Service Availability During Upgrades

Kubernetes cluster version upgrade involves upgrading the Kubernetes version of all the master and worker nodes in a cluster. So, to calculate service availability achieved during the Kubernetes cluster version upgrade, we consider the service outage caused during each node upgrade of the entire Kubernetes cluster.

For a kOps-created cluster, the measurement presented in Table 5-1 indicates that the outage is observed only during worker node upgrade, so, we utilized these measurements to calculate the total service outage. In the case of kubeadm created cluster, we do not observe any service outage when either master node (see sub-section 5.2.1.2) or worker node (see sub-section 5.2.1.3) is upgraded. Thus, as presented in Figure 5-5, our calculations conclude that high availability of 99.999% per year is maintained for the deployed application, during Kubernetes cluster version upgrade, for both kubeadm and kOps-created cluster.

5.2.5.2 Service Availability during Upgrade in the Presence of Failure

Next, we evaluated the failure scenarios during upgrade, as presented in sub-section 5.2.2 for pod failure, sub-section 5.2.3 for worker node failure, and sub-section 5.2.4 for upgrade process failure.

For the Stateless application, in the event of pod/worker node failure during Kubernetes cluster version upgrade, no service outage is observed (because of the nature of the application); this holds true for both kubeadm, and kOps created Kubernetes cluster. So, high availability of the application is maintained.

However, for the stateful application hosted on a single-master and two-worker node Kubernetes cluster, pod/worker node failure during the upgrade causes a service outage, thereby impacting the high availability of the application. In case of pod failure during upgrade we use the measurement presented in Table 5-3 and Table 5-7 and calculated that service availability for the stateful application is 99.998% in kubeadm created Kubernetes cluster; and using the measurement presented in Table 5-5 and Table 5-9, a service availability of 99.937% is offered in kOps created Kubernetes cluster. In case of a worker node failure, the service availability is 99.987% in kubeadm created Kubernetes cluster (see Table 5-11 and Table 5-15), and service availability of 99.936% is offered in a kOps-created Kubernetes cluster (see Table 5-13 and Table 5-17). These computations are shown in Figure 5-5 and represent the percentage of service availability guaranteed in a year if the cluster is upgraded 56 times with these outages happening in the cluster.

We also calculated service availability achieved with HA-master cluster, with three master and two-worker nodes. As shown in Figure 5-6 for the HA-master cluster, our calculations indicate that during upgrades in the presence of pod/worker node failure, the offered service availability is improved in a HA-master cluster when compared to single-master cluster.

In the event of pod failure during upgrade in a HA-master cluster (see Table 5-5 and Table 5-9), the service outages do not impact the high availability of the application, this is true for both kOps and kubeadm created cluster. For a kOps managed cluster, in the event of worker node failure during upgrade, the HA-master cluster offers improved service availability of 99.986% when compared to single-master cluster (see Figure 5-5 and Figure 5-6). For a kubeadm managed cluster, since upgrades do not cause additional outage in the event of worker node failure, the service availability offered remains same for both single-master and HA-master cluster, as shown in Figure 5-5 and Figure 5-6.

Finally, we also evaluated the upgrade process failure scenario presented in sub-section 5.2.4 and learnt that the high availability of the deployed application is not impacted.

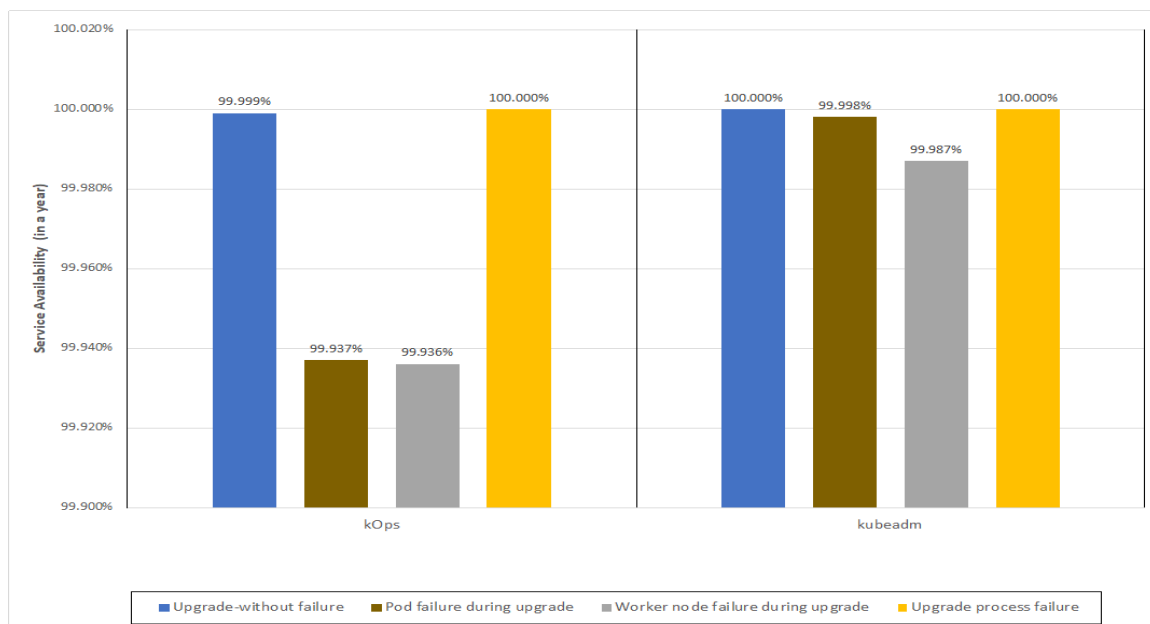


Figure 5-5: Single-master Kubernetes cluster: Service availability achieved when Kubernetes’ cluster version upgrades for a year, managed by kOps, and kubeadm tool

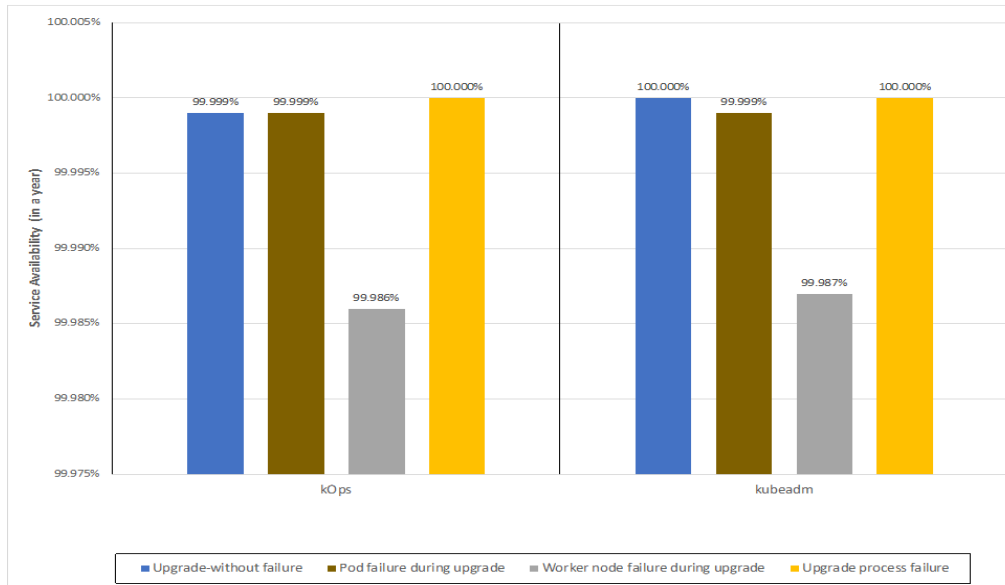


Figure 5-6: HA-master Kubernetes cluster: Service availability achieved when Kubernetes’ cluster version upgrade for a year, managed by kOps, and kubeadm tool

5.2.5.3 Conclusion

As shown in Figure 5-5, our results suggests that the high availability of application deployed in a Kubernetes cluster is maintained during Kubernetes cluster version upgrade in the absence of any failure and during upgrade process failure scenario. This is true for both single and HA-master cluster, managed by kOps and kubeadm. However, in a single master cluster, the high availability is impacted when failures happen during the ongoing upgrades. Furthermore, these impact on service availability can be mitigated in a HA-master Kubernetes cluster. As shown in Figure 5-6, HA-master cluster offers high availability during upgrade in the presence of pod failure for both kubeadm and kOps created cluster, and better service availability during upgrade in the presence of worker node failure for kOps created cluster. Thus, these results of our assessment partially validate our Hypothesis H₁ for the considered cluster setting and the associated measurements.

5.3 Overall Analysis and Potential Improvements

In this section, we provide a summary of the evaluation and issues identified during the Kubernetes cluster version upgrade. Then, we discuss the issues not handled by Kubernetes and provide a potential solution.

Through our evaluations, we learned that since the upgrade process of kOps involves creating a new node with the updated Kubernetes version and terminating the old node in the current version, it causes a drastic impact on the service outage as compared to upgrade during a kubeadm created Kubernetes cluster. Also, kOps can identify the upgrade process failure through cluster validation which is a part of the upgrade process. The upgrade process is marked as failed if the cluster is not validated within the cluster-validation timeout duration. However, for the kubeadm managed cluster, the detection of the upgrade process failure and the restoration measures are manual.

We present potential improvements to solve the problems associated with kubeadm created Kubernetes cluster, where manual intervention is required to detect and restore the Kubernetes version when its cluster's upgrade process fails.

As identified in sub-section 5.2.4.2, when the upgrade process of the master node fails, kubeadm does not detect the Kubernetes upgrade process failure; this causes inconsistency in versions of the master node components. Since a successful upgrade of the master node is essential for upgrades to follow, it also disrupts the upgrade process of the cluster. So, we propose potential improvements to solve the upgrade process failure of the master node. Next, we will explain our analysis associated with the upgrade process failure of the master node. During the master node upgrade process step shown in Figure 5-1, the following steps taken are performed to upgrade the master node components:

1. First, the images of the master node components (existing as static pods) in the desired Kubernetes version are pulled and kept in the *upgrade manifests* folder shown in Figure 5-7.
2. Next, for each master node component, kubeadm performs the following steps to upgrade them:
 - a. First, the current configuration of the master node component is backed up from the *current config* folder to the *backup manifest* folder, as shown in Figure 5-7.
 - b. Then, the new configuration is moved from the *upgrade manifest* folder to the *current config* folder. The Kubelet component on that node monitors this change and restarts the component associated with the new configuration file.
 - c. Once restarted, the component reflects the updated version, and the upgrade process completes for that master node component.
3. Once all the master node components are upgraded, the backup manifest folder will have all the old configuration files, and the upgrade manifest folder will be empty and eventually removed.
4. Finally, the *kubeadm-config* is updated with the Kubernetes version, and later this file will be referred by all other nodes for their upgrade.

So, we observed that when upgrade process failure is injected during the master node component upgrade (in Step 2), the master node components whose config transfer is successfully completed are upgraded to a newer version, while other components remain in the older version. Figure 5-7 shows when the master node upgrade process fails and highlights the state of the *current-config* folder in which master node components exist in different versions.

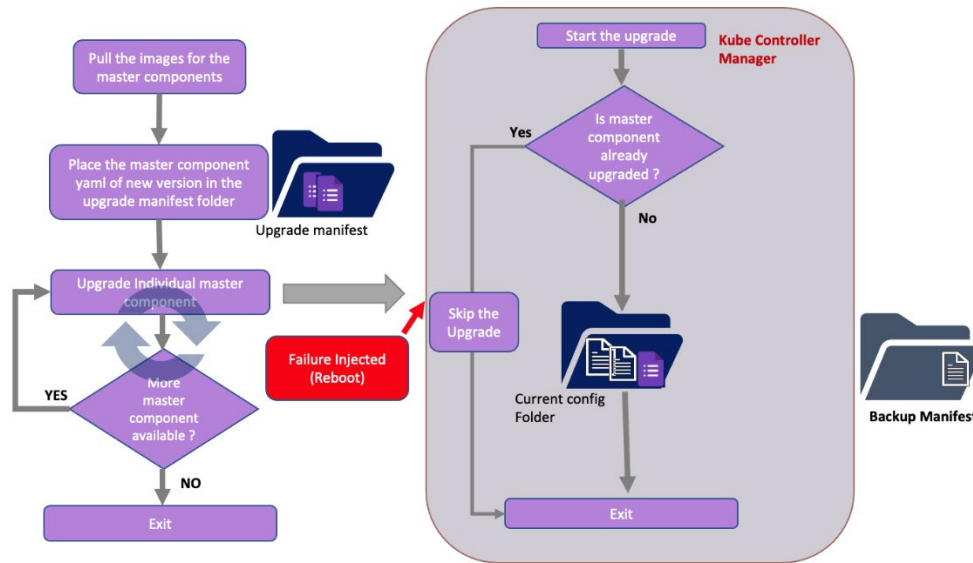


Figure 5-7: Post the master node upgrade process failure, the figure showing the status of the current config folder

To solve the mentioned problems, we propose the following solution that should be employed in an upgrade-manager tool (an external tool that manages Kubernetes cluster version upgrade):

1. The first step is to deploy a flag whose value is set to *false* by default when the upgrade starts and becomes *true* only when all the master node components are successfully updated. The *upgrade manifest* folder can be monitored to see if the upgrade has been successfully completed. That would lead us to the detection of the upgrade process failure. Also, a timeout duration must be specified. Once this timeout duration is exceeded; the upgrade manager tool should investigate the process failure. This timeout should consider various factors, such as the usual master node upgrade time, the status of the node, and the master node components running in it.
2. Next, if the value of the flag is not set to *true*, the upgrade command must be re-initiated. As per the logic of the kubeadm upgrade command, it skips upgrading the components that are already upgraded and only upgrades the components left to upgrade. This step affirms that all the master node components are in the same version. Once all

master node components get successfully upgraded, the *kubeadm-config* file is updated with the new Kubernetes version.

Once the two mentioned steps are completed, all the master node components will be in the same version. So, that is how an upgrade process failure could be automatically detected and handled for a kubeadm created Kubernetes cluster.

5.4 Conclusion

In this chapter, we evaluated the Kubernetes cluster version upgrade managed by kubeadm and the kOps tool available to practitioners. The experiments are performed on a single-master and HA-master Kubernetes cluster to analyze the impact of the Kubernetes cluster version upgrade on failure recovery and application services.

