DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, *SWEDEN 2020*

Project-based Multi-tenant Container Registry For Hopsworks

PRADYUMNA KRISHNA KASHYAP

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Project-based Multi-tenant Container Registry For Hopsworks

PRADYUMNA KRISHNA KASHYAP

Master in ICT Innovation - Cloud Computing and Services
Date: September 25, 2020
Supervisor: Dr. Jim Dowling
Examiner: Dr. Amir H. Payberah
School of Electrical Engineering and Computer Science
Host company: Logical Clocks AB
Swedish title: Projekt baserad Multi-tenant Container Registry För Hopsworks

Abstract

There has been a substantial growth in the usage of data in the past decade, cloud technologies and big data platforms have gained popularity as they help in processing such data on a large scale. Hopsworks is such a managed platform for scale out data science. It is an open-source platform for the development and operation of Machine Learning models, available on-premise and as a managed platform in the cloud. As most of these platforms provide data science environments to collate the required libraries to work with, Hopsworks provides users with Anaconda environments.
Hopsworks provides multi-tenancy, ensuring a secure model to manage sensitive data in the shared platform. Most of the Hopsworks features are built around projects, each project includes an Anaconda environment that provides users with a number of libraries capable of processing data. Each project creation triggers a creation of a base Anaconda environment and each added library updates this environment. For an on-premise application, as data science teams are diverse and work towards building repeatable and scalable models, it becomes increasingly important to manage these environments in a central location locally.
The purpose of the thesis is to provide a secure storage for these Anaconda environments. As Hopsworks uses a Kubernetes cluster to serve models, these environments can be containerized and stored on a secure container registry on the Kubernetes Cluster. The provided solution also aims to extend the multi-tenancy feature of Hopsworks onto the hosted local storage. The implementation comprises of two parts; First one, is to host a compatible open source container registry to store the container images on a local Kubernetes cluster with fault tolerance and by avoiding a single point of failure. Second one, is to leverage the multi-tenancy feature in Hopsworks by storing the images on the self sufficient secure registry with project level isolation.

Keywords: Cloud, Big Data, Hopsworks, Data Science, On-premise, Multi-tenancy, Container, Registry, Kubernetes.

Sammanfattning

Det har skett en betydande tillväxt i dataanvändningen under det senaste decenniet, molnteknologier och big data-plattformar har vunnit popularitet eftersom de hjälper till att bearbeta sådan data i stor skala. Hopsworks är en sådan hanterad plattform för att skala ut datavetenskap. Det är en öppen källkodsplattform för utveckling och drift av Machine Learning-modeller, tillgänglig på plats och som en hanterad plattform i molnet. Eftersom de flesta av dessa plattformar tillhandahåller datavetenskapsmiljöer för att samla in de bibliotek som krävs för att arbeta med, ger Hopsworks användare Anaconda-miljöer.
Hopsworks tillhandahåller multi-tenancy, vilket säkerställer en säker modell för att hantera känslig data i den delade plattformen. De flesta av Hopsworks-funktionerna är uppbyggda kring projekt, varje projekt innehåller en Anaconda-miljö som ger användarna ett antal bibliotek som kan bearbeta data. Varje projektskapning utlöser skapandet av en basanacondamiljö och varje tillagt bibliotek uppdaterar denna miljö. För en lokal applikation, eftersom datavetenskapsteam är olika och arbetar för att bygga repeterbara och skalbara modeller, blir det allt viktigare att hantera dessa miljöer på en central plats lokalt. Syftet med avhandlingen är att tillhandahålla en säker lagring för dessa Anaconda-miljöer. Eftersom Hopsworks använder ett Kubernetes-kluster för att betjäna modeller kan dessa miljöer containeriseras och lagras i ett säkert container-register i Kubernetes-klustret. Den medföljande lösningen syftar också till att utvidga Hopsworks-funktionen för flera hyresgäster till det lokala lagrade värdet. Implementeringen består av två delar; Den första är att vara värd för ett kompatibelt register med öppen källkod för att lagra behållaravbildningarna i ett lokalt Kubernetes-kluster med feltolerans och genom att undvika en enda felpunkt. Den andra är att utnyttja multihyresfunktionen i Hopsworks genom att lagra bilderna i det självförsörjande säkra registret med projektnivåisolering.

Keywords: Cloud, Big Data, Hopsworks, On-premise, Multi-tenancy, Container, Registry, Kubernetes.

Acknowledgements

This thesis marks the end of my degree programme as master in ICT innovation with a focus on Cloud computing and services at KTH, Royal Institute of Technology in Stockholm, Sweden and Technische Universität Berlin in Berlin, Germany. I would like to thank KTH and Logical Clocks AB for providing me with an opportunity to learn and conduct this research.

I would like to thank my supervisor at KTH, Jim Dowling (CEO and co-founder of Logical Clocks AB) for providing me with an opportunity to work with Logical Clocks and for all of his support, Antonios Kouzoupis (Software Engineer Logical Clocks AB) for his valuable guidance. I would especially like to thank my supervisor in the company, Theofilos Kakantousis (COO and co-founder of Logical Clocks AB) for providing advice and support all along my master thesis. This thesis work could not have been conducted without his valuable guidance.

Finally, I would like to express special gratitude to my family and friends for motivating me and supporting me throughout this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data is ruling the Globe with its utmost capacity, from anything we use daily to the things we use rarely, everything is measured, analyzed and capacitated by processing certain data. Hence the recent years has witnessed a huge explosion in the availability of data. The traditional database methods and tools with non-distributed centralized storage is not an option to process such large volumes of data due to low speed and efficiency. Not only does the methods and tools prove to be ineffective, sometimes the hardware availability in terms of memory or storage can be lacking, posing bigger challenges to process such data. Big data frameworks were hence introduced to deal with large or complex data that the traditional data-processing application software could not deal with.Big data has been in the intersection of business strategies and data science, it allows to use data as a strategic asset, equipping it with pertinent real-time information [1].

Apache Hadoop is an open source big data framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It's design is very efficient with capabilities to scale up from single servers to thousands of machines, each offering local computation and storage [2]. Although Apache Hadoop provides huge benefits, usage of such frameworks requires much knowledge.

Most data-driven organisations and enterprises need solutions to improve the stages of their data life cycle. Over the past few years, there has been different solutions to improve the data life cycle provided in the form of platforms having big data frameworks as a service. Such platforms hide the underlying complexity of these frameworks and help users, administrators and data scientists work together in unison providing a centralized structure. This thesis describes one such platform called Hopsworks, which aims to provide popu-

lar big data frameworks as a service. It is a managed platform for scale out data science available on-premise and on Amazon Web Services. Hopsworks is also capable of providing multi-tenancy feature to users with various user privileges to access data across different applications [3].

1.1 Motivation

For any data science experiments, there are always new libraries or new versions of these libraries required to build models. Teams, often use intermediate deployments and modular, layered development approaches for data ingest, data cleaning, computation, machine learning and visualization. Hence, they are most likely juggling with a lot of libraries with specific versions. These are managed efficiently by creating data science environments with the required libraries collated in them [4]. In Hopsworks, these environments are project based and provide multi-tenancy for users, but libraries and environments are usually available centrally on cloud platforms [5]. It is acceptable for an application running on a managed cloud service but for on-premise applications, as multiple teams need to have access to these environments, it becomes increasingly important to manage these environments in a central location locally. The focus of this thesis is to determine the best fit storage architecture for these environments.

1.2 Problem Description

Hopsworks comes with two flavours, an on-premise platform with a cluster architecture on top of the existing host machines and as a managed cloud service in the cloud. The Anaconda base environments are created with a trigger on project creation on both platforms and is updated with libraries by importing them into the environments. These environments are currently project specific and is created upon project creation. These environments are not light weight and are currently persisted on the file system in the pipeline work directory on the storage device. An efficient way to store these can be on a cloud storage centrally that simplifies sharing between the team members and across compute nodes or Graphical Processing Unit (GPU) servers. This might lead to latency issues as it needs to be accessed over networks. As a possible solution, these environments can be containerized and storing images of these containers can be an efficient way of storing them. However, this is not feasible as the number of images owned by the team or users can balloon up very quickly

in an organisation. The most efficient way to provide a storage for these container images is using a container registry. With managed services, a container registry is readily available and hence is feasible to use it. Environments can be huge in size and can go up to 20 gigabytes, storing and handling these can become a liability in terms of network latency and storage. An efficient way to store and manage these environments needs to be setup as an alternative for the managed cloud registry services.

1.3 Goals

The goal of this thesis work is to determine the best-fit open source secure container registry, compatible with Hopsworks and design a suitable storage architecture to host this registry on the platform. As Hopsworks aims to maintain the multi-tenancy features, develop integration scripts to extend this features onto the container registry to store the images with project based isolation.

1.4 Methodology

This thesis work although consists of some application integration, overall falls into the category of *System Design* and follows the *Design Science* paradigm as described by Hevner, March, Park and Ram [6]. Hence, the methodology selected for this thesis is *Design Science Research Methodology* as proposed by Ken Peffers et al [7].

This methodology is composed of the following six activities:

1. Problem identification and motivation: Define the main research problem and the motivation behind the research. Atomize the research problem into parts and justify the need for a solution.

2. Define the objectives for a solution: Discuss the qualitative and quantitative objectives of the solution proposed in the previous step and compare them with alternative solutions previously opted in the field.

3. Design and development: Creation of artifacts to solve the defined problem with a proof-of-concept system by, for instance, determining the architecture, infrastructure model or the functionalities proposed in the solution.

4. Demonstration: Demonstrates as to how the artifacts built as part of the solution solves the problem through experimentation or other meaningful methods. As this thesis deals with several theoretical architectures the experimentation for these are limited.

5. Evaluation: Determine the extent to which the system addresses the research problem. This is done through measurements or other suitable means.

6. Communication: Reiterate the problem, the systems design and the conclusions to the relevant audience. Provides scope for future work with respect to the existing system.

1.5 Ethics and Sustainability

Ethical issues arise with collection of data for the purpose of processing, and most of these issues are centered around privacy. This thesis bases it's work on the Hopsworks platform that provides users a General Data Protection Regulation (GDPR) compliant security model to manage sensitive data. Further, this thesis does not possess any direct or indirect ethical concerns and complies with the code of ethics of the host company and with the IEEE code of ethics [8]. No sensitive personal data is being obtained or used in the process. All data produced during implementation and testing phases do not contain any sensitive information that poses any security risks to individuals or organisations. Adequate references have been provided to any previous work utilized in this work.

This thesis does not have a direct impact on sustainability, but usage of microservice architecture and using Kubernetes to orchestrate these architectures provides proven ways to improve the performance of software systems. This can indirectly result in cost efficient and energy efficient software systems. This platform also provides a multi-tenant self security model, which allows efficient ways to use shared data leading to less redundancy of data in turn reducing storage. An addition to this platform in-terms of an efficient storage also providing multi-tenancy also leads to reduction of storage indirectly leads to less consumption of energy, reduction in electronic wastes and also reduces the carbon footprint.

1.6 Delimitation

The scope of this thesis is limited to building compatible architectures for Hopsworks on-premise platform to store the data science environments. The current architectural design does not support the managed cloud platform. The multi-tenancy feature integration is based on specific versions of the API used in developing the integration scripts and this thesis does not intend to provide support for future versions of these applications.