We also assessed the achievable service availability when the Kubernetes cluster version is upgraded (with and without failure) during its upgrade cycle in a year. We learnt that, in a single-master cluster, failure during an upgrade drastically affects the high availability. However, high availability is maintained during the upgrade (in the absence of failure) and when the upgrade process fails. As the service availability calculated and presented is specific to the cluster setting considered in our work, the service availability can vary in a different cluster setting and on the way the cluster gets upgraded. For example, when upgrades happen in large clusters, service availability may vary on various factors: the number of nodes upgraded at once, the distribution and replication of application pods across the worker nodes in the cluster, and so on. Our assessment aims to present an insight into handling large clusters with complex applications by understanding the achievable service availability in a small cluster (where the application pods are distributed evenly, and the cluster version of each node is upgraded one at a time).

Through our experiments, we identified shortcomings of tools managing Kubernetes cluster to recover the cluster's state if its Kubernetes version upgrade process fails. For example, in a kubeadm created Kubernetes cluster, if the master node upgrade process fails, then only master node components upgraded before the failure are in the upgraded version while the remaining components yet to upgrade run in the older Kubernetes version. So, the upgrade process failure causes inconsistency in versions of the master node components, and since a successful upgrade of the master node is essential for upgrades to follow, it disrupts the upgrade process of the cluster. So, we propose potential improvements to automate the restorative actions in the event of upgrade process failure: first, by detecting the failure by analyzing the state of the master node configuration file; and then handling the failure by re-initiating the upgrade that (by default) resumes the upgrade of master node components yet to be upgraded.

We acknowledge that there are some threats to the validity of our results. We considered the default configuration of Kubernetes for our experiments. Changing the default configuration would give different measurements for the calculated outage time. For example, reducing the status update time from the kubelet to the master node would reduce the overall outage time experience in the Worker node failure. However, we aimed to identify the maximum impact of upgrades in the default configurations of Kubernetes to act as a point of reference for the unaltered version of the Kubernetes cluster.

Furthermore, all the experiments were performed in two basic architectures, a single master and two worker nodes Kubernetes cluster, and HA architecture, consisting of three masters and two worker nodes. Kubernetes may behave differently in larger clusters which may impact the measurements presented in our experiments. Also, it is crucial to consider different applications before generalizing a particular criterion. For the experimentation, we considered

a simple stateless NGINX webserver application; the results might vary with the larger micro-service-based application. Still, our experiments indicate the availability and recovery guaranteed by Kubernetes for the application deployed during upgrades.

Chapter 6

Kubernetes Application Upgrade

In this chapter, we quantitatively evaluate the application upgrade strategies as provided by Kubernetes. In Section 6.1, we have explained the application upgrade strategies natively provided by Kubernetes for the applications deployed in its cluster. In Section 6.2, we have described the evaluation criteria; the research questions we aim to answer before discussing the results and analysis of the performed experiments and assessing the achievable service availability during upgrade. In Section 6.3, we provide potential improvements to some of the analyzed issues and conclude our evaluations in Section 6.4.

6.1 Current Practice of Upgrade

Kubernetes provides different strategies to upgrade an application deployed in a Kubernetes cluster. The strategies differ on the nature of the application, i.e., stateless or stateful, and can be defined directly in the respective specification file [27], also referred to as spec.

A stateless application is (usually) deployed in a Kubernetes cluster with the object of kind Deployment, and thus they are managed by the Deployment controller via ReplicaSet [28]. On the other hand, a stateful application is (usually) deployed with the object of kind StatefulSet, and thus they are managed by the StatefulSet controller.

6.1.1 Upgrade Strategies for Stateless Application

A Deployment's upgrade is triggered if the Deployment's Pod template (also called Deployment spec) is changed, for example, if the labels or the container images of the template

are updated. Other updates, such as scaling the Deployment, do not trigger an upgrade [29]. A stateless application managed by the Deployment controller can either be upgraded via *Recreate* or by *RollingUpdate* strategy and can be defined in the `.spec.strategy.type` field of the Deployment spec. The default application upgrade strategy is *RollingUpdate*.

In the Deployment spec, when the value of the `.spec.strategy.type` is set as *Recreate*, and the application version is updated in the spec, the Deployment controller first scales down the old ReplicaSet to zero, which causes all the existing pods running in the old version to be gracefully terminated. Then, it creates a new ReplicaSet for the desired version and scales up this ReplicaSet to the desired number of pods, which causes the creation of a desired number of pods in the new version (as mentioned in the spec). Figure 6-1 illustrates the application upgrade process of two pods from version v1 to version v2 via *Recreate* strategy.

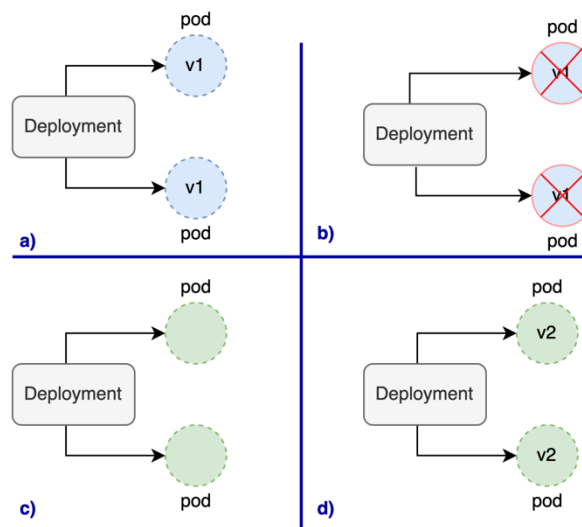


Figure 6-1: Illustration of *Recreate* strategy for stateless application

However, in the Deployment spec, when the value of the `.spec.strategy.type` is set as *RollingUpdate*, and the application version is updated in the spec, the Deployment controller first creates a new ReplicaSet with the updated version and scales up this ReplicaSet by one pod (by default) to create a pod in the updated version. Only once this newly created pod transitions to the *Running* state, it scales down the old ReplicaSet by one pod, deleting a pod in the

old version. This process continues till all the pods are upgraded to the newer version. Figure 6-2 illustrates the application upgrade process of two pods from version v1 to version v2 via the *RollingUpdate* strategy.

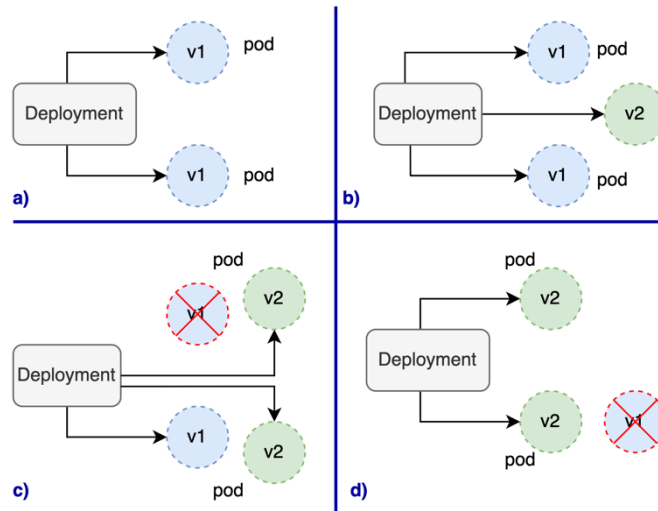


Figure 6-2: Illustration of *RollingUpdate* strategy for stateless application

6.1.2 Upgrade Strategies for Stateful Application

A stateful application managed by the StatefulSet controller can be either upgraded via *OnDelete* or by the *RollingUpdate* strategy [30]. These upgrade strategies can be defined in the *.spec.updateStrategy.type* field of the StatefulSet spec. The default application upgrade strategy is *RollingUpdate*.

In the StatefulSet spec, when the value of the *.spec.updateStrategy.type* is set as *OnDelete*, and the application's image version is updated in the spec, Kubernetes expects the pods in the old version must be manually deleted, and only then StatefulSet controller re-creates the pod with the updated version. Figure 6-3 illustrates the process of application upgrade from version v1 to version v2 using the *OnDelete* strategy, where both the pods in version v1 are deleted at once to initiate their upgrade in version v2.

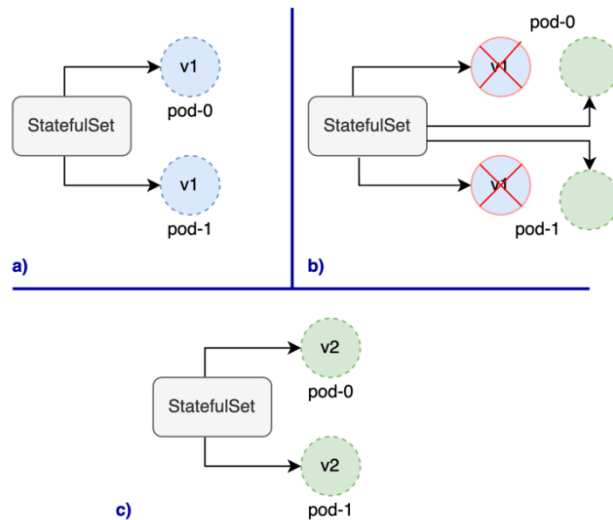


Figure 6-3: Illustration of *OnDelete* upgrade strategy for stateful application

In the StatefulSet spec, when the *spec.updateStrategy.type* is set as *RollingUpdate*, and the application's image version is updated in the spec, the StatefulSet controller starts upgrading the pod in the reverse ordinal number. This indicates that the pod with the highest ordinal number is selected for upgrade and is gracefully terminated; the StatefulSet controller then recreates the pod in the updated version and waits for the pod's transition to the *Running* state before it selects the next pod (if any) for the upgrade. This process continues till all the pods of that spec are updated. Figure 6-4 illustrates the application upgrade process of two pods from version v1 to version v2 using the *RollingUpdate* strategy.

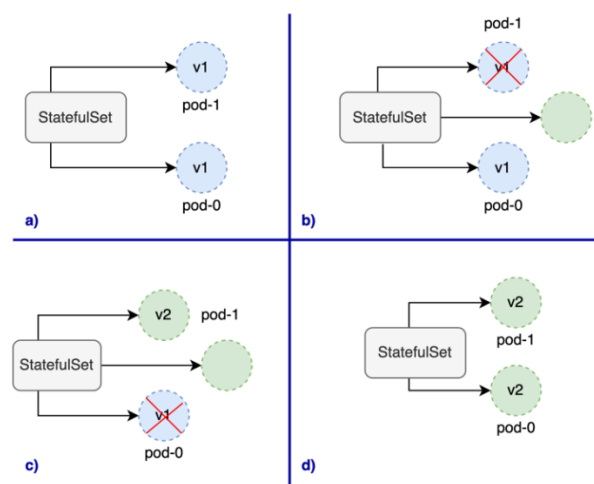


Figure 6-4: Illustration of *RollingUpdate* upgrade strategy for stateful application

6.2 Evaluation of Application Upgrade Strategies

In this section, we evaluate the application upgrade strategies provided by Kubernetes for stateless and stateful applications. Though several other strategies are available to the practitioner to upgrade the application, e.g., Blue-Green release, Canary release, etc., they are not considered for evaluation, as Kubernetes does not provide direct support for them.

Upgrading an application version may impact the service provided by the application. Since Kubernetes provides orchestration of the applications hosted in its cluster, we state the following hypothesis:

H₂: High Availability of application deployed in a Kubernetes cluster is impacted when the application upgrades, especially in the presence of failure.

So, to test H₂, we evaluated the upgrade strategy natively provided by Kubernetes, and defined the following Research Questions (RQ):

RQ1: What is the impact of Kubernetes' application upgrade on the application services?

RQ2-1: What is the impact of Kubernetes' application upgrade on recovery from a pod failure and on application services?

RQ2-2: How does the pod failure during Kubernetes' application upgrade impacts the failed pod's version?

RQ3-1: What is the impact of Kubernetes' application upgrade process failure on the services provided by the application?

RQ3-2: What measures are taken by the Kubernetes' Deployment/StatefulSet controllers to recover from the Kubernetes' application upgrade process failure?

All the experiments are performed on a kubeadm created single master and two worker nodes Kubernetes cluster running version 1.18.1, integrated with Docker version 19.06 as its container runtime. In sub-section 3.2.2, we described the stateless and stateful applications considered for the evaluation. As mentioned in sub-section 3.2.2 and Figure 3-2, the service recovery of the stateful VLC application is managed by the State controller [16]. Also, as observed, the time taken to pull the application image during the upgrade is a variable factor and can vary depending on the size of the image, internet speed etc. So, we avoid this overhead by pulling the application image on the physical machine prior to performing any experiment. The Auto-Metric collector tool automated the process of metrics collection, as described in Chapter 4. The results of all the experiments are in seconds with an accuracy of milliseconds.

6.2.1 RQ1: Evaluate the Impact of Kubernetes' Application Upgrade on the Application Services

In this sub-section, we explain the experiments performed, discuss the results, and present our analysis.

6.2.1.1 Experiments

The experiment aims to evaluate the impact on the services provided by the application that is being upgraded. While the application was upgrading, we monitored the pod-related events in Docker and kubelet logs of the worker nodes. The impact on application service is calculated as a service outage and service degradation.

6.2.1.2 Evaluating the Upgrade Strategies of Stateless Application

As mentioned in sub-section 6.1.1, Kubernetes provides two upgrade strategies for the stateless application managed by the Deployment controller: *Recreate* and *RollingUpdate*; what follows is the evaluation and analysis of these upgrade strategies.

Impact during *Recreate* Upgrade Strategy

The stateless NGINX application experiences a service outage of 11.229 seconds. As mentioned in the upgrade process flow, during *Recreate* strategy (see sub-section 6.1.1 and Figure 6-1), Kubernetes first deletes all the pods running in the old version at once by scaling down the number of replicas in the associated ReplicaSet to zero. Once the old ReplicaSet is deleted, the deployment controller creates a new ReplicaSet in the updated version and scales up this ReplicaSet to the desired number of pods, bringing up all the pods in the updated version at once. Since no pods are available to actively provide application service during the ongoing upgrade, the application experiences a service outage, and the measurements are shown in Table 6-1.

As the application experiences complete downtime for the duration of its upgrade, service degradation is not applicable in this scenario.

Impact during *RollingUpdate* Upgrade Strategy

The stateless NGINX application does not experience any service outage during its upgrade because of the upgrade flow of the *RollingUpdate* strategy (as mentioned in sub-section 6.1.1 and shown in Figure 6-2), where a new pod is created before deleting an old pod. Thus at least one pod is always available during the ongoing upgrade to provide application service.

As mentioned, in the *RollingUpdate* strategy, a new pod is created with the updated version, and once its state transitions to the Running state, the pod in the old version is gracefully terminated, so the desired number of pod replicas is always maintained. Thus, the application service is not degraded during an upgrade.