Hopsworks currently uses Docker version 19.03.8 and Kubernetes version v1.18.0, any adverse effects due to up-gradations are out of the scope of this thesis.

1.7 Outline

This thesis is organized inline with the methodology opted for the research. It comprises of five chapters

- Chapter 1 - This chapter looks into the problem description and motivation behind the research with an introduction to the topic. It also comprises of ethics and sustainability aspects of the thesis and concludes with the delimitation of the thesis.
- Chapter 2 - This chapter aims to provide the necessary theoretical background for the implementation.
- Chapter 3 - This chapter describes the implementation choices made, overview of the system architecture, infrastructure designs implemented as part of the thesis , integration methods and concludes with a description of the test environment setup to obtain the results of the implemented solutions.
- Chapter 4 - This chapter discusses the outcome and evaluates the value the solution provides to the system. It also provides an aspect of future work that could be performed around this work.
- Chapter 5 - This chapter reiterates the problem, the provided solution.

Chapter 2

Theoretical Background

This chapter aims to provide the necessary background towards data science, big data, virtualization and Kubernetes orchestration followed by a brief introduction to the Hopsworks platform and the necessary topics with regards to the Hopsworks platform that are relevant to this thesis.

2.1 Data Science

There are a plethora of definitions to data science due to its immense popularity. A few examples for the definitions are as follows.

> *Data Science is concerned with analyzing data and extracting useful knowledge from it.Building predictive models is usually the most important activity for a Data Scientist [9].*

> *Data science is an emerging discipline that draws upon knowledge in statistical methodology and computer science to create impactful predictions and insights for a wide range of traditional scholarly fields [10].*

But these definitions are contextual and reflect only some of the many use cases or contexts of data science as a discipline. But one of the better definitions basing it from over a hundred and fifty data science use cases and over three years of research was from Michael L. Brodie.

> *Data Science is a body of principles and techniques for applying data analytic methods to data at scale, including volume, velocity,*

> *and variety, to accelerate the investigation of phenomena represented by the data, by acquiring data, preparing and integrating it, possibly integrated with existing data, to discover correlations in the data, with measures of likelihood and within error bounds. Results are interpreted with respect to some predefined (theoretical, deductive, top-down) or emergent (fact-based,inductive, bottom-up) specification of the properties of the phenomena being investigated [11].*

Although the definition is quite compact and resourceful, like the other definitions, it will also mature over the next decade to be a better and distinct paradigm. The goal of data science, irrespective of its definition is focused on extracting valuable insights from the available data. For this, one should understand the key data assets that can be turned into pipelines that can produce maintainable tools and solutions. It is generally a team discipline, comprising of data scientists, who overlook the core functionality of projects, where as the analysts have a key role in transforming data into analysis and further into production grade values. The team also consists of Data engineers who help build pipelines to enrich these data sets and make available to the entire team [12].

Traditionally, data science project deployments involve complex and time-consuming tasks such as maintaining the libraries or packages and their dependencies or even specific versions of them. This could deviate the teams from its main objective to explore and analyze data. Environments provide an efficient way to handle these aspects, by providing package managers and other libraries built-in with their base versions. These are easy to manage and provides a modular approach to have intermediate deployments and layered development approaches for data ingest, data cleaning, computations and visualisation.

Anaconda distribution is one of the most widely used distributions across data science applications. It comes with over three hundred libraries that include Math, scientific analysis and visualization libraries built-in to its base environment. It can also harbour extra libraries, layer-wise into the environment on-demand. The distribution includes Conda, which is a package and environment manager that helps maintain multiple such environments. A basic scale-out architecture for an Anaconda environment is denoted in the Figure 2.1. The environment is hosted on either a bare-metal or a cloud-based cluster on a number of compute nodes with a distributed file system as part of the distributed storage. The other layers can be managed using the Anaconda environment [13].

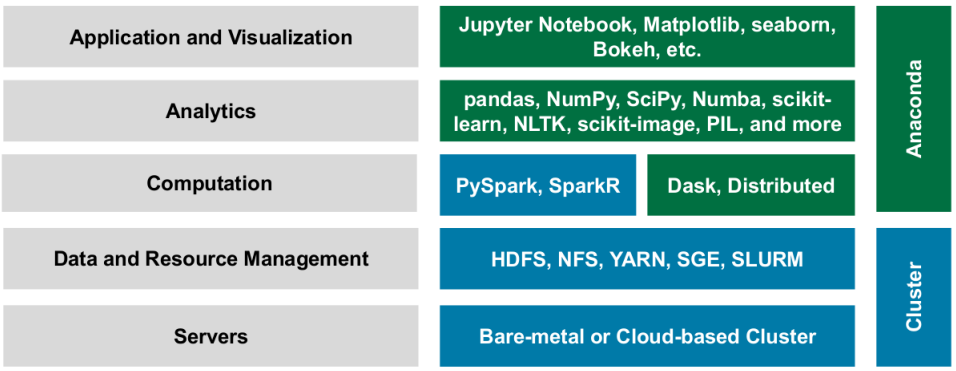


Figure 2.1: Anaconda scaled-out architecture from the Continuum Analytics 2016 [13]

2.2 Big Data

The Gartner's definition,circa 2001 of big data is still the go-to definition to describe dig data.

> *Big data is data that contains greater variety arriving in increasing volumes and with ever-higher velocity. This is known as the three Vs [14].*

Formally, big data is defined from 3Vs to 4Vs, with the forth referring to the veracity of data.

- Variety: It refers to the different types of data available. Traditionally, structured data was considered to be conventional as it would fit neatly into a relational database. With the rise of big data came along unstructured data or even semi-structured data in the form of texts, audio or videos, which additionally requires some metadata.
- Volume: Simply describes the amount of data that is relevant. It can be in terms of terabytes or even tens of petabytes of data for some data-driven organisations.
- Velocity: It denotes the rate at which the data is generated or received or perhaps even acted upon.
- Veracity: Accountability is an important aspect of data accumulation and this signifies the availability and aims to provide meaningful governance of data [15].

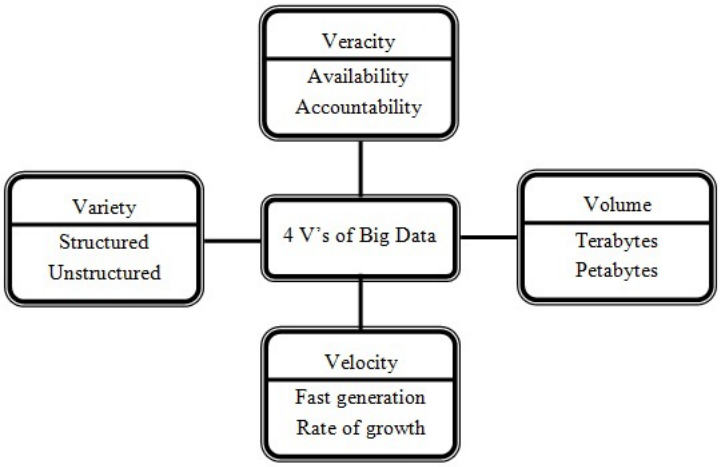


Figure 2.2: Characteristics of big data [15]

2.2.1 Challenges Of Big Data

The challenges of big data can be broadly classified into three main categories based on the data life-cycle.

1. Data challenges constitutes the problems that arise due to the 4Vs described in the definition. Volume can be problematic in terms of conventional storage, velocity in terms of streaming data, might require additional infrastructure set-up, veracity leading to complexity with respect to lack of accuracy or quality and finally, multiplicity of varieties of data leads to complexity in handling.

2. Processing challenges include collection, modification and representation of data, that is acceptable to visualize and analyze data in a way that is useful to the user.

3. Data management challenges refer to secured collection and storage of data. Focus revolves around privacy, ownership and governance issues that exist with data, these are usually policy based or even rules defined to govern the management of data on a state or even an International level [16].

2.2.2 Apache Hadoop

It is quite clear from the discussion above that traditional processing and storage systems are not sufficient to handle big data. Distributed processing is the

Figure 2.3: Classification of big data challenges [16]

key to solve these challenges at hand with regards to big data and Hadoop is an open-source framework which allows distributed processing of such large data sets across clusters of computers using simple programming models [2].The below points provide insights as to how Apache Hadoop mitigates some of these challenges.

- Hadoop is built to run on distributed systems or a cluster of computers. A simple problem could be limited Central Processing Unit (CPU) resources on a single compute node to perform a task. Solving this with a simple approach is to add more resources and Hadoop from the get-go is designed to run on a bunch of machines and utilize its resources collectively.

- Essentially, addition of resources to scale vertically with more powerful hardware is quite expensive. Hadoop provides more storage and compute power by adding more compute nodes to its cluster allowing to scale horizontally eliminating the need for expensive hardware.

- It also handles a variety of data from semi-structured to unstructured data as it does not enforce a schema on the data it stores.

- Finally, Hadoop provides an efficient solution to manage storage and computing on the same set of resources [17].

By addressing some important challenges, Apache Hadoop has now become the defacto standard for data-driven organization to have as part of their big data solutions.

2.2.3 Hops

Hops, short for Hadoop Open Platform-as-a-service is a next-generation distribution of Apache Hadoop that is highly available, scalable and provides features to customize metadata. Hops has mainly two sub projects, Hops Filesystem (HopsFS) and Hops YARN. We will focus on HopsFS as Hops YARN is not relevant to this thesis. Hops consists of a heavily adapted implementation of Hadoop Filesystem (HDFS) based on Apache Hadoop version 2.8 called HopsFS. It is a hierarchical file-system that manages the metadata using commodity databases.

Figure 2.3 denotes the architecture of HopsFS, It mainly consists of a Network Database (NDB) cluster and a Data Access Layer Application Programming Interface (DAL-API) constituting the metadata management layer with a block storage used for storing the data. It replaces the active-standby replication architecture in HDFS by providing stateless redundant Name-Nodes backed by an in-memory distributed database in the form of NDB. The DAL-API provides an abstraction over the storage and implements a leader election protocol using the database. The HopsFS / HDFS clients interact with the DAL-API and the Data Nodes for fetching or storing the data into the Data Nodes with the help of metadata stored in the NDB cluster.

This architecture allows HopsFS to omit the quorum journal nodes, the snapshot server and the Zookeeper services that are additionally required in Apache HDFS allowing it to store small files very efficiently making it the World's most scalable HDFS-compatible distributed file system [18]. It can be used as a standalone file system for various applications requiring high performance storage with smaller files in use. It is mainly used for storage of metadata and small files on the Hopsworks platform acting as a drop-in replacement for HDFS in the Hopsworks platform.

2.2.4 Ceph

Ceph is an open-source software storage platform that implements on a single distributed computer cluster. It readily provides three different interfaces for

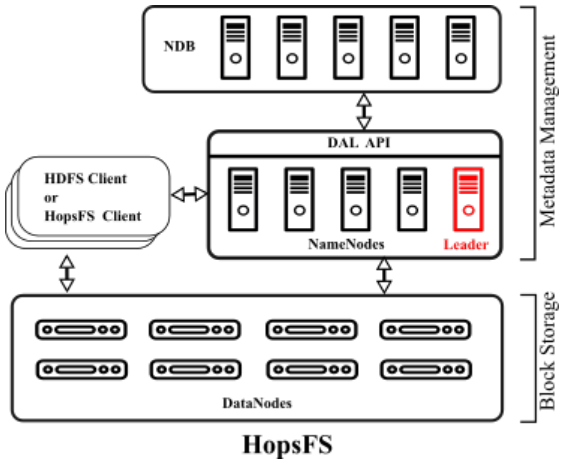


Figure 2.4: HopsFS Architeture [18]

storage, block storage, object storage and file-level storage. It aims to distribute operations without a single point of failure, scalable and freely available [19]. Ceph clusters usually involve setting up Ceph nodes, a network for their communication and the storage cluster. The Ceph storage cluster essentially needs a Ceph monitor, Ceph manager and Ceph object storage daemon (OSD). Additionally, a metadata server is required if we are running a Ceph file-system.

- Monitors: A Ceph monitor also referred to as 'ceph-mon' maintains maps of the cluster state, this includes the monitor map, manager map, OSD map and the Controlled Replication Under Scalable Hashing (CRUSH) map. CRUSH is an algorithm used by Ceph to place data on specific placement groups on then cluster. These maps stored by the monitors are vital for the daemons to coordinate with each other. These are also responsible to manage authentication between the daemons and the clients.

- Managers: A Ceph manager also referred to as 'ceph-mgr' is responsible to keep a track of the metrics and the state of the entire cluster. The state includes, storage utilization, performance metrics and the system load at a given point of time. These managers are also responsible to run Python based modules on them to provide web based Ceph dashboard and Representational State Transfer (REST) API for the cluster.

- Ceph OSDs: A Ceph OSD referred to as the 'ceph-osd' stores and handles the data replication, recovery, rebalancing and proovides the monitors some monitoring information.

- MDSs: A Ceph metadata server is only used in combination with a Ceph file-system implementation. It is used to store the metadata on behalf of the Ceph file-system. it allows a Portable Operating System Interface (POSIX) file-system user to perform basic commands like ls, find etc. on the cluster.

These Ceph clusters can be made highly available by making sure that a minimum number of instances of each of these components run on the system. These highly available clusters make use of CRUSH algorithm to scale, rebalance and recover dynamically based on requirement [20].

2.3 Virtualization

Virtualization is a process of running a virtual instance of a computer system with a layer of abstraction from the existing hardware. This abstraction helps host multiple operating systems on the same hardware. For any applications running on top these operating systems, it appears to the application as if they have their own resources with a dedicated machine [21].

2.3.1 Hypervisor

A hypervisor is a program that helps create and run virtual machines on the existing hardware. There are two types of hypervisors

- Type 1: Native or bare-metal hypervisors run guest virtual machines directly on the system's hardware. It behaves essentially like an operating system.
- Type 2: Hosted hypervisors are much similar to traditional applications that run on top of a host operating system, that can be started or stopped normally.

Figure 2.5 denotes a layer-wise comparison of the two types of hypervisors. The VM that runs on the bare-metal can utilize the computational resources better as they do not depend on the operating system for the same. This makes them more useful for enterprises in the Information Technology (IT) field. On the other hand, hosted hypervisors are better for personalizing as they allow multiple VMs to run on the same host operating system with the help of a virtualization software.

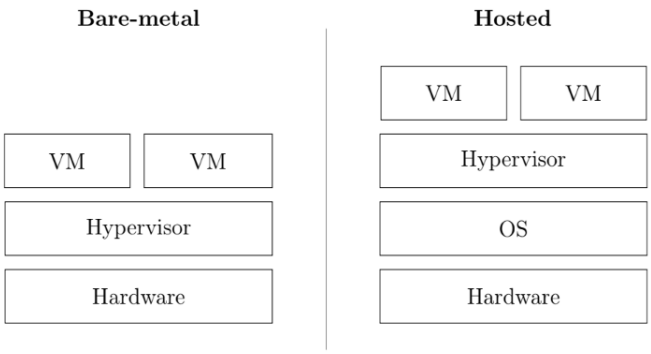


Figure 2.5: Hypervisor Bare-metal vs Hosted [22]

2.3.2 Virtual Machines

Virtual machines are emulated computer systems with operating systems that run on top of a host machine or another system. These virtual machines make use of hypervisors to access the resources or hardware to run on top of them. Although they can have any number resources attached to them at their disposal, they have very limited access to the host machines CPU and memory as they are sand boxed from the rest of the system. They are generally responsible to perform a specific set of tasks on the host machine that are risky to be performed on an actual physical machine [21].

2.3.3 Containerization

Containerization, also known as *OS-level virtualization* is a technique where the OS kernel supports multiple isolated user-space environments [23]. These isolated environments are called *containers*. The de-facto containerization software used in the modern day is Docker, which has its own containerization engine called *libcontainer*, that is used to create containers [24]. Also, there exists a standard format for these containers in order to maintain a generic format, the Open Container Initiative (OCI), which was formed by Docker and others to provide a standard specification on important features such as the container run time [25].

2.4 Docker

Docker is an open platform for development, shipment and running applications in an isolated environment. It helps developers easily maintain infras-

tructure the same way an application is maintained. Hence deployment of applications into production systems can be done with significantly less time. The Docker platform allows application to be loosely coupled in an isolated environment also called *containers*. These containers are very light-weight as they do not need a hypervisor, meaning they can run with the help of the host machine kernel and resources. This helps increase the number of containerized applications that can be run on a given hardware combination.

2.4.1 Docker Engine

The Docker engine is basically a client-server application which supports a set of tasks, workflows that are required to build, ship and run some containerized applications. The Docker engine consists of a set of components as shown in the figure 2.6.

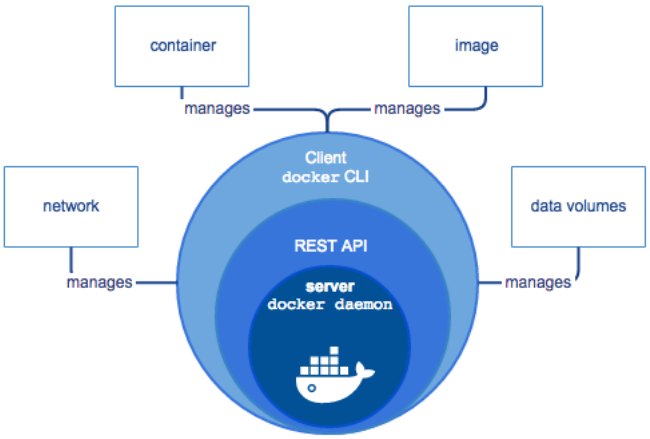


Figure 2.6: Docker engine components [26]

- The server itself is a process also called as the daemon process which is responsible for creation and maintenance of Docker objects, like images, containers, networks and volumes.
- The REST API is a means of communication medium for the programs which are running externally to interact with the Docker daemon to instruct some tasks to be done.

- The command Line Interface (CLI) acts as a client which uses the REST API to control and interact with the Docker daemon from client machines through some scripts or direct CLI commands [26].

2.4.2 Docker Architecture

Docker is built to have a client-server architecture. The client is responsible to interact with the Docker daemon which sits on a Docker host, to perform some tasks. As in any other client-server application, the client can either run on the same system or on a remote system. The communication between the client and the daemon is done using a REST API, over Uniplexed Information and Computing System (UNIX) sockets or some network interfaces.

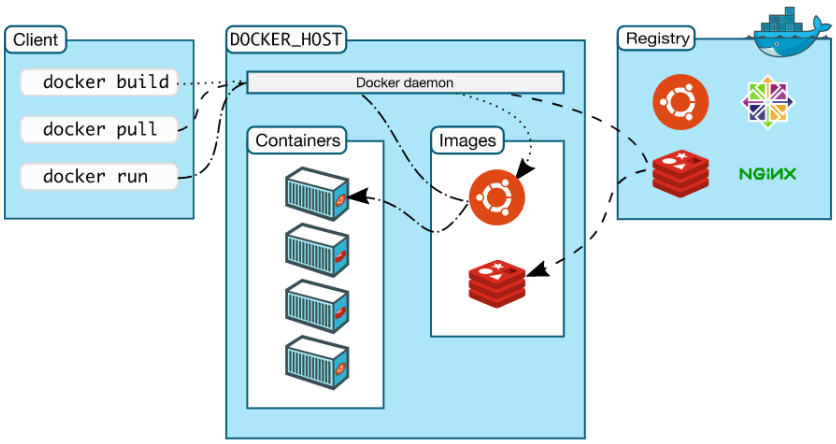


Figure 2.7: Docker architecture [26]

- Docker daemon: The Docker daemon or referred to as *dockerd* responds to Docker API requests and is responsible to maintain the Docker objects. It is also possible to communicate with other daemons using a Docker daemon to manage some Docker services.

- Docker client: The Docker client also referred to as just *docker* is a way for the users to communicate with Docker. When any Docker commands are sent from a client, the Docker daemon responds to them by carrying out the mentioned functionalities. These commands are usually communicated to the daemon through the Docker API. Also these clients can communicate with one or more daemons that are either present on the same server or remotely located.

- Docker registries: The *registry* is the prime location for storing the Docker images. These registries can be public or private self hosted registries. The public registries are open for anyone to use and the most used Docker registry is the Docker Hub. Docker is configured to always look for images on the Docker Hub by default on the use of Docker commands. Usually when the Docker daemon is asked to run a container in the environment, it primarily looks for the image on the local repository, if not present, then looks for the image of the specified name on Docker Hub. Docker can also be configured to retrieve or store images on a private registry.

- Docker objects: When the command Docker is used, it usually refers to creating an object. Objects can be of different types, images, containers, networks, volumes or other objects. The two objects most commonly used are the images and containers.

  - Images are read-only templates with instructions for Docker to create a container in accordance to the template. An image can be based on another image with some add-on or customization. In this case, the image used to build on is called the base image. Usually, base images are made up of operating system images with some custom environment or configuration set up for the application to run on inside the container. These images can be either be self built or used from a public repository. To build custom images from scratch, a *Dockerfile* needs to be created with simple syntax defining steps to create the image and run it. The important feature that makes this efficient is that, every time the Dockerfile is altered and re built, only the layers which are changed in the file are re-built and others are re-used. This makes these images very light-weight, small and fast to use and deploy.

  - Container is a runnable instance of an image. The containers are usually defined by its image or some user configurations that are passed while creating the containers. By default, all containers are isolated from other containers and the host machines, but it is configurable to define how isolated a network or subsystems should be from other containers. These containers can be created, started, stopped, moved or deleted with the Docker API using the Docker CLI commands through the client. These containers usually do not have persistent storage built but that can be configured by attaching

some storage to these containers which makes them save the state even after completing the container life-cycle.

Namespaces and Control Groups

Docker uses a technology called *namespaces* to provide such isolation for the containers, each container has a set of namespaces for different aspects of the container and each aspect is further bound to its namespace. For example, a process id or *pid* has its own namespace called the pid namespace and is completely seperated from the network interfaces, which in turn have their own namespace.

Control groups or *cgroups* on Linux is used by the Docker engine to restrict the application to a set of resources. These cgroups are used to share the available resources on the hardware among the set of containers running on them. For example, Docker engines can be used to limit the memory that a specific container can use from the available memory [26].