Table 6-1: Stateless application: Impact on the application services during its upgrade

Metrics (unit: seconds) / Upgrade strategy	Service Outage	Service Degradation
Recreate	11.229	N/A
RollingUpdate	0	0

6.2.1.3 Evaluating the Upgrade Strategies of Stateful Application

As mentioned in sub-section 6.1.2, Kubernetes provides two upgrade strategies for the stateful application: *OnDelete* and *RollingUpdate*. Following are the evaluation and analysis of these upgrade strategies.

Impact during *OnDelete* Upgrade Strategy

Since this strategy involves manual deletion of the pod for them to get upgraded, we performed deletion in the following three combinations to understand the maximum impact on the service outage of the stateful VLC application:

- 1. When the Upgrade Started with the Standby Pod:** In this scenario, we upgrade one pod at a time, beginning with the upgrade of a standby pod and then upgrading the active pod.

Service Outage: The stateful VLC application experiences a total service outage of 0.643 seconds because of the upgrade process of the *OnDelete* upgrade strategy, which involves the manual deletion of the pod that should get upgraded. So, when the standby pod is deleted for an upgrade, the application’s service is not impacted because the active pod remains available to stream the video. Once the standby pod gets recreated post upgrade, we delete the active pod for an upgrade. On deletion of the active pod, the stateful VLC application stops streaming the video, and the application experiences a service outage. When kubelet reports the deletion of the pod, the State controller performs the failover to the available standby pod. The time it takes for the State controller to perform the service recovery constituted a service outage of 0.643 seconds, as shown in Table 6-2.

- 2. When the Upgrade Started with the Active Pod:** In this scenario, we upgrade one pod at a time, beginning with the upgrade of an active pod and then upgrading the standby pod.

Service Outage: As the upgrade process of OnDelete involves manual deletion of the pod that should be upgraded, the stateful VLC application experiences a service outage of 1.287 seconds. So, when the active pod is deleted for an upgrade, the stateful VLC application stops streaming the video, and the application experiences a service outage. When kubelet reports the deletion of the pod, the State controller performs a failover to the available standby pod. The time it takes for the State controller to perform the service recovery constituted a service outage of 0.647 seconds. After the pod gets recreated, it is assigned a standby label. Now, when the next pod is deleted for an upgrade, which is the current active pod, so the application again experiences a service outage. Kubelet reports the deletion of the pod, and the State controller performs the failover to the available standby pod. The time taken by the State controller to perform failover constitutes the second service outage of 0.640 seconds. Since the failover happens twice in this scenario, it causes a total service outage of 1.287 seconds, as shown in Table 6-2.

- 3. When both the Pods are Upgraded at Once:** In this scenario, we initiate the upgrade of both pods simultaneously by manually deleting them.

Service Outage: The application experiences complete downtime and observes a service outage of 7.132 seconds, as shown in Table 6-2. As both the pods are deleted for their upgrade, no pods will be available to provide application service. The observed service outage is the time taken by the StatefulSet controller to create a pair of new pods and for the State controller to assign active and standby roles to these pods. Once these roles are assigned, the active pod starts the video streaming service.

Impact during *RollingUpdate* Upgrade Strategy

Since the impact of application services depends on the type of pod (active/standby) chosen as the starting point of upgrade, so, we considered the following two scenarios for evaluation:

- 1. When the Upgrade Started with the Standby Pod:** In this scenario, the standby pod is at the highest ordinal number. So, it gets upgraded first, and then the active pod is selected for upgrade.

Service Outage: The application experiences a service outage of 0.641 seconds, as presented in Table 6-2. As per the upgrade process of *RollingUpdate* mentioned in sub-section 6.1.2, the StatefulSet controller first terminates the pod and then re-creates it in the updated VLC image version. So, when the standby pod gets terminated, as part of the upgrade process, the application's service is not impacted because the active pod is available to stream the video. But once the selected standby pod gets upgraded, the next pod selected for the upgrade is the active pod. Now, when the active pod is removed for an upgrade, it causes a service outage. The State controller detects the termination of the active pod from the kubelet events and initiates the failover to the available standby pod. The resulting service outage is the time the State controller takes to perform the failover to the standby pod.

- 2. When the Upgrade Starts with the Active Pod:** In this scenario, the active pod is at the highest ordinal number. So, it gets upgraded first, and then the standby pod is selected for upgrade.

Service Outage: The stateful VLC application experiences a service outage of 1.293 seconds, as shown in Table 6-2. When the upgrade starts with the active pod having the highest ordinal number, the active pod is removed to be recreated with the updated VLC image version, and this causes the application to experience a service outage. The State controller

detects the termination of the active pod from the kubelet events and performs the failover to the available standby pod. The time taken by the State controller to perform the service recovery constitutes a service outage of 0.641 seconds. Once the pod selected for the upgrade is upgraded successfully, the StatefulSet controller picks the next pod (current active pod) for an upgrade. When the current active pod is removed for an upgrade, the application again experiences a service outage. When the State controller detects this, it again initiates the failover to the available standby pod. The time taken by the State controller to perform service recovery constitutes a service outage of 0.652 seconds. Since the failover happens twice, the application experiences a total service outage of 1.293 seconds.

Table 6-2: Stateful application: Impact on the application services during its upgrade

Upgrade strategy	Scenarios		Metrics (unit: seconds)
			Service outage
OnDelete	Upgrading one pod at a time	Upgrade starts with standby pod	0.643
		Upgrade starts with active pod	1.287
	Both pods upgraded at once		7.132
RollingUpdate	Upgrade started with standby pod		0.641
	Upgrade started with active pod		1.293

6.2.2 RQ2-1: Evaluate the Impact of Kubernetes' Application Upgrade on the Recovery from a Pod Failure and on Application Services

In this sub-section, we explain the different experiments performed, then we discuss the results of these experiments and present our analysis on the impact of application upgrade on the recovery from pod failure and on application services.

6.2.2.1 Experiment

The experiment aims to evaluate the impact of application upgrades in the presence of failure. This impact on failure recovery is measured in terms of failed unit outage time, and the

impact on the application service is calculated as service outage and service degradation. We simulated pod failure by killing the pod process from the OS.

To better compare and understand the upgrade's impact, we injected the pod failure without the application upgrade, while performing the application upgrade. When pod failure was injected in the absence of an upgrade, it gave a clear picture of the time it usually takes for the failed unit to be recovered and the duration of any service outage and service degradation experienced by the application. Then, we injected pod failure during the application upgrade to evaluate the impact of the upgrade on the failure recovery and application services.

We follow the same experiments for stateful application, just that the pod failure was injected in the active pod, i.e., the pod that was actively providing the service, to evaluate the maximum impact on the application services.

6.2.2.2 Evaluating the Upgrade Strategies of Stateless Application

Impact during *Recreate* Upgrade Strategy

Impact on Service Outage: The stateless NGINX application experiences a service outage of 11.263 seconds, so there is no additional impact on the service outage compared to the without failure scenario in RQ1. Since in *Recreate* strategy, all the pods running in the old version are deleted before they are re-created in the updated version, so, when a pod (running in the old version) is failed at the same time, the failed pod is not re-created (and not recovered from failure) by the Deployment controller, and the upgrade process continues to create new pods with the updated version. The results of the measurements are shown in Table 6-3.

Impact on Failure Recovery and Service Degradation: As the application experiences complete downtime for the duration of its upgrade, service degradation is not applicable in this scenario. Also, since the failed pod is not recovered from failure by the Deployment

controller and the upgrade process continues to create the pod in an updated version, the impact on failure recovery is not applicable in this scenario.

Impact during *RollingUpdate* Upgrade Strategy

Impact on Service Outage: As the *RollingUpdate* strategy creates a pod before terminating a pod in the older version, at least one pod is always available to provide the application service for the stateless NGINX application. So, the application does not experience any service outage.

Impact on Failure Recovery and Service Degradation: There is no additional impact on failure recovery (measured as the failed unit outage time); this happens because the Deployment controller detects the pod failure through kubelet events and re-creates the failed pod before upgrading another pod. This way, failure is given priority over upgrades to follow. Also, the time it takes for the failed pod to recover equals the time in which the actual number of pods was less than the desired number of pods. So, the observed service degradation equals the failed unit outage time of 2.841 seconds, as shown in Table 6-3.

Table 6-3: Stateless application pod failure during application upgrade

Upgrade Strategy	Scenarios	Metrics (unit: seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time /Service degradation	Service outage
	Without upgrade	0.849	0.403	1.536	2.788	0
Recreate	During upgrade	N/A	N/A	N/A	N/A	11.263
RollingUpdate	During upgrade	0.903	0.417	1.521	2.841	0

6.2.2.3 *Evaluating the Upgrade Strategies for Stateful Application*

Impact during *OnDelete* Upgrade Strategy

We considered the following scenario to evaluate the maximum impact on the application services. The pod failure is injected in the active pod, i.e., the pod that was actively providing the service, while the standby pod is manually deleted to initiate its upgrade.

Impact on Service Outage: When the standby pod is manually deleted to initiate its upgrade, at the same time the active pod is injected with failure, the application experiences an increased service outage of 4.191 seconds, as highlighted in red in Table 6-4. This increased service outage is because there are no pods to provide application service at that time. Next, as the standby pod finishes its upgrade before the failed active pod is recovered, the State controller makes the available standby pod as active, and the VLC application begins streaming the video.

Impact on Failure Recovery: We observe no additional impact on the failed unit outage time, as presented in Table 6-4. The deletion of the standby pod and the failure of the active pod simultaneously initiate; the failure recovery and upgrade operation by the StatefulSet controller; and the service recovery operation by the State controller. Since the StatefulSet controller is responsible for maintaining the desired replicas, it initiates the re-creation of the standby pod with the updated VLC image version. Once it is informed through the kubelet events about the active pod failure, it parallelly initiates the recovery of the failed pod. As the failed pod gets repaired parallelly while the upgrade process continues, there is no impact on the recovery of the failed pod.

Impact during *RollingUpdate* upgrade strategy

We considered the following scenario to evaluate the maximum impact on the application services. In our experiment setting, we have a standby pod at the highest ordinal number,

so it is selected first for the upgrade by the StatefulSet controller that issues termination of the standby pod to upgrade, and we inject the active pod, i.e., the pod that was actively providing the service, with failure.

Impact on Service Outage: When the standby pod is selected for an upgrade and consequently terminated, while the active pod is injected with failure, the application experiences an increased service outage of 4.147 seconds, as highlighted in red in Table 6-4. This increased service outage is because there are no pods to provide application service. Next, as the standby pod finishes its upgrade before the failed active pod recovers, the State controller makes the available standby pod as active, and the VLC application begins streaming the video.

Impact on Failure Recovery: While the StatefulSet controller issues termination of the standby pod to upgrade, we inject the active pod with failure, and we observed no additional impact on the failed unit outage time, as presented in Table 6-4. The termination of the standby pod for the upgrade, and the failure of the active pod, simultaneously initiate; the failure recovery and upgrade operation by the StatefulSet controller; service recovery operation by the state controller. Since the StatefulSet controller is responsible for maintaining the desired replicas, it initiates the re-creation of the standby pod with the updated version. And once it is notified through the kubelet events about the active pod failure, it parallelly starts the recovery of the failed pod. As in this strategy, the failure is given priority over the subsequent upgrades to follow, so there is no additional impact on the failed unit outage time.

Table 6-4: Stateful application pod failure during application upgrade

Upgrade Strategy	Scenarios	Metrics (in seconds)				
		Detection time	Repair time	Assignment time	Failed unit outage time	Service outage
	Without upgrade	0.943	2.583	1.593	5.119	1.612
OnDelete	Active pod is failed and standby pod is upgraded	1.000	2.618	1.580	5.198	4.191
RollingUpdate	Active pod is failed and upgrade starts with standby pod	0.921	2.594	1.651	5.166	4.147

6.2.3 RQ2-2: Evaluate the Impact on the Pod (Application) Version post Recovery from Failure

6.2.3.1 Experiment

In this evaluation, we present our observation as an extension of the experiment performed in RQ2-1. So, in RQ2-1, when pod failure was injected during the application upgrade, we observed the application version that the pod attained when it recovered from the failure. The aim is to identify Kubernetes' recovery actions in case of failures during upgrades and to understand whether failures lead to unintentional upgrades.

6.2.3.2 Evaluating the Upgrade Strategies for Stateless Application

For stateless applications, during an ongoing application upgrade using the *Recreate* strategy, when one of the pods (in the old version) is injected with failure; the failed pod is not re-created (and recovered), and the upgrade process continues to create pods in the updated version.

During an ongoing application upgrade using the *RollingUpdate* strategy, when a pod in the old version is injected with failure while another pod is upgrading, we observed that once the failed pod recovers, it retains its old version when restarted.

6.2.3.3 Evaluating the Upgrade Strategies of Stateful Application

For stateful application, during an ongoing application upgrade, irrespective of the upgrade strategy, i.e., *OnDelete* or *RollingUpdate*, when the pod in the old version is injected with failure, the failed pod retains the old version when it recovers.

6.2.4 RQ3-1: Evaluate the Impact of the Kubernetes' Application Upgrade Process Failure on the Availability of the Application

In this sub-section, we explain the different experiments performed, discuss the results of these experiments, and present our analysis.

6.2.4.1 Experiment

The experiment aimed to evaluate the impact of the Kubernetes application upgrade process failure on the application services.

We simulated application upgrade process failure by specifying a non-existent application image version in the *image* field of the spec file. Based on the defined upgrade strategy, the respective controllers detect the change in the *image* field in the spec file and initiate the upgrade. But since the specified application's image did not exist, the upgrade did not progress, and we considered that as an upgrade process failure.

Impact during *Recreate* Upgrade Strategy

For the stateless application, the application experiences complete downtime because of the upgrade process flow of *Recreate* strategy. As analyzed, the upgrade starts by deleting all the pods in the older version; the Deployment controller then tries to create those pods with the newer version; however, as the image does not exist in the application's repository, creation of a new pod is not possible. Since the pods keep trying to pull the image and get stuck in *ErrImagePull* or *ImagePullBackOff* state, none of the application pods is available to provide service.

Impact during *RollingUpdate* Upgrade Strategy

In the case of the *RollingUpdate* strategy, the application service is not impacted. As analysed, in this strategy, the Deployment controller first creates a pod with the mentioned image version and only once the new pod transitions to the *Running* state it terminates the old

pod. However, in this scenario the newly created pod never goes into the Running state because of the non-existent image version; the Deployment controller does not terminate the existing pods running in the older version. So, the desired number of application replicas is maintained, and the application's service is not impacted.

Table 6-5: Stateless Application: Impact on application availability due to upgrade process failure

Scenario	Upgrade Strategy	Impact on application availability
Upgrade strategy failure	Recreate	Complete downtime
	RollingUpdate	No impact

6.2.4.2 Evaluating the Upgrade Strategies of Stateful application

Impact during *OnDelete* Upgrade Strategy

For stateful application, if the strategy type specified is *OnDelete*, and the *image* field of the StatefulSet spec is updated with a non-existent VLC application image version, the pod that has been manually deleted for an upgrade is affected, while other existing pods keep running. Since this strategy involves manual deletion of the pod for them to get upgraded, we have performed upgrades in the following three combinations to understand the maximum impact on the service outage of the stateful VLC application:

1. **When the Standby Pod is Upgraded:** As the active pod of the application is still available, the application's service is not impacted. However, the application becomes failure intolerant as there would be no standby pod available for the state controller to perform the failover in case of further failure.
2. **When the Active Pod is Upgraded:** The application experiences a service outage of 0.647 seconds, as presented in Table 6-6; as the active pod is deleted for its upgrade, the State controller detects it via the deleted event reported by kubelet and performs a failover to the available standby pod. The time taken by the State controller to perform the service recovery by making the standby pod as active, constitutes a service outage of 0.647 seconds.

Also, the StatefulSet controller is responsible for maintaining the desired replica of the pod and tries to create a new pod using the image version mentioned in the spec. But, as the image version is non-existent, the newly created pod does not transition to *Running* State, the application becomes failure intolerant.

3. **When all the Pods are Upgraded at Once:** The application experiences complete application downtime, as presented in Table 6-6. As the image mentioned in the spec is non-existent, when the StatefulSet tries to create pods, they do not transition to the *Running* state but instead change to *ErrImagePull* or *ImagePullBackOff* state. Also, as none of the application pods are available to provide application service, the application experiences complete downtime.

Evaluating the Impact of *RollingUpdate* Upgrade Strategy

When the strategy type is set as *RollingUpdate*, and the image field of the StatefulSet spec is updated with a non-existent VLC application image version, the experiments are performed in the following two ways to evaluate the impact on the application's availability:

1. **When the Upgrade Started with the Standby Pod:** In this scenario, the application's availability is maintained since the active pod is available to provide application service. As analyzed, the StatefulSet controller terminates the standby pod (as it is at highest ordinal) and initiates its re-creation in the updated version. But, as the *image* specified in the spec is non-existent, the pod re-creation fails. Since the newly created pod does not transition to the *Running* state but instead transitions to *ErrImagePull* or *ImagePullBackOff* state, the upgrade process does not progress further (to select the next pod, i.e., active pod to upgrade). The application becomes failure intolerant as there would be no standby pod available to perform failover in case of further failure.

2. **When the Upgrade Started with the Active Pod:** In this scenario, the application experiences a service outage of 0.639 seconds, as presented in Table 6-6. When the pod with the highest ordinal number is an active pod, it is selected first for the upgrade. The StatefulSet controller terminates the selected pod and initiates its re-creation in the updated version. As the active pod gets terminated, the VLC application stops streaming the video and the application experiences a service outage. When the State controller detects the termination of an active pod from the kubelet events, it initiates the failover to the available standby pod. The time taken by the State controller to complete the failover by making the standby pod as active, constitutes a service outage. Since the image mentioned in the spec is non-existent, the newly created pod does not transition to the *Running* state but instead transitions to *ErrImagePull* or *ImagePullBackOff* state; the upgrade process does not continue further. The application becomes failure intolerant as there would be no standby pod available to perform failover in case of further failure.

Table 6-6: Stateful Application: Impact on application availability due to upgrade process failure

Upgrade strategy	Scenarios: Upgrade process failure		Impact on application availability
OnDelete	Upgrading one pod at a time	Upgrade starts with standby pod	No outage (Failure intolerant)
		Upgrade starts with active pod	Service outage of 0.647 seconds (Failure intolerant)
	Both pods upgraded at once		Complete downtime
RollingUpdate	Upgrade started with standby pod		No outage (Failure intolerant)
	Upgrade started with active pod		Service outage of 0.639 seconds (Failure intolerant)

6.2.5 RQ3-2: Evaluate the Remediation Measures Taken by the Respective Controller (Deployment and StatefulSet Controller) when Kubernetes' Application Upgrade Process Fails

In this sub-section, we explain the different experiments performed, discuss the results of these experiments, and present our analysis.

6.2.5.1 Experiment

In this evaluation, we present our observation as an extension of the experiment performed in RQ3-1. So, in RQ3-1, when the application upgrade process failed, we observed if the Deployment / StatefulSet controller identified this failure and if they took any measures to restore the application's state. The aim was to evaluate the restorative measures offered by the respective controllers when Kubernetes' application upgrade process fails.

6.2.5.2 Evaluating the Upgrade Strategies of Stateless Application

For stateless applications managed by the Deployment controller, *progressDeadlineSeconds* is the field (in the deployment spec) that specifies the time the deployment controller should wait before marking the deployment as failed. The default value of this field is 600 seconds, but it can be configured. When the stateless application is injected with application upgrade process failure, after the *progressDeadlineSeconds* is exceeded, the deployment controller marks the deployment's progressing status as false, with the reason "*progressDeadlineSeconds exceeded*". At this time, when we check the progressing status of the deployment with the command `kubectl rollout status deployment <deployment-name>`, it displays an error stating "*error: <deployment-name> exceeded its progress deadline*", as shown in Figure 6-5. So, apart from reporting the progressing status as an error, there is no other action taken by the Deployment controller to stop the pod creation process or to trigger the rollback to a stable application version. The application's status must be manually restored to a stable version.

```
root@kworker1:~# kubectl rollout status deployment myapp-deployment
Waiting for deployment "myapp-deployment" rollout to finish: 1 out of 2 new replicas have been updated...
error: deployment "myapp-deployment" exceeded its progress deadline
```

Figure 6-5: Snippet of Progressing status of "myapp-deployment" Deployment when progressing status fails

6.2.5.3 Evaluating the Upgrade Strategies of Stateful Application

For stateful application managed by the StatefulSet controller, unlike the Deployment controller, there are no measures taken by the StatefulSet controller to mark the progressing

status of the upgrade as failed. So, when the application upgrade process fails, the pod(s) continuously tries to pull the image with the specified non-existent image version. The StatefulSet controller does not trigger automatic rollback to bring the application to a stable version, and manually the application status must be restored.

Furthermore, during the manual restoration process, even when the StatefulSet spec is updated with the valid image version, the stateful pod that is in process of creation as part of upgrade remains stuck in the *ErrImagePull* or *ImagePullBackOff* condition and must be manually deleted. Once deleted, the upgrade process is re-initiated for these pods, and they are re-created with the correct application version updated in the spec and transitions to the *Running* state.

6.2.6 Assessing the Achievable Service Availability during Kubernetes' Application Upgrade, and in Presence of Failure during Upgrade (H₂)

In this sub-section, we discuss the results of our evaluation to test our hypothesis H₂. The stateless application considered for our experiment is NGINX webserver application, its upgrade cycle is approximately once per month in a year [31] [32]. This indicates that a NGINX webserver application must be upgraded at most 12 times a year. The stateful application considered for our experiment is VLC video streaming application, and as per the information in [33] [34], VLC releases between 1 to 8 upgrades each year, indicating VLC application must be upgraded at most 8 times a year.

We use the measurements of our evaluation to calculate service availability using Equation 2. The calculations represent the service availability achieved when the stateless NGINX webserver application is upgraded for 12 times a year, and the stateful VLC video streaming application is upgraded 8 times a year, in a Kubernetes cluster. Furthermore, each service availability calculation (for every evaluated scenario) considers the upgrade of all instances of an

application deployed in a Kubernetes cluster. Also, these calculations are specific to the experiment settings considered in our work and its associated measurements.

6.2.6.1 Service Availability during Upgrades

We utilized the measurements of our evaluations as presented in Table 6-1 and Table 6-2, to calculate service availability for stateless and stateful application respectively for their associated upgrade strategies. As shown in Figure 6-6, our calculations indicates that high availability of the stateless and stateful application is maintained when they are upgraded for a year.

6.2.6.2 Service Availability during Upgrades in the Presence of Failure

Next, we evaluated the failure scenarios during application upgrade, as presented in sub-section 6.2.2 for pod failure, and sub-section 6.2.4 for upgrade process failure.

For the stateless application, during application upgrade in the presence of pod failure, we utilize the measurements of our evaluation in Table 6-3, to calculate service availability. For the stateful application, we utilize the measurements of our evaluation in Table 6-4 to calculate service availability. As shown in Figure 6-6, our calculations indicate that outages caused during stateless/stateful application upgrade in the presence of pod failure, do not impact the high availability of the deployed stateless/stateful applications in Kubernetes cluster.

In the event of upgrade process failure, we utilize the measurements presented in Table 6-5 and Table 6-6 for the stateless and stateful application respectively to calculate service availability in this scenario. For stateless application, when the application fails to upgrade with upgrade strategy selected as Recreate, it causes complete downtime for the application until system administrator manually recovers the application. Since application experiences complete downtime until it is manually handled, calculating service availability is not possible

in this case. However, when stateless application fails to upgrade with upgrade strategy selected as RollingUpdate, the calculations shown in Figure 6-6. indicate that high availability of the application is maintained. For stateful application, when the application upgrade process fails with the strategy type selected as RollingUpdate, the high availability of the application is maintained. However, when stateful application fails to upgrade with upgrade strategy used as OnDelete, it causes complete downtime of the application. Since application experiences complete downtime until it is manually handled by system administrator, calculating service availability is not possible in this case.

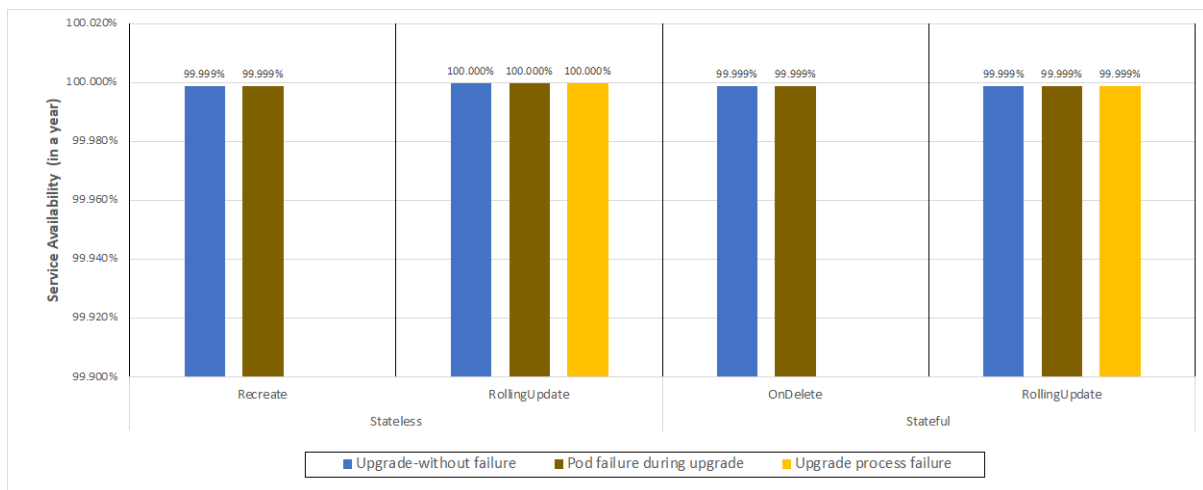


Figure 6-6: Service availability achieved when application deployed in a Kubernetes cluster upgrades for a year (with and without failure)

6.2.6.3 Conclusion

As shown in Figure 6-6, our computations for service availability indicate that the high availability of applications deployed in a Kubernetes cluster is impacted only in case of upgrade process failure. Thus, these results of our evaluation partially validate our Hypothesis H₂ for the considered cluster setting and associated measurements.

6.3 Overall Analysis and Potential Improvements

In this section, we summarize the evaluation and issues identified during the Kubernetes application upgrade. Then, we discuss the issues not handled by Kubernetes and provide a potential solution.

Through our evaluations, we learned that the *Recreate* upgrade strategy leads to a complete service outage for stateless applications. With the *OnDelete* upgrade strategy, we can control the number of pods that can be unavailable during an upgrade, and this is a significant measure of maintaining high availability for stateful applications. We also discovered that abrupt failure does not lead to unintentional upgrades, i.e., post pod failure; when restarted, the pod retains its application version. This applies to all upgrade strategies provided by Kubernetes except *Recreate* strategy. For example, during the *RollingUpdate*, the failed pod is brought back to the version it was in when it failed. However, this is not true for the *Recreate* strategy for stateless application, where the failed pod is not recovered.

Although in the performed experiments in some of the scenarios, we observed an increased service outage due to the ongoing upgrades, Kubernetes eventually handled them. In case of application upgrade process failure discussed in sub-section 6.2.5.2 and sub-section 6.2.5.3, manual intervention was required to restore the application's state, and Kubernetes controllers do not take any remediation measures, so we propose potential improvements to solve the problems that Kubernetes do not handle.

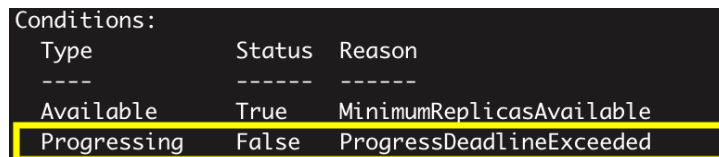
6.3.1 Stateless Application

As mentioned in the results for stateless application, if the application upgrade process fails due to a non-existent pod image, post the `.spec.progressDeadlineSeconds` [35] has exceeded, the Deployment controller marks the deployment's progressing status as false, with the

reason *progressDeadlineSeconds* exceeded. An example of a deployment with the name “*my-app-deployment*” whose *progressDeadlineSeconds* is exceeded is shown in Figure 6-5.

The main issue is that the Deployment controller takes no measures to stop the process of pod creation. The pod keeps trying to pull the image with the specified non-existent image version. Also, the Deployment controller does not trigger automatic rollback to a stable application version, and the application’s version must be manually restored.

The first step of the solution is to detect the Kubernetes’ application process failure by making use of the progressing status updated by the Deployment controller when *progressDeadlineSeconds* is exceeded. Once the *progressDeadlineSeconds* duration has been exceeded, the upgrade manager tool can monitor the Deployment status for the condition shown in Figure 6-7. This Deployment’s condition can be seen in the *Conditions* field of the respective Deployment by using the command *kubectl describe deployment <deployment-name>*.