2.5 Kubernetes

Kubernetes is a container orchestration solution which helps in deploying, scaling and managing containers across multiple physical and virtual machines. The first unified container management system was developed at Google and was internally called *Borg*. It was primarily used to manage long-running services and batch jobs on shared machines primarily to increase the resource utilization and thereby reducing costs. Omega was an off-spring of Borg which was developed once again as an internal project at Google but with improved architecture and by storing the state of the cluster on a centralized transaction oriented store.Then came the third container management system called Kubernetes which on contrary to the first two was made as an open source product with increasing interests for developers in Linux systems [27].

2.5.1 Kubernetes Architecture

Kubernetes uses the master-slave architecture where the master and slaves are hosts in a selected network. When we deploy Kubernetes, we get a cluster, the cluster consists of a set of worker machines called *nodes*. These nodes are responsible for running containerized applications on them. Every cluster consists a minimum of one node. The node responsible to manage the worker nodes in the cluster earlier referred to as the Master node is now simply called

a control plane. Any node can be a control plane in the cluster, but as default, the node on which the cluster is deployed becomes the control plane unless specified otherwise.

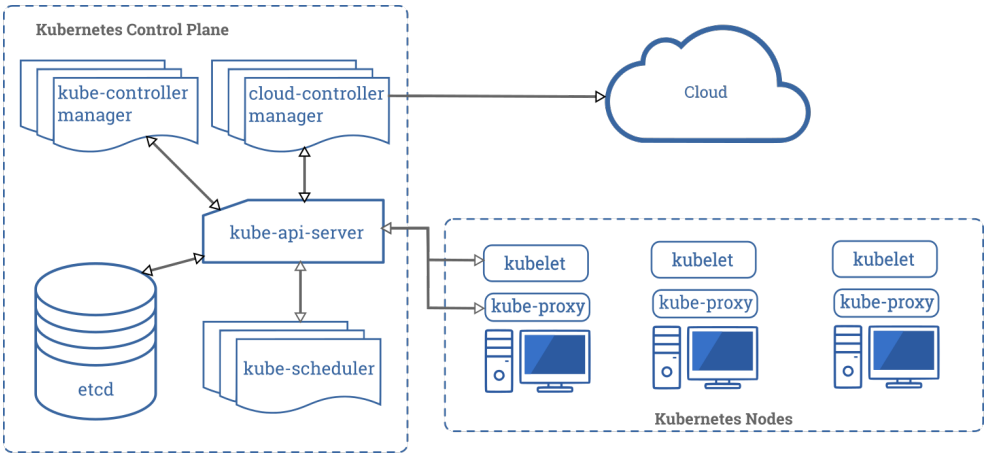


Figure 2.8: Kubernetes Architecture [28]

Control Plane and it's Components

The control plane is responsible for most of the global decisions taken in the Kubernetes cluster. The control plane is responsible for main tasks like distribution and scheduling of application containers on the nodes in the cluster. They are also responsible for maintaining the application state, scaling or rolling out updates of the applications. The control planes schedule containers inside units called *pods* on the nodes available in the cluster.

The Pod constitutes the atomic unit of a Kubernetes cluster which is capable of running one or more containers in them and communicate with the control plane through the API. These pods ideally do not run on the same node as the control plane, exceptions being test clusters having single node in the cluster.

- Kube-apiserver: The API server acts as the communication gateway for the other components of the cluster and external applications to communicate with the cluster. It is responsible to expose the Kubernetes API and acts as the front-end to the Kubernetes Control plane. It also deals with hosting API's for applications deployed inside the cluster to be reachable for the external applications. Kube-apiserver is also designed to scale and can run multiple instances as required to balance the traffic between these instances.

- Etcd: It is a consistent and highly available key-value store that is responsible to store all of the cluster data. It generally stores all data related to nodes, containers and other components which are part of the cluster, hence on any highly available cluster, it is replicated to have a backup on a different node.
- Kube-scheduler: It maintains the scheduling of pods and other resources in the cluster. When a pod is created, it generally is not assigned to a node, the kube-scheduler checks for such newly created pods and selects a node for them to run on. Scheduling of resources are done based on several factors like policies, hardware and software requirements of the application being deployed in the pod, affinity or anti-affinity to a certain node, inter-workload interference and deadlines.
- Kube-controller-manager: It is responsible to run all the controller processes. Controllers are separate processes but are compiled as part of a single binary and run as a single process.
  - Node controller: Takes care of the nodes health and responds when a node goes down.
  - Replication controller: Makes sure the replication controller objects matches the relevant pods for each replication.
  - Endpoints controller: Responsible to join the services with the pods.
  - Service account and token controllers: It creates default tokens and accounts to access the namespaces created.
- Cloud-controller-manager: It is similar to a kube-controller manager but is used to embed the cloud specific control logic. It is usually present in a cluster run on cloud and a cluster run on own premises or a test cluster will not possess it. Like the kube-controller-manager, cloud-controller-manager also logically has a number of processes but is compiled into a single process.
  - Node controller: Takes care of the node's deletion in the cloud post stopping the node based on instructions.
  - Route controller: Makes sure to setup routes in the cloud infrastructure.
  - Service controller: It is responsible to create, update or delete the load balancers provided by the cloud provider.

- Domain Name System (DNS): Although it is part of the addons in a kubernetes cluster, most clusters should possess a cluster DNS. It is like any other DNS server but is responsible to serve DNS records for Kubernetes services [28].
- Kubectl: The kubectl or Kubernetes control is a command line tool that lets users control the Kubernetes cluster. It uses the Kubernetes configuration file in the host system to set up it's configuration [29].

2.5.2 Persistent Volumes and Claims

Kubernetes clusters have a distinct storage mechanism to persist data of the applications that run inside the pods. By default, all data is lost once the pod is killed or data only persistent until the life-cycle of the pods last. But for most of the modern day applications, this is a bad trait to have. Hence, Kubernetes persistent volumes provide an abstract layer that makes sure the applications running on the cluster do not have to take care of the storage or persistence, instead the Kubernetes administrator with the help of API resources, persistent volumes and persistent volume claims, takes care of it. The persistent volume is a piece of storage that has been provisioned by the administrator or dynamically provided using a storage class. They are volume plugins that can be attached to pods for the applications to make use of but have life-cycle independent of the pod using the volume. These volumes are API's that are capable of capturing the details of the implementation of storage, be it a network file system, Internet Small Computer System Interface (ISCSI) or even a cloud provided storage system like an S3, Block devices or disks [30].

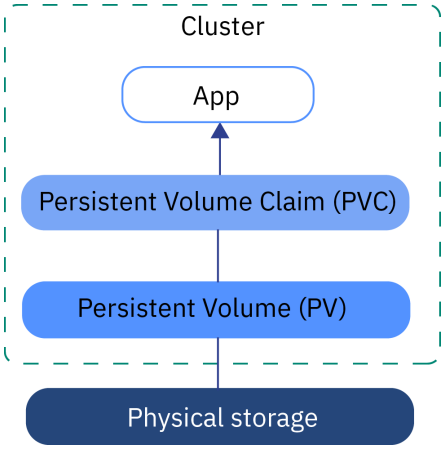


Figure 2.9: Kubernetes persistent volumes and claims [31]

The persistent volume claims are requests that link the applications running inside the pods with the available persistent volumes. As these storage classes and volumes are provisioned by administrators, these are abstracted for users. Users can request specific levels of resources in the form of CPU and memory through pods and specific size of storage or access modes using these claims. when a claim is specified by the user for an application, Kubernetes makes sure to search a relevant provisioned persistent volume or Storage class available in the cluster and provisions the same dynamically when the pods are created in the cluster. Figure 2.9 denotes a simple structure portraying an application claiming a persistent volume which in turn consumes storage from the physical storage to provision the same to the application inside the cluster [31].

2.5.3 Helm

Helm is a package manager for Kubernetes. Helm has a client only architecture with the client itself called *helm*. Figure 2.10 indicates the client only architecture in Helm, where it directly interacts with Kubernetes API server to create, modify or delete the Kubernetes resources [32].

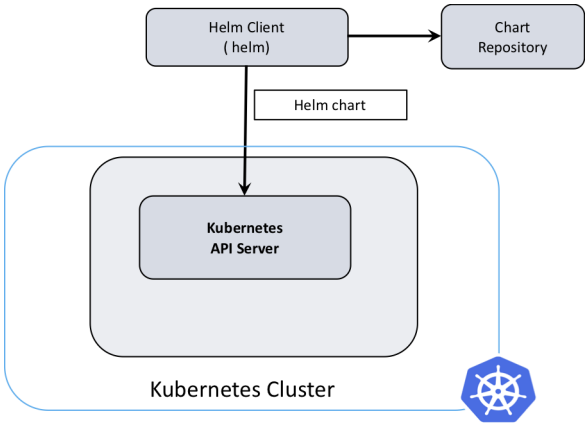


Figure 2.10: Helm Architecture [32]

Helm uses packaging formats called *charts*. These are a collection of files describing a related Kubernetes resource. A simple chart is capable of doing things like setting up a memory-cached pod to a complex web application deployment. These charts have a public repository where they are stored for usage of the public users and by default, Helm looks for the chart defined in the commands in the local repository and if not found, then points towards the Helm chart repository. These charts can be created as files and laid out in the

form of a particular directory tree, further packaged into versioned archives to be deployed [33].

2.6 Hopsworks

Hopsworks is a managed platform for scale-out data science, with support for GPU integration and big data frameworks. It manages to unite a number of open-source analytic and machine learning frameworks with a unified REST API. It is a platform which allows design and operation of data analytic and machine learning applications.

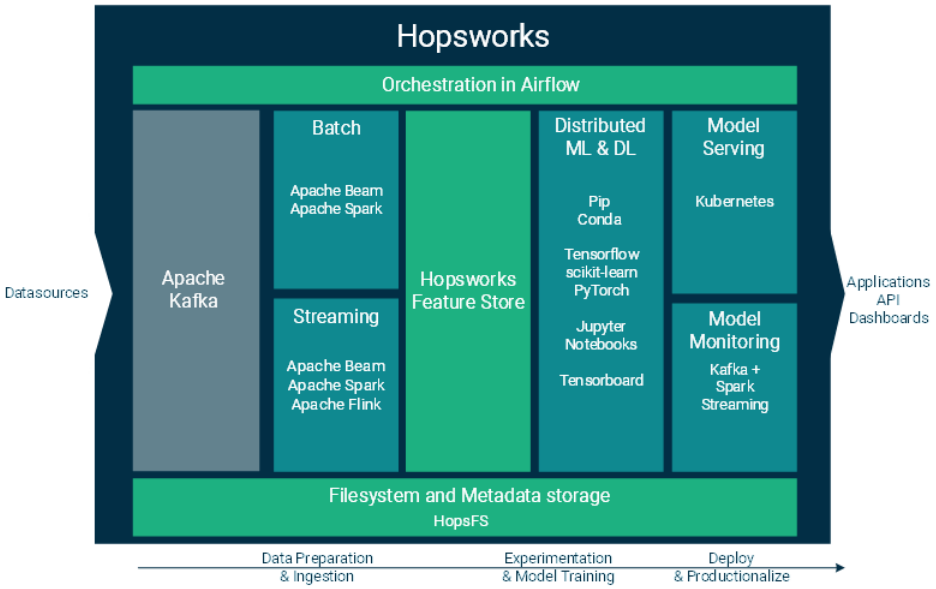


Figure 2.11: Hopsworks Architecture [34]

Figure 2.11 denotes the architecture of Hopsworks platform, it consists of the data preparation and ingestion layer that provide features with support for batch and streaming data in terms of Apache Beam, Apache Spark and Apache Flink. As part of experimentation and model training, it provides Jupyter notebooks in Python which can be used to leverage several popular interfaces like Tensorflow, PyTorch and Scikit-learn. For deployment of these models, Hopsworks provides model-serving options on Kubernetes and monitoring with Apache Kafka and Apache Spark. The feature store is a one of a kind, World's first open-source feature store that provides an organised way to store the features of data sets on a central location to ease access. These

workflows can be orchestrated by Airflow while running on HopsFS which provides the file system and a meta data storage layer for the platform [34].