Type	Status	Reason
Available	True	MinimumReplicasAvailable
Progressing	False	ProgressDeadlineExceeded

Figure 6-7 : Snippet showing the value of the Conditions field of the Deployment

Next, if the deployment fails with the *ProgressDeadlineExceeded* condition, as a restorative measure, the upgrade manager tool should take the following recovery action (via rollback). The rollback to a stable revision (decided by the admin) is done using the following steps:

1. Find the revision to rollback using the following command

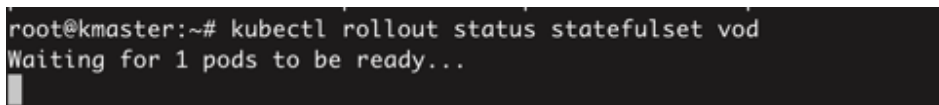
```
kubectl rollout history deployment/<name-of-deployment>
```

2. Rollback to the stable revision using the following command

```
kubectl rollout undo deployment/<name-of-deployment> --to-revision=<revision number>
```

6.3.2 Stateful Application

Unlike a stateless application, for stateful application, the StatefulSet does not have a field like *progressDeadlineSeconds* in its spec. An example of the output of the rollout status of the StatefulSet with the name “vod” is shown as a snippet in Figure 6-8.



```
root@kmaster:~# kubectl rollout status statefulset vod
Waiting for 1 pods to be ready...
```

Figure 6-8: Snippet showing progressing status of StatefulSet “vod” stuck during the upgrade

It gets stuck in this condition forever. The StatefulSet controller takes no measure; to stop the upgrade process or mark the progressing status as failed. The pod continuously tries pulling the image with the specified non-existent image version.

As a solution, the upgrade manager tool can monitor the progressing status of the rollout by having a user-configurable timeout so that it is not stuck watching forever. Along with the progressing status, this timeout should consider conditions like *ImagePullBackOff*, *ErrImagePull*, or other Pod/Replica-specific errors. When the timeout duration is exceeded along with those conditions/errors, it should rollback to the previous version using the following rollback steps.

1. Find the revision to rollback using the following command

```
kubectl rollout history statefulset/<name-of-statefulset>
```

2. Rollback to the desired revision using the following command

```
kubectl rollout undo statefulset/<name-of-statefulset> --to-revision=<revision number>
```

6.4 Conclusion

In this chapter, we presented and evaluated different application upgrade strategies provided by Kubernetes for stateless and stateful applications. Then, we analyzed the results of our evaluation for the impact of the application upgrade process on service availability and failure recovery and discovered shortcomings of Kubernetes to detect and recover from the application upgrade process failure.

We assessed the achievable service availability when an application deployed in a Kubernetes cluster is upgraded (with and without failure) during its upgrade cycle in a year. Through our calculation, we learnt that high availability is impacted when the application's upgrade process fails. Also, upgrade process failure for stateless applications with upgrade strategy as Recreate would always cause complete application downtime in any cluster setting. Furthermore, the service availability calculations presented are specific to the applications considered in our work and may vary depending on factors such as the number of application pods that are upgraded at once (if the upgrade process is manually controlled), application characteristics: such as the failover mechanism, application size, and so on. Our assessment aims to give the cluster administrator an overview and intensity of the impact on service availability for a small cluster running simple applications to allow for better governance of complex applications with multiple replications.

Through our experiments, we learnt that during upgrade process failure, although the Deployment controller detects the failure while the StatefulSet controller does not, none of them takes any restorative actions; as a result, the pod keeps trying to pull the image with the specified non-existent image version, thereby impacting application services. Also, the respective controller does not trigger automatic rollback to a stable application version; it requires

manual intervention to restore the application to a stable version. So, we provide potential improvements in such a scenario. Our solution would identify the application upgrade failure and trigger automatic rollback to the stable application version, to maintain the application services.

We acknowledge that there are some threats to the validity of our results. For example, all experiments are conducted in a small cluster consisting of only a master and two worker nodes. Kubernetes may behave differently in larger clusters which may impact the measurements presented in our experiments. Also, the availability measurements may vary depending on the application's complexity and the collocated applications managed by Kubernetes. We understand that these factors may impact the results of our study. The mapping of the metrics to the concrete events is the biggest threat and requires more investigation as one can map them differently, in which case all the measurements could be different. However, we believe that even with a different mapping, what may change is the split between detection and repair times, thus resulting in the same failed unit outage time. For example, suppose different events are considered to mark the detection time. In that case, we may observe a decrease/increase in the detection time, which adds to the recovery time. Still, the total failed unit outage time would be the same since it represents the duration in which the failed unit was not providing service.

Chapter 7

Container Runtime Upgrade

In this chapter, we quantitatively evaluate container runtime upgrade. In Section 7.1, we discuss the integration of the container runtime tool in Kubernetes and the current practice of upgrading the container runtime. In Section 7.2, we explain the evaluation of container runtime upgrade, the research questions we aim to answer, and the results and analysis of the performed experiments. In this section we also assess the achievable service availability during upgrade. In Section 7.3, we provide potential improvements to some of the analyzed issues and conclude the chapter in Section 7.4.

7.1 Current Practice of Upgrade

Container Runtime is a set of scripts and software tools to run and maintain the work of a container [36]. In a Kubernetes cluster, kubelet manages the container workload on every worker node and communicates with the container runtime to decide on the lifecycle of a container. In Kubernetes version 1.6, Container Runtime Interface (CRI) [37] was introduced to standardize how different container runtimes communicate with the Kubernetes cluster. Figure 7-1 shows how kubelet uses CRI to communicate with the container runtime running on that worker node.

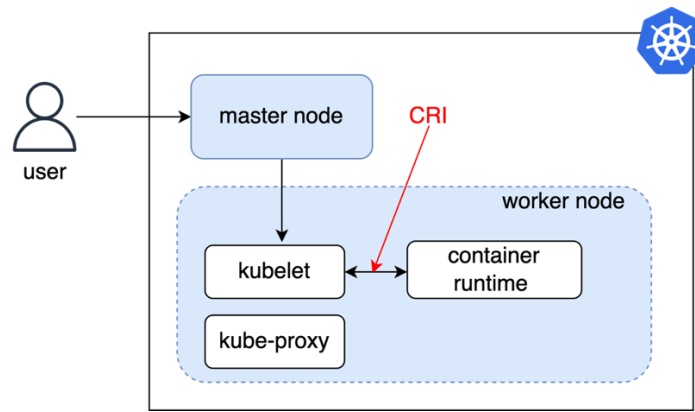


Figure 7-1: Illustration of the placement of CRI in a Kubernetes cluster

CRI is a gRPC [37] Interface and should not be confused with cri-containerd, CRI-O etc. Any container runtime that integrates with the Kubernetes cluster must have the implementation of CRI called CRI shim, shown in the yellow box in Figure 7-2.

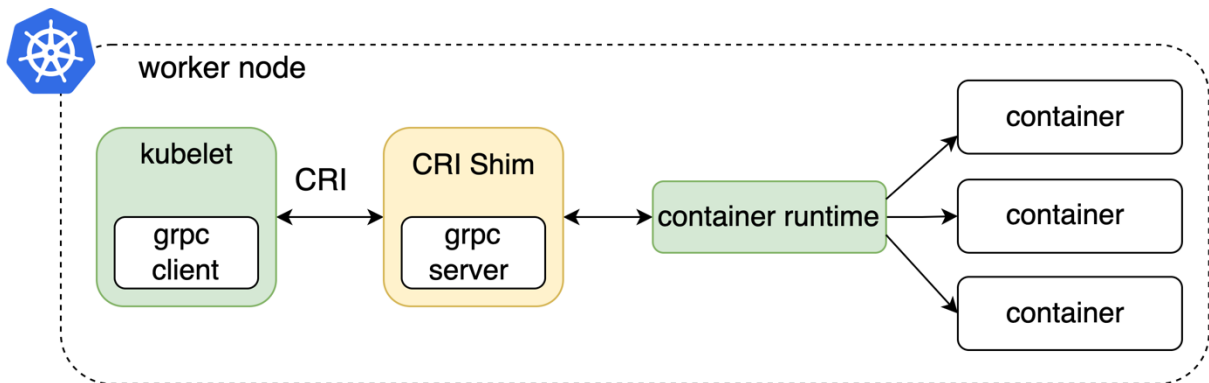


Figure 7-2: Interaction of kubelet with the containers running on that node via CRI

In a Kubernetes cluster, the container runtime must be manually installed and upgraded. For evaluating container runtime upgrade in a Kubernetes cluster, two of the most popular container runtimes, namely Docker and CRI-O, are considered. Based on our analysis and evaluation of the upgrade process of Docker and CRI-O, we have come up with their upgrade process flows, which are presented in Figure 7-4 and Figure 7-7, respectively. In these Figures, the blue indicates the manual operations to be performed by an administrator, while the yellow shows the automated procedures triggered in response to the manual operations. Only actions significant to our work are shown in detail.

7.1.1 Docker

Docker has been the oldest and one of the most popular container runtimes used with Kubernetes. The Docker shim is the CRI layer for facilitating communication with the Docker container runtime. As shown in Figure 7-3, Docker shim converts all the CRI requests issued by Kubernetes into a call to Docker daemon, which is forwarded to containerD that communicates with runC to perform the requested operation on the containers.

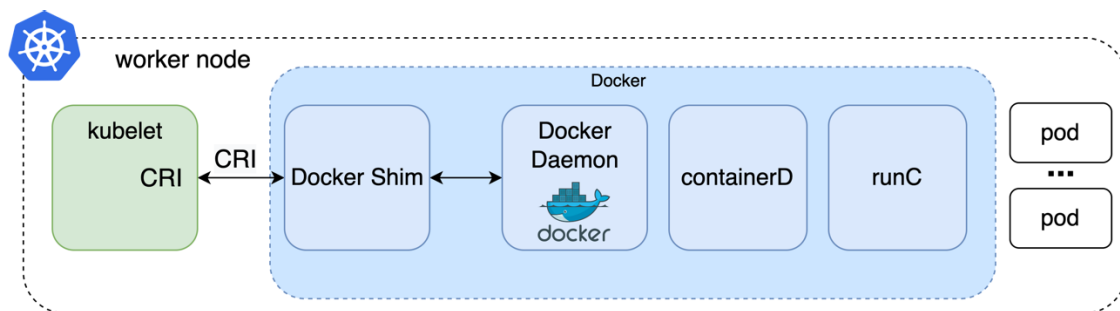


Figure 7-3: Illustration showing Docker as a container runtime in Kubernetes

Also, when a pod is deployed in a Kubernetes cluster integrated with Docker as the container runtime, apart from the application container mentioned in the spec, an additional container for the pod process is created. This pod container acts as a parent for the associated application container, and failure of the pod container also fails the associated application container(s).

Figure 7-4 illustrates the upgrade process flow of the Docker upgrade of a node in a Kubernetes cluster. The upgrade process starts by installing Docker packages and setting the Docker repository. These steps enable installing the desired Docker version from the repository. Next, we install and apply the desired Docker version; this is the primary step in the upgrade process flow. In this step, the desired Docker version is applied, during which Docker terminates all the containers it manages on that node. On a worker node, the containers managed by Docker are the kube-proxy, core-dns component, calico component, pod container,

and application containers of the application hosted by that node. So, all these containers are terminated. The container termination step is skipped if it is a patch version upgrade, and the *live-restore* field in the Docker configuration file (*daemon.json*) is defined as true. Next, Docker is restarted, and once Docker becomes available, it starts all the containers - if terminated earlier. Then, the Docker configuration is updated, and the Docker service is re-started to reflect the version on that node.

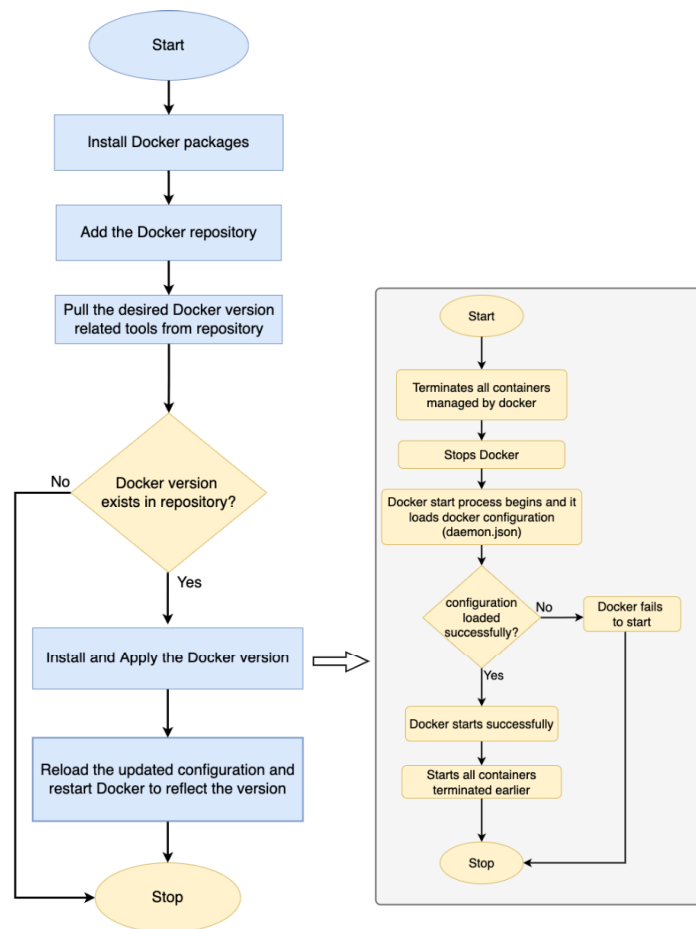


Figure 7-4: Docker upgrade process flow for nodes in a Kubernetes cluster

7.1.2 CRI-O

CRI-O is an implementation of Kubernetes CRI to enable using OCI (Open Container Initiative) [38] compliant runtimes. Its scope is tied to CRI. The CRI-O project in CNCF can provide a typical CRI shim capability that converts the CRI request issued by Kubernetes into

a call to OCI-compliant runtime, such as runC that performs those operations on the container. The combination of these layers is described in Figure 7-5.

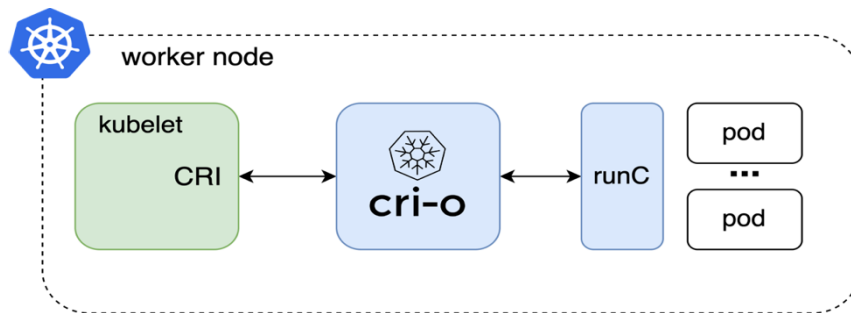


Figure 7-5: Illustration showing CRI-O as a container runtime in Kubernetes

Also, in a Kubernetes cluster integrated with CRI-O as the container runtime, each container is managed by a process called common, as shown in Figure 7-6. Common acts as a parent and is responsible for transmitting information about the container to the CRI-O, which takes administrative actions accordingly. Furthermore, unlike Docker, it does not create an additional container for the pod process.

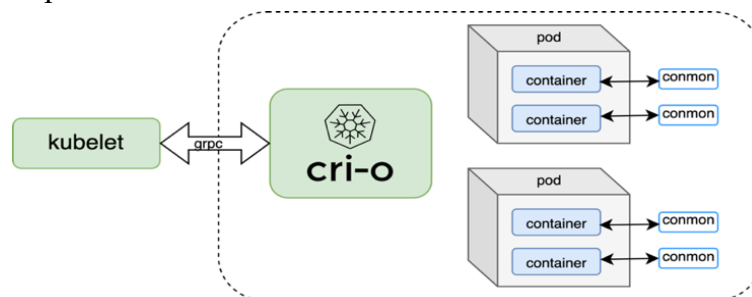


Figure 7-6: Illustration showing the common component of CRI-O

Figure 7-7 illustrates the upgrade process flow of the CRI-O upgrade of a node in a Kubernetes cluster whose container runtime is CRI-O. The upgrade process starts by installing CRI-O packages and setting its repository. Then, the desired (available) CRI-O version is pulled from the repository. In the next step, which is the main step in the upgrade process flow, the desired CRI-O version is applied, and CRI-O is stopped and started (post it loads the CRI-O configuration successfully). Only once CRI-O becomes available it reconciles the state of all the common processes for each container. This way, CRI-O supports live upgrades for its

version. Then the system configuration is updated, and the CRI-O service is re-started to reflect the updated CRI-O version on that node.

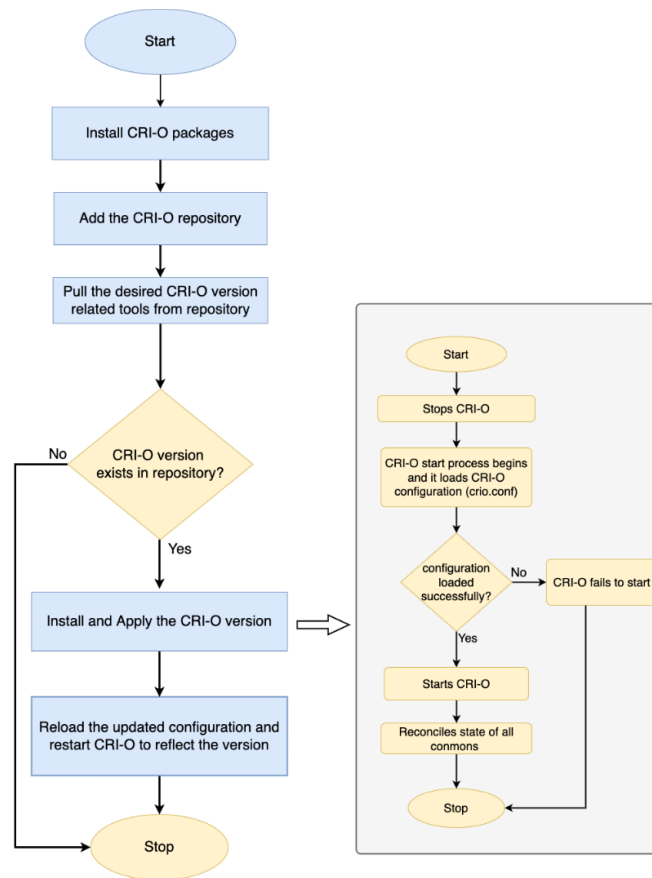


Figure 7-7: CRI-O upgrade process flow for nodes in a Kubernetes cluster

7.2 Evaluation of Container Runtime Upgrade

In this section, we evaluate the upgrade process of Docker and CRI-O as container runtime integrated in a Kubernetes cluster from the perspective of availability. Upgrading a container runtime version may directly impact the service provided by the application, as Kubernetes manages the application by communicating with the container runtime tool via Kubelet (as shown in Figure 7-1). So, we state the following hypothesis:

H₃: High Availability of application deployed in a Kubernetes cluster is impacted when the container runtime upgrades, especially in the presence of failure.

So, to test H_3 we evaluated the upgrade process of container runtime tool and defined the following Research Questions (RQ):

RQ1: What is the impact of container runtime upgrade on the application services?

RQ2: What is the impact of container runtime upgrade on the recovery from application container failure and on application services?

RQ3-1: What is the impact of container runtime upgrade process failure on the services provided by the application?

RQ3-2: How is the process of recovery handled from container runtime upgrade process failure

- A. When the failure is external
- B. When the failure is internal

To answer these research questions, we performed experiments and evaluated their impact. Also, to assess the maximum impact on the application services, we evaluated the container runtime upgrade of the worker node that hosts these application pods. We considered a single master and two worker node kubeadm created Kubernetes cluster. Unlike Docker, in CRI-O, an additional container for the pod process is not created. To make the results comparable between Docker and CRI-O, we considered the application container failure scenario for RQ2. For Docker, we upgraded the minor, major and patch releases. For CRI-O, we evaluated the upgrade of minor and patch releases (as the major version is not released yet). All the experiments are performed in the default configuration of Kubernetes and the container runtime. The process of collection of metrics was automated by the Auto-Metric collector tool, which is explained in Chapter 4.

7.2.1 RQ1: Evaluating the Impact of Container Runtime Upgrade on Service Availability of Hosted Application Instances

7.2.1.1 Experiments

The experiment aimed to evaluate the impact of container runtime upgrades on the service availability of the application hosted on that worker node. While the container runtime of a worker node was being upgraded, we monitored the pod-related events in the Docker/CRI-O and kubelet logs. The impact on application services is measured as service outage and service degradation.

For stateful application, we upgrade the container runtime of the worker node bearing the active pod to evaluate the maximum impact on the application service.

7.2.1.2 Evaluating the Impact of Docker Upgrade on Application Services

During the Docker upgrade, for patch releases, for both stateful and stateless applications, we do not observe any impact on the application services. As described in sub-section 7.1.1, since we enabled the *live-restore* functionality of Docker, the pods are not terminated due to the Docker upgrade, so the application service is not impacted during an upgrade.

During Docker upgrade for major and minor releases, we present our observations and analysis as follows:

- For the stateless application, the pods are recreated, but due to the nature of the application, there is no service outage observed, as the pods running on other worker nodes continue to provide the application service. However, there is an observed service degradation of 12.891 seconds, as shown in Table 7-1. As analysed, this happens because of the upgrade process of Docker in which it terminates all the containers managed by it, i.e., application pod, Kube-proxy, calico (the networking component), and core-dns

Pods, before the Docker restarts. Once the Docker is back and running, it restarts all the containers terminated earlier. Also, only after the networking component (kube-proxy, Calico, Core-DNS) is restarted and becomes functional, the application pod is assigned an IP address. So, the service degradation observed and measured is for the total duration when the Docker service remains unavailable because of its restart due to the upgrade process and the time it takes to re-create the application container, which was earlier terminated on account of the upgrade.

- For stateful application, we observe a service outage of 0.681 seconds, as shown in Table 7-1; this happens because of the Docker upgrade process, which involves terminating and restarting the pod managed by it. In this experiment, since we upgraded the Docker version of the worker node hosting an active pod, the pod termination causes the VLC application to stop streaming the video, and the application experiences a service outage. When the State controller is informed about this event, it performs the failover to the available standby pod. The observed service outage is the time the State controller takes to complete the failover.

Table 7-1: Impact on application services during Docker upgrade

Scenario	Kind of Application	Metrics (unit: seconds)	
		Service outage	Service degradation
Docker upgrade	Stateless application	0	12.891
	Stateful application	0.681	N/A

7.2.1.3 Evaluating the Impact of CRI-O Upgrade on Application Services

During the CRI-O upgrade of the worker node, for minor and patch releases, for both stateless and stateful applications, we observed no impact on the application pods hosted by

that worker node. As analyzed, during the upgrade process of CRI-O, the containers managed by it are not terminated, and thus the application service is not impacted.

7.2.2 RQ2: What is the Impact of Container Runtime Upgrade on Recovery from Application Container Failure and on Application Services?

7.2.2.1 Experiment

The experiment aimed to evaluate the impact of container runtime upgrades on the recovery from an application container failure that was simulated by killing the application process running on the OS. The impact on failure recovery was measured in terms of failed unit outage time, and the impact on application services was measured in terms of service outage and service degradation.

For better comparison and to understand the impact, we injected the application container failure in the absence of the upgrade and then while performing the container runtime upgrade. First, application container failure was injected in the absence of an upgrade, which gave us a clear picture of the time it usually takes to recover the failed unit and the application services. Then, while the container runtime of a worker node was upgrading, we injected application container failure on another worker node; to evaluate the impact of the upgrade on application services.

For stateful application, we injected failure in the application container of the active pod, i.e., the pod actively providing the service, to see the maximum impact on the application service.

7.2.2.2 Evaluating the Impact of the Docker Upgrade

During Docker upgrade for major and minor releases, we present our observations and analysis as follows:

Impact on Failure Recovery

For the stateless and stateful application, when the application container is injected with failure on one of the worker nodes while another worker node's Docker version is upgraded, we do not notice any additional impact on the failure recovery. As analyzed, the kubelet component on that worker node (whose application container is failed) detects the failure, and it reacts according to the defined restart policy (by default set to Always) and restarts the container. So, the failure recovery is not impacted due to the ongoing container runtime upgrade on another worker node. The measurements are presented in Table 7-2 and Table 7-3.

Impact on Service Degradation

For stateless application, we observed an increased service degradation of 12.738 seconds, as highlighted in red in Table 7-2 and Table 7-3. As analyzed, this happens because when the Docker is upgraded (on one of the worker nodes), all the available containers managed by Docker on that worker node are terminated; then, Docker is restarted, followed by the re-creation of the terminated containers. So, the application service is degraded for the total duration of the Docker upgrade until the application container is re-created.

Impact on Service Outage

For stateless applications, we observe a service outage of 1.235 seconds, as highlighted in red and shown in Table 7-2. As analyzed, when we inject application container failure in one of the application pods, while the other application container is removed because of the upgrade, no pods are available to provide application service. As the time taken to recover the application container terminated as part of the upgrade is more than the recovery of the failed application container, the failed application container becomes available before the application container removed due to the upgrade is recreated. So, the observed service outage is the duration it takes to recover the failed application container.

For stateful application, we observe an increased service outage of 2.941 seconds, as highlighted in red and shown in Table 7-3. Once the application container associated with the active pod is injected with failure, the state controller is notified. It tries to perform a failover to the standby pod, but as the standby pod is not available on account of its removal due to the container runtime upgrade, the failover is not possible. So, the State controller waits for a pod to become available, and the application continues to experience downtime. Now, since the failed application container becomes available before the application container removed due to container runtime upgrade, the State controller assigns its associated pod with the active label, and the application resumes service. The observed service outage of 2.941 seconds is the sum of the durations taken for the restart of the failed application container, i.e., 2.300 seconds, and the time taken by the state controller to assign an active label to its associated pod, i.e., 0.641 seconds so that application resumes application service.

Table 7-2: Stateless application container failure during Docker upgrade

Scenario	Metrics (unit: seconds)					
	Detection Time	Repair Time	Assignment Time	Failed unit outage time	Service degradation	Service outage
Without upgrade	0.804	0.201	0.151	1.156	1.156	0
During upgrade	0.897	0.194	0.144	1.235	12.738	1.235

Table 7-3: Stateful application container failure during Docker upgrade

Scenario	Metrics (unit: seconds)				
	Detection Time	Repair Time	Assignment Time	Failed unit outage time	Service outage
Without upgrade	0.841	1.265	0.140	2.236	1.488
During upgrade	0.866	1.319	0.115	2.300	2.941

7.2.2.3 Evaluating the Impact of the CRI-O Upgrade

During CRI-O upgrade for minor and patch releases, we present our observation and analysis as follows:

Impact on Failure Recovery/Service Degradation

For both stateless and stateful applications, when the application container is failed on one of the worker nodes, while the CRI-O version of another worker node is upgraded, we do not observe any additional impact on the failure recovery. As analyzed, the kubelet component on that worker node (whose application container is failed) detects the failure, where it reacts according to the defined restart policy and restarts the container. So, the failure recovery is not impacted due to the ongoing CRI-O upgrade on another worker node.

Also, for stateless application, since the containers managed by CRI-O are not terminated during their upgrade, so the upgrade does not cause any additional service degradation. The observed service degradation is the time the application container takes to recover from failure, which is equal to the failed unit outage time. The results of the measurements are presented in Table 7-4.

Impact on Service Outage

As discussed in sub-section 7.1.2, unlike the Docker upgrade, the containers are not terminated in the case of the CRI-O upgrade, so the application services provided by those containers are not impacted.

So, for stateless application, when we inject failure in one of the application containers, another application container keeps running on another worker node (whose CRI-O is being upgraded) and remains available to provide application service. Thus, the application does not experience any service outage.

In the case of stateful application, when the application container associated with the active pod is injected with failure during an ongoing CRI-O upgrade of the worker node hosting application container associated with the standby pod, we do not observe any additional impact on the service outage time. This happens because of the nature of the upgrade of CRI-O, where

the containers managed by it are not impacted. So, when the State controller is informed about the failure of the application container of the active pod, it is able to perform failover to the available standby pod. Thus, an ongoing upgrade does not cause any additional impact on the service outage. The measurements of the experiments are presented in Table 7-4 and Table 7-5, for stateless and stateful application, respectively.

Table 7-4: Stateless application container failure during CRI-O upgrade

Scenario	Metrics (unit: seconds)				
	Detection Time	Repair Time	Assignment Time	Failed unit outage time / Service degradation	Service outage
Without upgrade	0.740	0.226	0.140	1.106	0
During Upgrade	0.859	0.240	0.127	1.226	0

Table 7-5: Stateful application container failure during CRI-O upgrade

Scenario	Metrics (unit: seconds)				
	Detection Time	Repair Time	Assignment Time	Failed unit outage time	Service outage
Without upgrade	0.820	1.305	0.135	2.260	1.467
During Upgrade	0.769	1.279	0.141	2.189	1.423

7.2.3 RQ3-1: Evaluating the Impact of Container Runtime Upgrade Process Failure on Application Services

7.2.3.1 Experiments

The experiment aims to analyze the impact of upgrade process failure on the availability of the application and if the container runtime engine provides any recovery mechanism.