2.6.1 Projects, Users and Datasets

There are mainly three major concepts in the Hopsworks platform, projects, users and data sets. As Hopsworks aims to provide a GDPR complaint security model for managing sensitive data across a shared platform, its security model is built around projects. Figure 2.12 provides an overall structure of projects in Hopsworks with isolation options based on data sets, codes and users. Project contains three major entities, data sets, users and programs. They can also encapsulate sensitive data sets inside of each project which can be prevented from being exported by the users or even cross-linking them with other data in other projects. So, as depicted in the figure 2.12, each project will have isolated data sets, its own programs and associated users with privileges. These can be shared among projects or even exported only upon approval from the data owners. Data owners and data scientists are the predefined roles that is part of the role based access control within the project that is implemented on the Hopsworks platform. Data owners as described previously are primary access holders responsible for the data and to decide upon who can access it, where as the data scientists are the processors of data in the system.

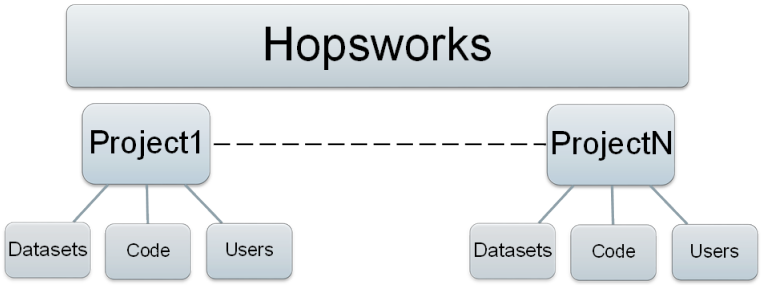


Figure 2.12: Hopsworks project structure [34]

2.6.2 Data Sharing Without Replication

As part of the GDPR compliance, Hopsworks enables its users to share the data among different projects without actually replicating the data into different clusters. Most of the competitors providing similar functionality by creating a whole new cluster for the sensitive data sets by replicating them, but in Hopsworks, it can be shared securely among projects without copying data.

These features can be used for data in the form of Apache Hive databases, Apache Kafka topics or in the form of sub trees in HopsFS.

2.6.3 Security

Hopsworks implements a project based multi-tenancy security model with the use of Transport Layer Security (TLS) certificates in place of Kerberos for user authentication with new certificates generated for use in every project. To realize such a security model, Hopsworks has a dynamic role based access which makes sure the users are not provided with a static global role, but the roles for users can differ dynamically with the project they are working on. This means, a data owner of a project can have a role of a data scientist in another project or vice versa. This dynamic roles ensure a strong and unique multi-tenancy between projects in Hopsworks [34].

2.6.4 Conda Environments

In Hopsworks, each project is equipped with a Conda environment. The user of the project with the right access can install or update libraries to the environment without interfering with the projects. These environments are replicated across all machines of the cluster by recording the command used to update the environment in the database and then sending an update through the Kagent process to get the same updated across nodes in the cluster. This can either be done through the command line or with the user interface. An efficient way to store different versions of these environments with project based isolation is sought for in this thesis [5].

Chapter 3

Implementation

This chapter aims to present the implementations and experimental methods used to finalize on the most feasible frameworks and its integration with the Hopsworks platform. This chapter focuses on implementations of different architectures in the preliminary sections and further discusses integration of these frameworks with the Hopsworks platform. The implementation was performed in order, with a test setup developed to check the feasibility of the open-source registries that were selected to be hosted on the Hopsworks platform. As a result of the tests conducted, Harbor was chosen to be implemented as the container registry on Hopsworks. Specific architecture for the system to be built was then defined providing the overall functionalities of the system. This architecture was then realised by splitting the implementation into two major parts. The first part involved setting up the required storage infrastructure for the registry to store its images and metadata. This involved setting up multiple storage infrastructures that were later evaluated based on tests performed against them. The second part was to integrate the hosted registry with the Hopsworks platform to extend the project based multi-tenancy feature from Hopsworks while storing the images on the registry. This was done using integration scripts automatically triggered from the Hopsworks platform based on the functionality. As part of the integration, different authentication techniques were also implemented to determine the best fit. This integration was then evaluated by setting up tests to check the behaviour of the entire system.

3.1 Container Registry

This section aims to explain the proposed open-source registry options and further discusses the short-comings of these registries against implementations

on a test environment with a simple comparison of their features. Hopsworks itself being an open-source platform, the proposed container registries were also from the open-source community. Trow and Harbor were the two self hosted container registries proposed and both of these were setup on a test environment with a single node Kubernetes cluster before finalizing one among them to be deployed on Hopsworks.

3.1.1 Test Environment Setup

As part of the test environment setup, as Hopsworks uses a Kubernetes cluster for model serving, we try and utilize this setup to host a container registry on top of the Kubernetes cluster. This makes the registry local to the cluster and reduces the latency of having to pull images from public repositories.

To realize this test setup, an instance of Minikube was setup locally with a single node cluster. Minikube runs the cluster inside a virtual machine on the server. Table 3.1 represents the Kubernetes cluster specifications. This setup was built individually for both Trow and Harbor registries to be tested upon.

Instance Type	Minikube (single-node)
CPU	2
Memory	4GiB
Storage	10GiB
Kubernetes Version	v1.18

Table 3.1: Kubernetes cluster specification for test environment

3.1.2 Trow

Trow is an open-source self hosted registry which aims to provide a secure and fast way to distribute the images on a Kubernetes cluster. It can also prevent unauthorized or potentially insecure images from being pulled into the cluster using a deny/allow list [35].

Trow was setup on a Minikube cluster following the standard instructions [1]. As part of the installation, Trow required a volume on Kubernetes to persist the images stored in the registry. No attached storage options were available as part of direct storage in Trow for the alpha release [36]. Although Trow consisted of a front-end and a back-end, both were combined into a single

[1] Trow-setup - https://github.com/ContainerSolutions/trow/blob/master/install/INSTALL.md

executable and the front-end was just capacitated to receive Hypertext Transfer Protocol (HTTP) requests and further communicate with the back-end with information with respect to file handlers. The registry does not come originally with a user interface. Security in Trow consists of signed TLS certificates obtained from the Kubernetes certificate authority and is made available on the hosted machine. Routing in this case is achieved only through editing manually the */etc/hosts* on each of the nodes and further adding the Trow certificates to the appropriate stores on each of the cluster nodes. This is acceptable for a testing environment but not a feasible solution for large cluster in production environments.

3.1.3 Harbor

Harbor is an open-source registry that secures the artifacts with policies and a role-based access control system. It also provides features to test for vulnerabilities in images. It is a Cloud Native Computing Foundation (CNCF) graduated project [37].

For installing harbor on the test Kubernetes cluster, the package manager Helm was installed and the required Helm charts to run the registry was to be fetched to the local chart repository. The installation was done based on the standard instructions provided for Harbor Helm [2]. With a default installation with no changes done to the charts in terms of configuration provides an ingress controller automatically updated in the Kubernetes cluster with an Nginx resource acting as a front-end to capture the HTTP and Hypertext Transfer Protocol Secure (HTTPS) requests. Harbor also provides a user interface for user login and provides role-based access control to monitor or access images based on projects and authorizations with differing access to create, push or pull images to and from the repository can also be granted to users with an administrative access. In terms of security, Harbor provides an option to use existing TLS certificates in the form of Kubernetes secrets to be inserted into the registry or if not provided, generates a certificates to be injected into the cluster in the form of a secret.

3.1.4 Summary

As a result of the basic test environment setup and hosting both the registries on the test cluster, Harbor proved to be a better choice with mature community

[2]Harbor HA - https://goharbor.io/docs/2.0.0/install-config/harbor-ha-helm/

support, high availability of feature selections and enhanced security capabilities. As both registries are built on top of the Docker registry, the performance in terms of pushing or pulling images are very similar. Hence, with a simple comparison of the features it was deemed fit to chose Harbor without extensive tests performed against the registries. Also, with Trow still being an alpha release, Harbor was the right choice to be implemented on the Hopsworks platform.

Feature	Trow	Harbor
Multi-tenancy	No	Yes
User Interface	No	Yes
Self-service Security	Yes	Yes
Chart Repository	No	Yes
Vulnerability Scanning	No	Yes

Table 3.2: Feature comparison Trow vs Harbor

3.2 System Architecture

Hopsworks platform uses Kubernetes architecture to provide users with model serving capabilities. One other major service that runs on the Kubernetes infrastructure are the Jupyter servers which are triggered with the creation of Jupyter Notebooks in the Hopsworks Platform. The proposed architecture aims to utilize the existing Kubernetes infrastructure to provide secure container image storage for Hopsworks.

In the existing architecture, each project inside of Hopsworks that provides users with a unique Conda environment is currently Dockerized into images and stored on public repository as base image with a copy in the local repository. These are pulled to the local repository on need basis, ideally on project creation and provisioned in the Hopsworks platform. Further these images are pushed to a local self hosted Docker registry. To reduce latency and to secure these Conda images, a secure Harbor registry is hosted on the Kubernetes infrastructure. The base image is then pushed into the Harbor registry as a one time process. Henceforth, on trigger of each project creation, the base image is pulled from the Harbor registry and provisioned to the user in the Hopsworks Platform. Further, the users in the Hopsworks platform are allowed to customize their environment by adding custom libraries or packages

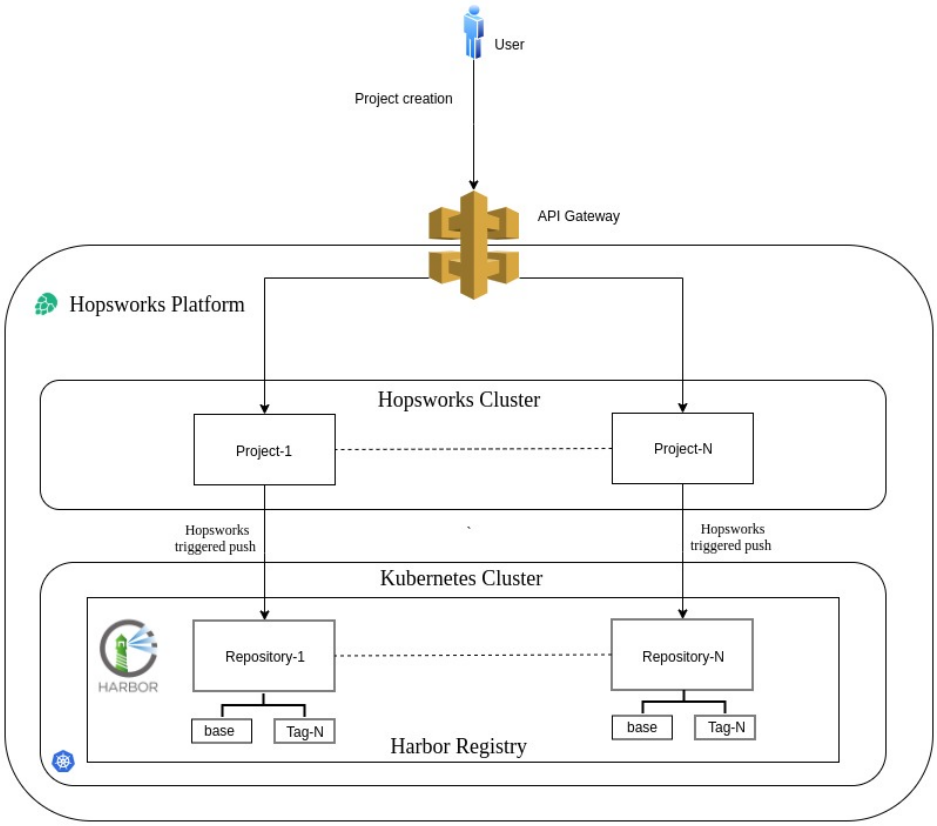


Figure 3.1: System Architecture

into these Conda environments. As part of the proposed architecture, on each update of the Conda environment, a new Docker image is built with the updated Conda environment and pushed to the Harbor registry. As denoted in the Figure 3.1, each time a project is created on the Hopsworks platform by the user through the API gateway, a subsequent project with the same meta data is created in the Harbor registry with a trigger from the Hopsworks system. Upon project creation in the Registry, a base image used to populate the project's Conda environment is pushed to the relevant project in the Registry. Ideally on project creation, the base image is pushed to the Harbor registry with a relevant tag. Upon updating the Conda environment on an existing project, Hopsworks triggers a subsequent Docker image build with the updated environment and pushes the updated Conda image to the Registry with a relevant tag. Also the Harbor user interface is planned to be made available for the Hopsworks users, thereby, allowing the users to access their Conda environment images in their relevant projects with appropriate versioning.

This architecture was implemented in two parts, the first was storage infrastructure. The second, involved the integration of the hosted registry with the Hopsworks platform with automated triggers.

3.3 Infrastructure Design

Harbor registry images need to be persistent and needs to last post the life-cycle of the pod or the registry itself. For this to be achieved, Harbor registry provides a variety of storage options. Most of these storage options involve setting up managed storage infrastructure with Amazon Web Services, Google cloud platform, Microsoft Azure or Swift. These infrastructures are not feasible options as they are managed paid services, hence a basic file-system option available with Harbor was chosen to store the image in the pods, but this does not guarantee persistence.

As part of the file-system storage, Docker manifest files created as part of the images creation, are stored in a particular path in the file-system of the registry pod. In Addition to this, Harbor uses a PostgreSQL to store the meta data of the registry and a Redis storage for caching. Hence these storage units have to be made highly available on the cluster to achieve a highly available architecture.

3.3.1 Persistent Storage with Local File-system

Harbor registry images as discussed in the aforementioned section, stores the images on the Kubernetes pods in the local file-system of the container running inside the pod. To persist this data on the pod, post completion of the pod life-cycle, Kubernetes allows the use of persistent volumes to retain the data populated on the pod.

This infrastructure was designed to leverage the persistent volume option available in Kubernetes to store data on the local file-system. It involves creation of a storage class with dynamic storage allocation capabilities on a provided mount path in the local file-system of the node that is part of the Kubernetes cluster. Persistent volumes created in the Kubernetes clusters denotes the specification of the volume size to be utilized in the storage class, which itself provides the required storage dynamically. Persistent volume claims made with the pods that run the Harbor application to access the persistent volumes were created. These persistent volumes were utilized by the pods with the help of persistent volume claims to store the pod data on the specified path on

the cluster node. Although this provided a simple persistent storage mechanism, the data was stored on a single location on a cluster node, making the data vulnerable as it had no replications. To avoid this, the Harbor registry was mirrored on another node in the cluster, with the exact same local-storage mechanism pointing to its local storage.

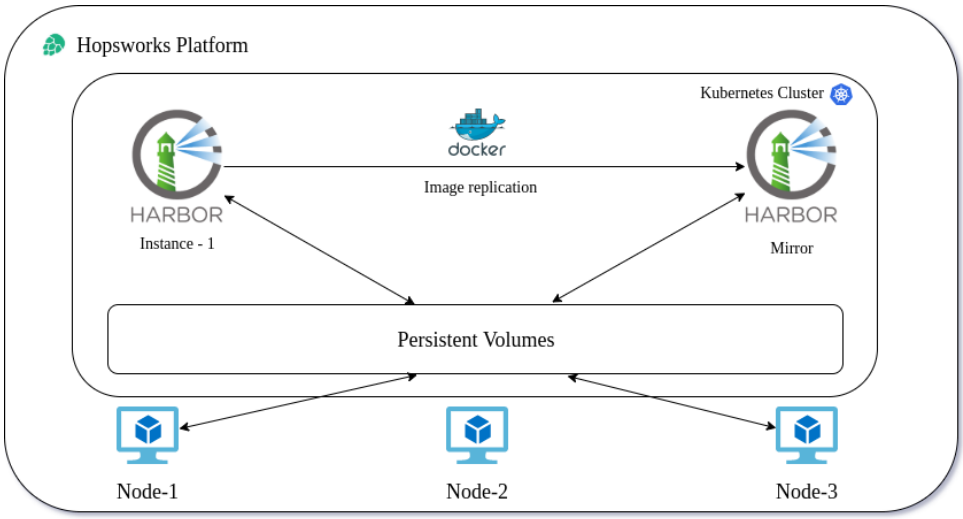


Figure 3.2: Local File-system Storage Architecture

Figure 3.2 represents a simple architecture denoting the Harbor instance-1 utilizing the persistent volume that stores on node-1 and the mirrored Harbor instance utilizing a persistent volume that stores on node-3. This mirroring mechanism ensured duplication of the images and upon failure of a node, the Docker images were still available on the cluster on another node.

3.3.2 Persistent Storage with a Ceph Cluster

This implementation involved setting up a Ceph cluster on the existing Hopsworks platform, for the Harbor registry to make use of in the form of Kubernetes persistent volumes. This made sure that all the data on the pod to be persisted, was stored on a highly available Ceph cluster.

Figure 3.4 represents the overall architecture of the Ceph cluster infrastructure. In this architecture, the Harbor registry is highly available itself as part of the Kubernetes deployment and further the data is also stored on a highly available distributed storage unit in the form of Ceph. This approach makes sure there is no single point of failure for the Harbor registry in the existing system and makes it highly available. There are a number approaches to deploy a

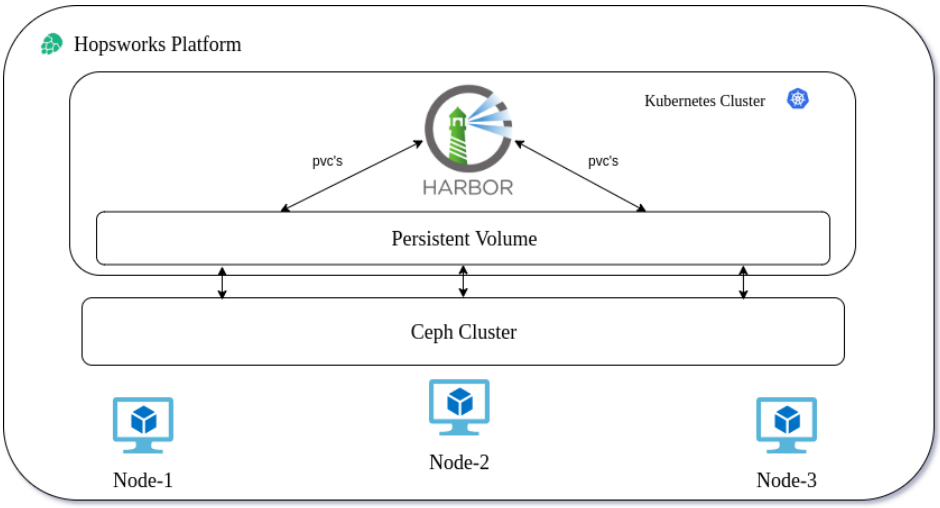


Figure 3.3: Ceph Storage Architecture

Ceph cluster, in this thesis, the state-of-the-art Kubernetes operator Rook was used to deploy the Ceph cluster on the Hopsworks system.

Rook Operator

Rook is an open-source cloud-native storage operator for Kubernetes. It is a self sufficient, self-managing, self-scaling and self-healing storage service designed for Kubernetes. It also supports a number of other distributed storage deployments on the Kubernetes cluster apart from the Ceph storage cluster, but Ceph is the most stable of them all. Rook uses the power of Kubernetes to deliver its services via the Kubernetes operator for each of its storage provider[38]. The architecture of rook heavily leverages the Kubernetes approach to deploy the Ceph storage cluster on it.

Figure 3.4 portrays the role Rook operator plays in integrating the Ceph storage with the Kubernetes Cluster. With a Kubernetes cluster having Ceph running on it, makes sure that the Kubernetes applications can mount block devices or file-systems managed by rook into them. Rook is a simple container that has everything needed to bootstrap and monitor a Ceph storage cluster. The operator itself runs on Kubernetes as a pod and deploys the components of the Ceph cluster on Kubernetes. It creates the monitors, OSDs and managers on the cluster in the form of pods and further monitors the components to provide the storage. It makes sure that the Ceph OSD daemons provide a Reliable Autonomic Distributed Object Store (RADOS) storage as well as manage other daemons. The Rook operator also manages the overall Custom Resource

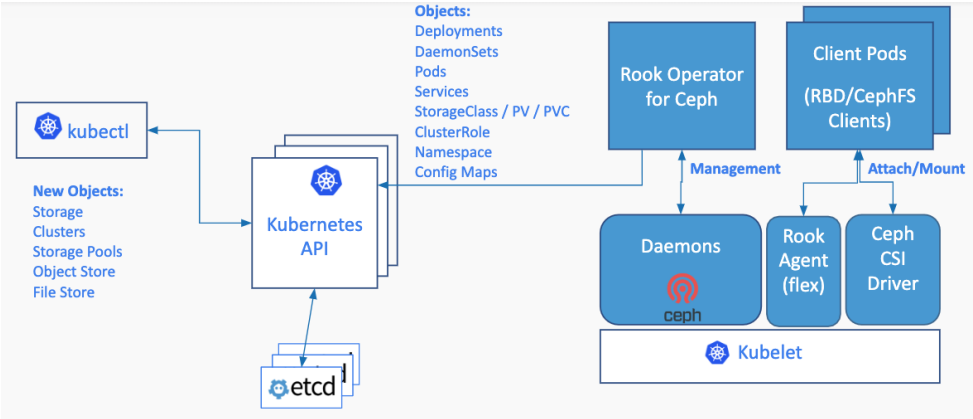


Figure 3.4: Ceph Storage Architecture with Rook Operator [39]

Definitions (CRDs) for pools, objects, stores and file-systems by creating them and other necessary artifacts to run these services. It ensures the cluster health, improvises on the cluster expansion and shrinking based on the requirements and watches the desired state change through the API service continuously and applies those changes. The operator also configures a Ceph-Container Storage Interface (Ceph-CSI) driver automatically to mount the storage onto the Kubernetes pods.