External failure was simulated by upgrading the container runtime to an incorrect version. As the upgrade process involves pulling the image from the repository, if the container runtime version is non-existent, it wouldn't be available in the container runtime repository, and as soon as the incorrect version is detected, the upgrade process fails.

Internal failure was simulated by misconfiguring the container runtime configuration file (*daemon.json* for Docker, *crio.conf* for CRI-O). Container runtime refers to this configuration file during its restart and fails to start due to misconfiguration, thus failing the upgrade process.

7.2.3.2 Evaluating the Impact on Application Services because of Container Runtime Upgrade Process Failure due to External Failure

Since the results and the analysis of the performed experiment are same for Docker and CRI-O, we present them together.

For Docker and CRI-O, due to external failure, we observe no impact on the application, as the version of the container runtime specified is non-existent, its image wouldn't be available in the repository to be pulled, the upgrade will fail even before the version are applied and container runtime is restarted, as shown in Figure 7-4 and Figure 7-7.

7.2.3.3 Evaluating the Impact on Application Services because of Container Runtime Upgrade Process Failure due to Internal Failure

As analyzed, on a worker node, if its container runtime upgrade fails due to internal failure, the Docker/CRI-O service fails. As the kubelet of the worker node creates and sends lease updates to the master node every 10 seconds (default interval), the lease update fails in the event of container runtime failure. Next, the kubelet retries it using exponential backoff that starts at 200 milliseconds and is capped at 7 seconds [39]. Thus, in the case of container runtime failure, Kubernetes takes around 7 seconds to 17 seconds to mark the impacted node as *NodeNotReady*. Once the node is marked as *NodeNotReady*, Kubernetes waits for the *pod-eviction timeout* duration (by default 300 seconds) before the pods running on the worker node whose container runtime service has failed are marked for deletion. Pods marked for termination are removed and created on an available healthy worker node. Furthermore, depending on

the way Docker and CRI-O handle the application during its upgrade, the application availability is impacted. So, we present their analysis separately as follows:

Impact on Application Services during Docker Upgrade Process Failure

During the Docker upgrade process failure, the application services are impacted. As analyzed, due to the upgrade process flow of Docker (as mentioned in sub-section 7.1.1 and Figure 7-4), the containers managed by Docker are terminated, then the Docker engine is restarted. During the restart of the Docker engine, it refers to its Docker configuration, and since it was misconfigured to simulate an internal failure, the Docker engine fails to start. As a result, all the containers terminated earlier are not started since the Docker engine responsible for their start has failed. Although, as analyzed and explained earlier, Kubernetes eventually marks the node's status as *NodeNotReady* because of lease update failure, this causes downtime for any application instance running as pods hosted on that node. It is only post *pod-eviction-timeout* duration that these application pods are recreated on an available healthy worker node; what follows is the impact on application services measured as service degradation and service outage:

Impact on Service Degradation: Stateless application experiences a service degradation of 316.919 seconds, as shown in Table 7-6. This is the total duration Kubernetes takes: to mark the node's status as *NodeNotReady*, to evict the pod running on the impacted node, and to recreate the pod on an available healthy worker node.

Impact on Service Outage: Stateless application do not experience service outage due to the nature of the application, as pods running on another worker node continue to provide service. However, the stateful application experiences a service outage of 15.597 seconds when container runtime upgrade failure occurs on a node running an active pod, as presented in Table 7-6. This happens because, as observed in this case, it takes 14.950 seconds for Kubernetes to

mark the node's status as *NodeNotReady*. When it does, the State controller gets information about this event and performs the failover to the available standby pod running on another worker node, and the application resumes its service. So, the application experiences a service outage for the total duration for which Kubernetes marks the impacted node as *NodeNotReady* and the time taken by the State controller to perform failover to the available standby pod.

Table 7-6: Impact on the application services due to Docker upgrade process failure (internal failure)

Scenario	Kind of Application	Metrics (unit: seconds)	
		Service outage	Service degradation
Docker upgrade process failure – Internal failure	Stateless application	0	316.919
	Stateful application	15.597	N/A

Impact on Application Services during CRI-O Upgrade Process Failure

Unlike Docker, in case of CRI-O upgrade process failure on a worker node (due to internal failure), the application pod running on the worker node does not get impacted. However, they become failure intolerant as the container runtime is not available (due to its failure) to recover them in the event of failure. Furthermore, as analyzed and explained earlier, Kubernetes eventually marks the node as *NodeNotReady*, but only post *pod-eviction-timeout* duration, any application instance hosted on that node, is recreated on an available healthy worker node, what follows is the impact on application services measured as service degradation and service outage:

Impact on Service Degradation: In the case of a stateless application, it experiences service degradation of 2.014 seconds, as presented in Table 7-7. As analysed, the observed service degradation is the time duration in which the application pods are created and started on another worker node after it is evicted from the old node (marked as *NodeNotReady*) post *pod-eviction-timeout* duration.

Impact on Service Outage: Stateless application does not experience service outage as the pod running on another worker node continues to provide service. However, the Stateful application experiences a service outage of 0.654 seconds. So, post container runtime failure of the worker node hosting an active pod, when the lease update fails, Kubernetes marks the worker node as *NodeNotReady*. At this time, the State controller gets information about this event and performs the failover to the available standby pod running on another worker node, and the application resumes its service. The observed service outage is the time the State controller takes to perform failover.

Table 7-7: Impact on the application services due to CRI-O upgrade process failure (internal failure)

Scenario	Kind of Application	Metrics (unit: seconds)	
		Service outage	Service degradation
CRI-O upgrade process failure – Internal failure	Stateless application	0	2.014
	Stateful application	0.654	N/A

7.2.4 RQ3-2: Evaluating the Recovery Actions Taken by Container Runtime Tools when their Upgrade Process Fails

7.2.4.1 Experiments

In this evaluation, we present our observation as an extension of the experiment performed in RQ3-1. So, in RQ3-1, when the container runtime upgrade process failed, we observed if the container runtime identified this failure and if they took any measures to restore the application's state. The aim was to evaluate the failover offered by the respective container runtime when the Kubernetes container runtime upgrade process fails.

7.2.4.2 Evaluating the Recovery Actions when the Failure is External

No recovery operation is required in case of a container runtime upgrade process failure due to external failure, as the Docker is not restarted, and the container runtime remains available. We observed and analyzed the same behaviour for a Kubernetes cluster integrated with CRI-O as the container runtime.

7.2.4.3 Evaluating the Recovery Actions when the Failure is Internal

In case of a container upgrade process failure due to internal failure, the container runtime fails, causing the lease update to fail, and due to this Kubernetes changes the status of that node to *NodeNotReady*. The recovery process, in this case, must be manually handled. Once the cause of the error is manually detected and resolved, the Docker / CRI-O should be restarted to reflect the changes. Docker or CRI-O took no remediation measures.

7.2.5 Assessing the Achievable Service Availability, during Container Runtime Upgrade, and in Presence of Failure during Upgrade (H₃)

In this sub-section, we discuss the results of our evaluation to test our hypothesis H₃. For Docker, its edge version containing improvement and bug-fixes is released every 1-2 month, and stable version is released every quarter [40]. This concludes that a Docker must be upgraded for at most 16 times (12 edge and 4 stable releases) in a year. While for CRI-O that is a dedicated container runtime for Kubernetes cluster, it follows the release-cycle of Kubernetes minor version (that are released every quarter) but does not follow Kubernetes patch version cycle (releasing every week in a year). In CRI-O, patch versions are released when necessary [41]. Since number of patch version upgrades are not mentioned by CRI-O, for our calculation we consider maximum CRI-O upgrades in sync with Kubernetes version releases i.e., 56 times a year. Next, we utilize the measurements of our evaluation to calculate service availability using Equation 2. The calculations represent the service availability achieved when

the Docker is upgraded for 16 times a year, and CRI-O is upgraded 56 times a year in a Kubernetes cluster. Each service availability calculation (for every evaluated scenario) considers the container runtime upgrade of every worker node in the cluster. Also, these calculations are specific to our experiment settings and its associated measurements.

7.2.5.1 Service Availability during Upgrades

We utilized the measurements of our evaluations as presented in Table 7-1, to calculate the service availability during Docker upgrade. As shown in Figure 7-8, our calculations indicate that the high availability of 99.999% per year is maintained for the deployed application (stateless and stateful) when Docker is upgraded. In case of CRI-O, as explained in sub-section 7.2.1.3, the deployed application is not impacted during its upgrade. Thus, high availability of application is maintained when CRI-O upgrades in a Kubernetes cluster.

7.2.5.2 Service Availability during Upgrades in the Presence of Failure

Next, we evaluated the failure scenarios during Docker and CRI-O upgrade, as presented in sub-section 7.2.2 for application pod failure, and sub-section 7.2.3 for upgrade process failure.

As presented in Table 7-2 and Table 7-3, Docker upgrade in the presence of application pod failure causes service outage for the application. We utilized the results of these measurements to calculate service availability, and as shown in Figure 7-8, our calculation indicate that the high availability of the application is not impacted. In the case of CRI-O upgrade in presence of application pod failure, we calculated service availability by using the measurements presented in Table 7-4 and Table 7-5, and our calculation as shown in Figure 7-8, indicates that the high availability of the deployed applications is maintained in this scenario.

In the event of upgrade process failure, we utilize the measurements presented in Table 7-6 and Table 7-7 to calculate the service availability achieved in case of Docker and CRI-O

respectively. The calculation as shown in Figure 7-8, indicates that the high availability of the application is maintained even when Docker or CRI-O upgrade process fails.

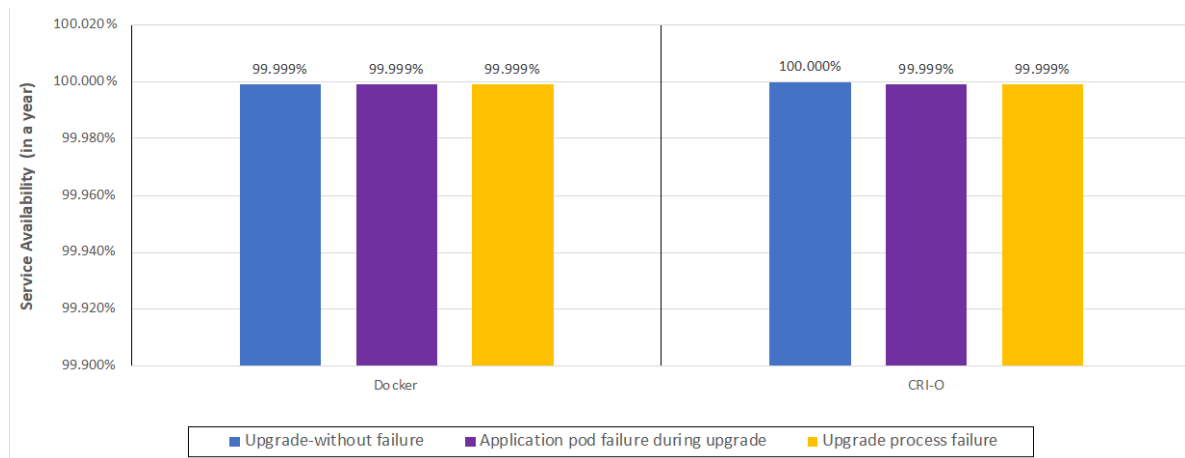


Figure 7-8: Service availability achieved when container runtime integrated with Kubernetes cluster upgrades for a year (with and without failure)

7.2.5.3 Conclusion

As shown in Figure 7-8, our results indicates that the high availability of application deployed in a Kubernetes cluster is not impacted when Docker and CRI-O upgrades (with and without failure) for their respective upgrade cycle in a year. Thus, these results of our evaluation refute our Hypothesis H₃.

7.3 Overall Analysis and Potential Improvements

In this section, we provide a summary of the evaluation and issues identified during the Container runtime upgrade on a worker node in a Kubernetes cluster. Then, we discuss the issues not handled by Kubernetes and provide a potential solution.

Through our investigations, we learnt that application container failure during the Docker upgrade causes an additional impact on the managed application services. However, such impact can be avoided by using CRI-O as a container runtime. We also discovered that a Kubernetes cluster integrated with CRI-O offers better service availability when compared with a Kubernetes cluster integrated with Docker. Furthermore, we analyzed that Kubernetes

depends on its interaction with the Container runtime to manage the pod, and in the absence of this communication, it cannot do so. For example, in the case of container runtime upgrade process failure due to internal failure, Kubernetes do not discover the actual state of the containers hosted on the impacted worker node, post its container runtime fails. Although in the mentioned case, Kubernetes changes the status of the impacted node to *NodeNotReady*, but the time it takes to handle the impact caused due to upgrade process failure is more than 300 seconds, which is the total allowed downtime over one year for the systems with high availability requirements. So, we proposed improvement in this case to reduce this time.

We present potential improvements to solve the issues identified during container runtime upgrade process failure (due to internal failure): First, the time taken by Kubernetes to detect the failure and handle the applications impacted is more than 300 seconds; Second, the affected node must be manually recovered.

Before proposing the solution, we explain our analysis of the upgrade process failure. When the container runtime service (of the worker node) fails (as defined in sub-section 7.2.3.3), we see that the application service gets impacted for Docker integrated Kubernetes cluster, and the application service is maintained in the CRI-O integrated Kubernetes cluster, what follows is the detailed step to understand this process:

- Due to the unavailability of container runtime service, kubelet is unable to fetch the status of the containers and send the status to the master node; in the absence of these updates, the master node marks the node's status as "*NodeNotReady*".
- Then, Kubernetes waits for 300 seconds (by default), after which the pods on the worker node are marked for deletion and removed from the service endpoint list. It then creates a new pod on the available healthy worker node.

In the case of CRI-O, as the pods become failure intolerant and lose connection with the master, so in the case of any pod failure, it will neither be communicated to the master node, nor the failed pod would be recovered. In the case of Docker, application instances hosted on the impacted node remain unavailable, but Kubernetes is unaware of their current state. For both Docker and CRI-O, once Kubernetes marks the affected worker node as *NodeNotReady*, it is only after 300 seconds that Kubernetes marks these pods for deletion and creates a new pod in the available worker node.

The following measure can be taken to reduce the impact of possible disruption due to the mentioned problems:

1. **Backup:** The upgrade manager tool should take the backup of the container runtime config file even before starting the container runtime upgrade.
(Docker – “etc/docker/daemon.json”, CRI-O – “etc/crio/crio.conf”)
2. **Identification of Failure:** When container runtime fails due to misconfiguration of the config file, the upgrade manager tool can look out for the occurrences of the below-mentioned two events occurring simultaneously in this scenario, along with checking the status of the container runtime service.
 - The first event would be “*NodeNotReady*” and,
 - The second event for the same node would be “*ContainerGCFailed*” or “*ImageGCFailed*”
3. **Restoration:** If failure is identified (in step 2), copy the file from the Backup (in step 1) and place it in the container runtime’s respective config file location.
(Docker - “etc/docker/”, CRI-O - “etc/crio/”)

- 4. Reload and Restart the Container Runtime Services:** This will stabilize the condition of the node by regaining communication with the master node, thus avoiding pod eviction.