Rook basically provided a simpler user experience for the admin to create and manage the physical resources, at the same time provide configuration options to perform advanced configurations changes. [39].

This operator was made use of to implement the Ceph cluster on Kubernetes for the Harbor repository to further leverage as a storage system. Block storage approach was set up with the Ceph cluster and the Ceph-CSI driver automatically configured to be used with Rook was utilized to mount the storage onto the pods. Necessary persistent volume claims, persistent volumes and storage classes were deployed in the form of additional CRDs on the Kubernetes cluster to make use of the available Ceph cluster.

3.3.3 Testing scenarios for the implemented Infrastructures

The infrastructures described have different mechanisms for storage. Performances vary with type of storage mechanisms. The tests conducted were intended to determine these performance differences. The test were performed on a Kubernetes cluster running on the Google cloud platform. The integrated Google Kubernetes Engine (GKE) was used to quickly deploy the clusters on

the go and the above infrastructures were setup on them. Further, Harbor registries with the exact same configuration and version was hosted on each of these Google Kubernetes clusters to test the performances of the Registry on different storage infrastructures. The below table represents the specifications of the Google Kubernetes cluster.

Cluster Type	v1.15.12-gke.2 (default)
Instance Type	Ubuntu
vCPU	2 per node
Memory	7.5GiB per node
Storage	100GiB per node
Node Count	3

Table 3.3: Kubernetes cluster specification for test environment on GKE

A 3-node default GKE cluster was hosted which made use of the latest v1.18 Kubernetes version, 2 vCPU's per node and 100 GiB storage per node were provisioned. Ubuntu was used as the instance type for the node pool as the default Container-optimized Operating System (COS) does not come with the Rados Block Device (RBD) module which was required for the Rook operator[3]. The tests were performed by hosting a Harbor registry having each of the storage infrastructures to store the images and metadata on them. An iterative approach was used to build the cluster and test each infrastructure one at a time, to make sure the performances were measured accurately.

The test script was developed in Python, It was designed to measure the performance metrics of the container registry.The overall criteria considered for the tests were the push and pull performances of the Harbor Registry on different storage infrastructures. The values in the script that could be customized using parameters were iterations, concurrency and repository address. Iterations referred to the number of images that needed to be created from the Dockerfile as it in turn determined the iterations of the pull and push commands executed against them. Concurrency referred to the number of threads made available to the script to conduct the test. Repository address determined the address of the registry that was intended to be tested.

[3]https://github.com/rook/rook/pull/2456

Push Concurrency

This functionality in the test script determined the response time to a push command by the client to the registry by logging the result into an output file. Functionalities of the script are listed below.

- The script was designed to first build 100 images based on the Dockerfile that was provided to it. This number was provided as part of the iteration count parameter to the test script.
- It then ran a client with the provided iteration count and concurrency value, where the client was able to push the images created in the previous step and write a response time to a log while doing so.
- Tests were provided with manual concurrency values from the user in the form of cycles. The values provided in this test for the concurrency were 1, 3 and 6. As the concurrency value was limited to the number of threads that are made available to the cluster, it was only feasible to limit the maximum value to 6.
- Post completion of the tests against the provided concurrency value, the obtained logs in the comma separated value (CSV) format was then converted into graphs and tabular formats to evaluate the results.

Pull Concurrency

This functionality in the test script was aimed to provide the response time for a pull command by the client from the registry by logging the result on to an output file. Functionalities of the script are listed below.

- Similar to the push approach, the script was designed to build images based on the Dockerfile and the iteration count values provided to it. The iteration count provided was 100 to maintain consistency through out the tests.
- The second step involved pushing of these images to the registry and upon completion, deleting the builds from the local Docker repository.
- It then ran a client with the provided concurrency value, where the client was able to pull the images uploaded to the registry previously, onto the local Docker repository and log the response times while doing so.

- Concurrency values were provided manually in the form of cycles with the values 1, 3 and 6.
- The log files in the CSV format were then converted into graphs and tabular formats to evaluate the results obtained.

3.4 Hopsworks Integration

This thesis was intended to provide a highly available container registry on the Hopsworks platform that would also extend the multi-tenancy feature across to the registry. The Harbor registry has multi-tenancy features itself, which was integrated with Hopsworks to extend the feature with respect to Docker images stored on them. The idea behind the integration was to create a project, create a user and add the user to the relevant project automatically on Harbor when the same was done on the Hopsworks platform. The Docker image generation and storage was triggered every time the user changed the built-in project environment.

Figure 3.5 demonstrates integration of the Harbor and the Hopsworks platform using Python scripts. Working of the integration scripts are described below.

- User creation on the Hopsworks platform creates a trigger to the integration script that is responsible to create a user on the Harbor registry. Although Hopsworks has an extensive user creation approach with multiple steps involving creation, confirmation and activation of a user account before it is made available on the platform, the integration script is designed to create a user on Harbor aligned to the creation phase of the Hopsworks user creation, by using the same metadata that is used in Hopsworks.
- Project creation by a user on Hopsworks nominates the user to be the owner of the project and triggers an integration script that creates a project on Harbor with the same metadata and marks the user to be the owner of the project on Harbor. In Hopsworks, projects have quota and storage limitations that can be configured, these features are not extended onto Harbor and an unlimited quota is provided to projects on Harbor.
- Member addition feature in Hopsworks allows the project owner to add existing members already present in the Hopsworks platform to be added

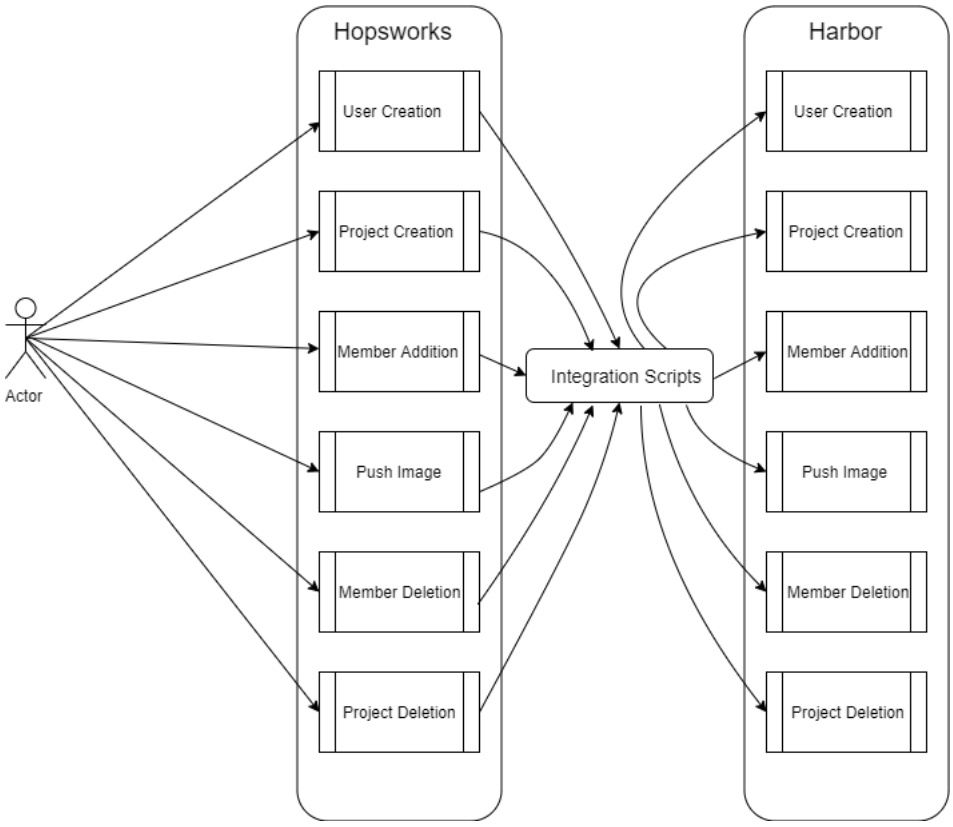


Figure 3.5: Integration of Hopsworks Multi-tenancy features into Harbor using scripts

onto a specific project with a desired role. Upon addition of a member to a project, an integration script is invoked that adds the specific user on Harbor into the selected project with an equivalent access in the project.

- Push image is a background process that pushes an image into the registry, every time a user updates the project environment in Hopsworks. The process involves updating of an existing environment, building a new Docker image of the newly built environment and further pushing it to the local repository. In this case, an integration script is invoked upon detection of a environment change, after completion of the docker image build, the image push is redirected to the Harbor registry by specifying the user that has updated the specific project on Hopsworks, this data is used to push the image onto a project on Harbor where the image needs to be stored. It also involves creation of a repository inside the project and pushes the image with a tag consisting of the current

timestamp.

- Member deletion functionality on Hopsworks allows the project owners to remove a member already existing in a project and revoke all access to the member. This triggers an integration script which removes the specified user on Harbor from the specified project. This feature just refrains the member to access the projects on both platforms and does not delete the member metadata entirely.

- Project deletion functionality on Hopsworks allows the owners to delete unwanted projects on Hopsworks. This functionality triggers a deletion of the entire project on Harbor, including all the repositories and images with all tags present in the project.

Harbor is equipped with three authentication methods. Database authentication, that stores the user information in the Harbor database with a simple salting of the password. This method is the default authentication made available on Harbor upon installation. Lightweight Directory Access Protocol (LDAP)/Active Directory (AD) authentication, where the Harbor instance is connected to an external LDAP directory server and the user accounts are allowed to be created and maintained by the LDAP provider.OpenID Connect (OIDC) provider authentication, where an external OIDC provider is connected to the Harbor instance and the user creation, maintenance is managed by the OIDC provider [40].

On the other hand, Hopsworks also allows multiple authentication methods, but the default authentication technique and the LDAP authentication methods offered by Hopsworks were chosen to maintain consistency with the Harbor authentication methods.

3.4.1 LDAP Authentication

The LDAP authentication involved building a stand-alone LDAP server on the Hopsworks cluster for both Hopsworks and the Harbor registry to authenticate against. Open LDAP is an open-source implementation of the LDAP server that is most preferred in open source applications. An Open LDAP server architecture was set up with an LDAP server and the applications were authenticated against them [4]. Figure 3.6 demonstrates the architecture of the system with the LDAP server. Steps to authenticate a user with the LDAP mechanism in the implemented architecture are listed below.

[4]OpenLDAP setup - https://www.openldap.org/

1. User attempts to login through the Harbor or Hopsworks user interface.
2. The LDAP authentication method enabled on both platforms prompts the credentials provided by the user to be authenticated against the LDAP server integrated with the system.
3. LDAP responds with a failure or a success, where a failure leads to step 6, a success leads to step 4.
4. The applications request for the required Access Control List (ACL), Roles from the database.
5. The database responds with the required data.
6. Response to the users, if authentication fails, login fails. If authentication succeeds, login is successful and tokens are generated further as part of the application feature.