The proposed improvement reduces the time associated with handling the impact on the deployed pods of the worker node (whose container runtime service fails during the upgrade). The measured impact, as presented in Table 7-6, will be reduced to only the time taken by Kubernetes to detect and mark the impacted worker node as *NodeNotReady*. So, following the steps mentioned as solution earlier, the impact on application service is reduced to approximately 94%. Furthermore, the proposed improvement also provides a way to recover the worker node from container runtime upgrade process failure, thus helps to maintain the capacity of the Kubernetes cluster.

7.4 Conclusion

In this chapter, we presented and evaluated container runtime upgrades in a Kubernetes cluster for stateless and stateful applications. We assessed the service availability achieved when Docker and CRI-O upgrades (with and without failure) during their respective upgrade cycle in a year. Through our calculations, we learnt that the high availability of the application is not impacted when container runtime upgrades. Also, the service availability calculations presented are specific to the cluster setting, scenarios and container runtime tool considered in our work, and these calculations may vary in other cases. For example, when upgrades happen in large clusters, service availability may differ depending on the number of nodes whose container runtime is upgraded at once or the distribution of application pods across the nodes in the cluster, or the additional impact caused by container runtime during its upgrades. With our calculation, we provide an understanding of achievable service availability in a small cluster, where the number of application pods is distributed evenly in a cluster, and container runtime

on each node is upgraded one at a time. Our evaluation aims to provide insight into handling such upgrades in a larger cluster with complex applications.

Then, we analyzed the results of our evaluation and discovered shortcomings of the evaluated container runtime to recover from upgrade process failure (caused due to internal failure) and limitations of Kubernetes to handle the impact of such failure on application services. So, we proposed potential improvements that reduce the impact on application services by approximately 94%. Furthermore, the proposed improvement also provides a way to recover the impacted worker node, so it helps to maintain the cluster's capacity.

We acknowledge that there are some threats to the validity of our results. For example, all our experiments are conducted on a small cluster consisting of only a master and two worker nodes. Kubernetes may behave differently in larger clusters, that may impact the measurements presented in our experiments. Also, the extent to which the provided measurements accurately assess its intent is another threat to validity related to the tools and mechanisms used in our experiments. We rely on the events and associated timestamps reported in Kubernetes and Container runtime logs. However, to reduce any intended impact, we consistently use the same timestamp extraction process throughout all our experiments. The mapping of the metrics to the concrete events is the biggest threat and requires more investigation as one can map them differently, in which case all the measurements could be different. However, we believe that even with a different mapping, what may change is the split between detection and repair times, thus resulting in the same failed unit outage time. For example, suppose different events are considered to mark the detection time. In that case, we may observe a decrease/increase in the detection time that adds to the recovery time, or inversely, but the total failed unit outage time would be the same since it represents the duration in which the failed unit was not providing service.

Chapter 8

Conclusion

In this thesis, we identified various upgrade levels in the context of a Kubernetes cluster, i.e., Kubernetes cluster version upgrade, Kubernetes application upgrade and Container runtime upgrade. We performed different experiments to evaluate these levels and defined metrics to analyze the results of the performed experiments. To automate the metric collection, we devised and implemented the Auto-Metric collector tool; it monitors the defined events, collects their timestamps, and calculates the defined metrics automatically. Then, we analyzed the results from the perspective of service availability, identified the causes of the shortcomings, and proposed potential improvements; to automatically handle the cases not addressed by Kubernetes.

The main goal of implementing the Auto-Metric collector tool is to lessen manual efforts and avoid human error. This tool effectively observes the failure events, collects their event timestamps, and calculates the defined metrics. Finally, we evaluate the calculated metrics to analyze the impact of the upgrade on application services and failure recovery.

In the Kubernetes cluster version upgrade, we evaluated the Kubernetes version upgrade of a cluster created and managed by kubeadm and the kOps tool. Through our experiments, we also learnt that the upgrade process in a kOps managed Kubernetes cluster (on AWS) drastically impacts the application service and failure recovery compared with the kubeadm created Kubernetes cluster. We evaluated the results of the performed experiments to calculate the

achievable service availability when Kubernetes cluster version is upgraded in a year. The results state that high availability of application's service is impacted in the event of failure during upgrade. Through our evaluation and analysis, we observed that the kubeadm tool managing the Kubernetes cluster does not handle its upgrade process failure, so we proposed a potential improvement for this case. In our improvement, we analyze the state of the master node to detect if the upgrade process has failed, and if detected, we re-initiate the upgrade process to complete the Kubernetes version upgrade of the master node. Our improvement ensures that all master node components are in the intended version and that the configuration file is updated with the correct Kubernetes version so that the remaining nodes can refer to it for their upgrade.

In the Kubernetes application upgrade, we presented and evaluated different application upgrade strategies provided by Kubernetes for stateless and stateful applications. Through our experiments, we discovered that application upgrade using the Recreate upgrade strategy causes the maximum service outage. For both stateless and stateful applications, during the RollingUpdate strategy, we investigated that Kubernetes gives priority to failure recovery over upgrades to follow, due to which the failure recovery time does not get impacted during an upgrade. However, in Recreate strategy for stateless application, the failed pod is not recovered, and the upgrade process continues. We considered the results of the performed experiments to calculate the achievable service availability when application is upgraded in a year. The results state that high availability of application's service is impacted only during upgrade process failure scenario. Through our experiments, we also analyzed that during upgrade process failure (simulated using a non-existent image version in the specification file), the Deployment/StatefulSet controller neither detects the failure nor takes any actions to stop the process of pod creation; as a result, the pod keeps trying to pull the non-existent image version,

thereby impacting application services. Also, the respective controller does not trigger automatic rollback to a stable application version; it requires manual intervention for such restoration. So, we provide potential improvements to solve these problems to reduce the impact on application services in the event of the upgrade process failure.

In the Container runtime upgrade, we presented and evaluated the upgrade process of Docker and CRI-O as container runtime tools integrated in a Kubernetes cluster. Through the different experiments performed, we learnt that application container failure during the Docker upgrade induces an additional impact on application services; however, CRI-O supports live upgrades, thereby causing zero downtime for the managed application services when it upgrades. We also discovered that a Kubernetes cluster integrated with CRI-O as a container runtime tool offers better service availability than Docker. We assessed the results of the performed experiments to calculate the achievable service availability when container runtime is upgraded during their upgrade cycle in a year, which states that application's high service availability is not impacted during container runtime upgrade. We learnt that the time taken by Kubernetes to handle the impact caused due to Docker upgrade process failure is 300 seconds, which is the total allowed downtime over one year for the systems with high availability requirements. So, we proposed improvements that reduce the time to handle the upgrade process failure by approximately 94%. Furthermore, the proposed improvement also recovers the impacted worker node, so it helps to maintain the cluster's capacity.

This thesis investigated three different levels of upgrade in a Kubernetes cluster, reported on experiments performed to evaluate these upgrades, provided analysis, assessed the service availability achieved during upgrades, and proposed potential improvements from some of the identified shortcomings. The next logical step is to devise and implement an upgrade manager

tool to manage upgrades in a Kubernetes cluster while reducing the impact on service availability.

Bibliography

- [1] M. Toeroe and F. Tam, *Service Availability: Principles and practice*, John Wiley & Sons, 2012.
- [2] M. Nabi, M. Toeroe, and F. Khendek, "Availability in the cloud: State of the art," *Journal of Network and Computer Applications*, vol. 60, no. 2016, pp. 54-67.
- [3] Kubernetes 2022, "Overview," [Online]. Available: <https://kubernetes.io/docs/concepts/overview/>. [Accessed 10 September 2022].
- [4] M. Nabi, "Automating the Upgrade of IaaS Cloud Systems," 2019, pp. 11-12.
- [5] S. Fu, J. Liu, X. Chu, and Y. Hu, "Toward a Standard Interface for Cloud Providers: The Container as the Narrow Waist," *IEEE Internet Computing*, vol. 20, no. 2, pp. 66-71.
- [6] "Deploy on Kubernetes," Inc, Docker, [Online]. Available: <https://docs.docker.com/desktop/kubernetes/>. [Accessed 17 August 2022].
- [7] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81-84, 2014.
- [8] Cloud Native Computing Foundation, "Foundation, Kubernetes | Cloud Native Computing," The Linux Foundation, [Online]. Available: <https://www.cncf.io/projects/kubernetes/>. [Accessed 2022 September].

- [9] etcd-io, "etcd/README.md at main · etcd-io/etcd," [Online]. Available: <https://github.com/etcd-io/etcd/blob/main/raft/README.md>. [Accessed 23 July 2022].
- [10] "cri-o," Cloud Native Computing Foundation, [Online]. Available: <https://cri-o.io/>. [Accessed 12 August 2022].
- [11] "GitHub - containernetworking/cni: Container Network Interface - networking for Linux containers," containernetworking, [Online]. Available: <https://github.com/containernetworking/cni> . [Accessed 08 August 2022].
- [12] The Kubernetes Authors, "Nodes Conditions," 05 April 2022. [Online]. Available: <https://kubernetes.io/docs/concepts/architecture/nodes/#condition>. [Accessed 05 July 2022].
- [13] The Kubernetes Authors, "Kubeadm," [Online]. Available: <https://kubernetes.io/docs/reference/setup-tools/kubeadm/>. [Accessed July 2022].
- [14] "Welcome - kOps - Kubernetes Operations," [Online]. Available: <https://kops.sigs.k8s.io/>. [Accessed 14 July 2022].
- [15] "GKE overview | Google Kubernetes Engine (GKE) | Google Cloud," [Online]. Available: <https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview>. [Accessed 13 July 2022].
- [16] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A kubernetes controller for managing the availability of elastic microservice based stateful applications," *Journal of Systems and Software*, vol. 175, no. 0164-1212, p. 110, 2021.

- [17] P. A. 2014-2022, "Prometheus - Monitoring system & time series database," [Online]. Available: <https://prometheus.io/>. [Accessed August 2022].
- [18] T. K. Authors, "Deploy and Access the Kubernetes Dashboard," [Online]. Available: <https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>. [Accessed 18 August 2022].
- [19] "Analyzes resource usage and performance characteristics of running containers," GitHub - google/cadvisor, [Online]. Available: <https://github.com/google/cadvisor> . [Accessed 22 July 2022].
- [20] E. B.V, "Filebeat overview | Filebeat Reference [8.4] | Elastic," [Online]. Available: <https://www.elastic.co/guide/en/beats/filebeat/current/filebeat-overview.html>. [Accessed 14 August 2022].
- [21] "What is Elasticsearch?," Elasticsearch, [Online]. Available: <https://www.elastic.co/what-is/elasticsearch>. [Accessed 14 August 2022].
- [22] "ntp.org: Home of the Network Time Protocol," [Online]. Available: <http://www.ntp.org/>. [Accessed 15 August 2022].
- [23] "Version Skew Policy," Kubernetes, [Online]. Available: <https://kubernetes.io/releases/version-skew-policy/#supported-versions>. [Accessed 12 August 2022].
- [24] The Kubernetes Authors, "Upgrade A Cluster," Kubernetes, [Online]. Available: <https://kubernetes.io/docs/tasks/administer-cluster/cluster-upgrade/>. [Accessed 16 July 2022].

- [25] The Kubernetes Authors, "Creating a cluster with kubeadm," [Online]. Available: <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/>. [Accessed 22 September].
- [26] "Working with Instance Groups - kOps - Kubernetes Operations | Sigs.k8s.io," [Online]. Available: <https://kops.sigs.k8s.io/tutorial/working-with-instancegroups/>. [Accessed 3 September 2022].
- [27] The Kubernetes Authors, "Understanding Kubernetes Objects, Kubernetes," [Online]. Available: <https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/#object-spec-and-status>. [Accessed 12 June 2022].
- [28] The Kubernetes Authors, "ReplicaSet," Kubernetes, [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>. [Accessed 16 June 2022].
- [29] The Kubernetes Authors, "Deployments," Kubernetes, [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#updating-a-deployment>. [Accessed 24 June 2022].
- [30] The Kubernetes Authors, "StatefulSets," Kubernetes, [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#update-strategies>. [Accessed 4 July 2022].
- [31] NGINX, "'How often are NGINX and NGINX Plus released?'," NGINX, 25 Sep. 2020. [Online]. Available: <https://www.nginx.com/faq/how-often-is-nginx-and-nginx-plus-released/>. [Accessed 28 Nov 2022].

- [32] NGINX, "Nginx.org," NGINX, 19 Oct 2022. [Online]. Available:
<http://nginx.org/en/CHANGES>. [Accessed 1 Dec 2022].
- [33] Videolan, "VLC Releases - VideoLAN," Videolan.org, 2022. [Online]. Available:
<https://www.videolan.org/vlc/releases/>. [Accessed 30 Nov. 2022].
- [34] W. Contributors, "VLC media player," 27 Nov. 2022. [Online]. Available:
https://en.wikipedia.org/wiki/VLC_media_player. [Accessed 1 Dec 2022].
- [35] The Kubernetes Authors, "Deployment-Status," Kubernetes, [Online]. Available:
<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#progress-deadline-seconds>. [Accessed 29 August 2022].
- [36] Kubernetes, "Container Runtimes," Kubernetes, [Online]. Available:
<https://kubernetes.io/docs/setup/production-environment/container-runtimes/>.
[Accessed 17 July 2022].
- [37] The Kubernetes Authors, "Introducing Container Runtime Interface (CRI)," Kubernetes, [Online]. Available: <https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/>. [Accessed 17 July 2022].
- [38] "Open Container Initiative - Open Container Initiative," Opencontainers.org, [Online]. Available: <https://opencontainers.org/>. [Accessed 31 August 2022].
- [39] The Kubernetes Authors, "Nodes," Kubernetes, [Online]. Available:
<https://kubernetes.io/docs/concepts/architecture/nodes/#heartbeats>. [Accessed 17 August 2022].

[40] "GitHub - cri-o/cri-o: Open Container Initiative-based implementation of Kubernetes Container Runtime Interface," 23 Nov. 2022. [Online]. Available: <https://github.com/cri-o/cri-o>. [Accessed 25 November 2022].

[41] D. Inc, "About Docker CE," 04 June 2020. [Online]. Available: <https://docker-docs.netlify.app/install/>. [Accessed 21 Nov. 2022].