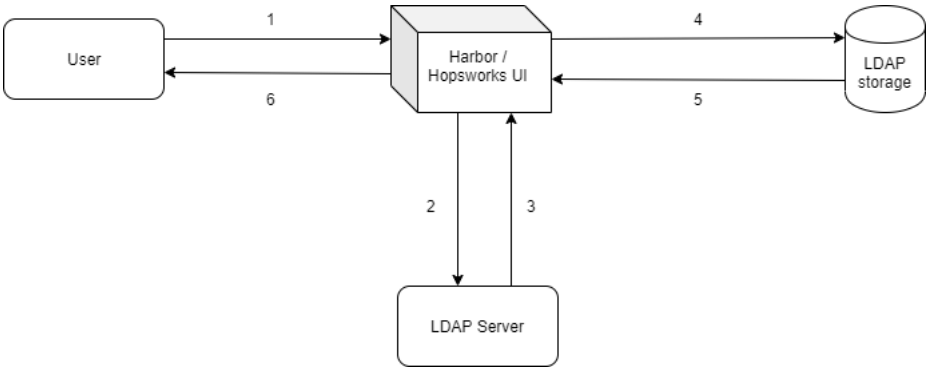


Figure 3.6: LDAP Server Architecture

3.4.2 Database Authentication

Database authentication is a method where a simple hashing technique is used to store the metadata including the password in the application database. On provision of the password by the user at the stage of account creation, the password is hashed and stored in the local database. Steps involved in this authentication mechanism are listed below.

1. User attempts to login through the Harbor or the Hopsworks user interface.

2. User name and password provided at the user interface is authenticated against the credentials stored in the local database.

3. Upon successful authentication against the credentials stored in the database, login is successful and relevant token is generated as part of the feature in the respective applications. On failure, login fails and user is denied access to the application.

3.4.3 Authorization

Both Hopsworks and Harbor provide Role-based access control as part of the authorization mechanisms to provide multi-tenancy. Hopsworks provides an admin role that has super user privileges of the application, with a project owner role that provides the user with full read and write permissions to a project with a possibility of zero or more users in it. A project owner is capable of inviting or adding users with two possible roles, data scientist, where the user has read-only privileges and is able to run jobs on the data, on the other hand, a data owner role has complete access to the data and the functionality in the project [41].

Harbor provides a similar role-based access control, where an administrator role has super user privileges of the Harbor application. A project admin is the owner of the project with complete access to a project. Project admin has the privilege of adding existing members to the project with a master role, developer role or a guest role. The master role provides users with the ability to scan images, create replication jobs or delete the images. The developer role allows users the read and write privileges in the project. The guest role allows the users to read from the specified projects and are also able to pull and re-tag images in the project [42].

The integration scripts use these roles provided by both applications and maps each of roles in Hopsworks with a role in Harbor, upon user creation and member addition into the projects. The admin user in Hopsworks has admin privileges in the Harbor platform as well, a project owner in Hopsworks is mapped to a Project admin in Harbor. At the time of member addition into the project, a user with a role data scientist on Hopsworks receives a developer role on the Harbor platform and a user with role data owner is provided with a master role in the Harbor platform. The other roles provided in the Harbor registry was neglected as part of this implementation.

3.4.4 Synchronization and Reconciliation

Hopsworks integration with Harbor extended the multi-tenant features of the Hopsworks platform onto the Harbor registry using intermediate Python scripts. These scripts were called upon from the Hopsworks platform to update the metadata on the Harbor registry with its relevant functionality being used in the Hopsworks platform. Although these Python scripts are called internally by the Hopsworks code, as the two entities are independent applications themselves, there are chances of a functionality to have not completed successfully on either platforms due to external factors. An update on Hopsworks not being successful could have already triggered for an update on the Harbor registry which might be successful. On the other hand, sometimes a successful update on Hopsworks leading to a trigger call to update the relevant metadata on Harbor registry, might fail to update the metadata on the Harbor registry due to some external factors. These incidents lead to a mismatch in the metadata on both the platforms leading to inconsistency.

Reconciliation jobs comparable to cron jobs were created to handle such incidents, where jobs run periodically to check for inconsistencies in the metadata present on the Hopsworks platform to that of the Harbor registry. It was designed to give precedence to the Hopsworks platform and synchronize the data on the Harbor registry and correct the inconsistencies identified by cleaning up the data. These jobs are configurable and can read the required information from the Hopsworks platform periodically and update accordingly on the Harbor registry, this helped maintain consistency on both the platforms. To monitor these jobs, logs were created for the jobs and saved on to output files which provided the status of the job. This way a more consistent system can be guaranteed, post completion of the integration.

3.4.5 Testing scenarios for Hopsworks Integration

The integration as discussed in the aforementioned sections, involved calling of Python scripts at certain parts of the Hopsworks modules, with an indifference in the programming languages. Hence, after an elaborate research on the possible test methods that could be chosen to test the implementation in this thesis, the manual testing approach to test the desired functionalities was chosen.

TestLodge is an online, cloud based test management tool that is useful to create and manage test plans, test requirements, test suites, test cases with

ease [5]. The manual tests against the integrated platform were performed with the use of this tool. It provided an organised approach to build the test cases and document the test results upon completion of the tests. The tests involved building six test suites, one for each multi-tenant functionality integrated between the Hopsworks and the Harbor platforms. For ease of usage, the test suites were created with the same names as that of the features mentioned in the aforementioned section. Each test suite consisted of various test cases created to test the functionality upon integration. Examples of test cases used to perform the tests is presented in the Appendix chapter at the end of the thesis.

[5] TestLodge - https://www.testlodge.com/

Chapter 4

Results and Discussion

This chapter presents the quantitative data obtained through executing performance tests against the infrastructure that was developed for this research, while the latter section discusses about the qualitative data obtained for the tests performed against the proof-of-concept system that was built as part of this thesis. The results obtained through the experiments aims to answer questions related to performance efficiency, and availability while the latter aims to provide explanations related to the desired system behaviour with the actual behaviour of the system upon integrating the registry with the Hopsworks platform. At the end of the chapter, It also describes some possible future work that can be incorporated taking this thesis as the basis for the work.

4.1 Quantitative Results

Three test cases with indifferent concurrencies were executed as part of the tests for each infrastructure. The Docker images were first built using a Dockerfile, then pushed into the registry whilst recording the push performances for a hundred iterations. The tables below denote the maximum, minimum , average and 90th percentile values of push/pull times in seconds for different concurrencies for each of the infrastructures implemented. The percentile is taken into consideration as there was an indifferent spike in values for a few iterations with a huge difference between the average and the maximum/minimum values.

Table 4.1 and Table 4.2 specify the push and pull performances for the Ceph cluster infrastructure setup. Docker daemon does not perform better with higher concurrencies for push or pull commands, they tend to deplete drastically in terms of performance with increase in concurrency. The best

performances are recorded with concurrency 1 where the Docker daemon is given one thread to perform push and pull operations to the registry.

Concurrency	Maximum	Minimum	Average	Percentile 90%
1	10.3568	1.8042	2.1560	2.4606
3	28.7090	16.2563	23.5303	25.9974
6	34.2733	12.2557	28.8011	31.9492

Table 4.1: Pull performances for Ceph cluster infrastructure in seconds

Concurrency	Maximum	Minimum	Average	Percentile 90%
1	5.9563	2.9114	3.02223	3.0662
3	273.1366	7.8121	47.5520	57.9080
6	97.0058	46.0710	67.4251	74.4499

Table 4.2: Push performances for Ceph cluster infrastructure in seconds

Table 4.3 and Table 4.4 provides results of the performances of push and pull commands by a Docker daemon for a local file-system storage infrastructure. The trend with lower performances with higher concurrencies can be deduced with the results even for local file-system storage infrastructure. The best performances, similar to the Ceph cluster architecture, can be deduced to be obtained with concurrency 1 in terms of push and pull commands trying to manage the images in the registry.

Concurrency	Maximum	Minimum	Average	Percentile 90%
1	2.7064	1.6566	1.8756	2.2170
3	21.9744	9.2732	15.4830	17.7900
6	35.7457	17.5459	30.4995	33.9605

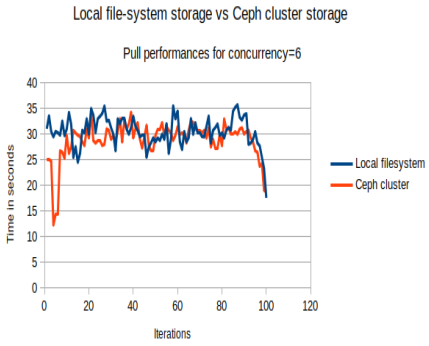
Table 4.3: Pull performances for local file-system infrastructure in seconds

The pull and push performances for concurrencies 3 and 6 are graphically represented below, it is evident that the overall performances of the local storage when compared to that of a Ceph cluster storage are lower for these concurrencies. The pull performances with concurrency 6 and concurrency 3 for local storage is consistently lower than that of the Ceph cluster storage, although,

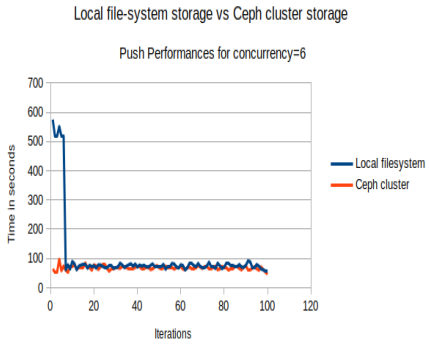
Concurrency	Maximum	Minimum	Average	Percentile 90%
1	10.4080	4.6826	4.9934	5.0840
3	321.1059	11.7018	43.7488	41.0400
6	574.6916	56.3650	101.4719	86.9933

Table 4.4: Push performances for local file-system infrastructure in seconds

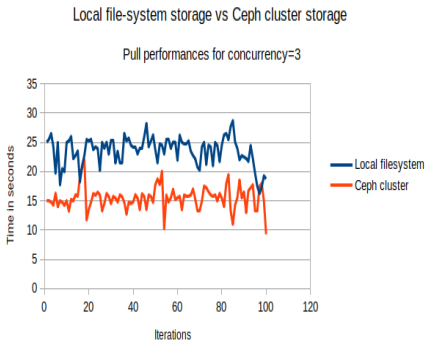
there were a few iterations, where the performances were similar for both as seen with the spikes in the graph. However, the push performances were observed to be very consistent and similar for both storage infrastructures apart from a drastic spike with the first iteration in the local storage infrastructure. Although, there exists a difference in the average push performances, a 90th percentile would bare more relevant observations for these results.



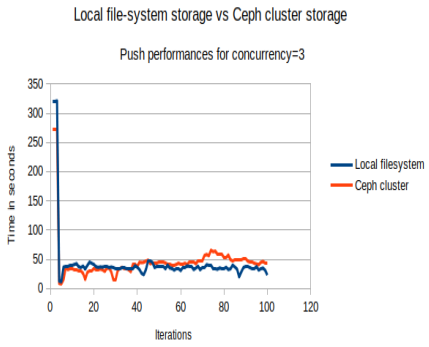
(a) Pull performances (concurrency=6)



(b) Push performances (concurrency=6)



(c) Pull performances (concurrency=3)



(d) Push performances (concurrency=3)

As part of the performance tests, different concurrencies were chosen. But in a production system, the Docker daemon is ideally provided a single thread

to perform its tasks which ideally works with concurrency 1. Hence, the results of concurrency 1 are considered to be evaluated and discussed further.

When Docker daemon was run with single concurrency, the system initially took more time to push the artifacts of the Docker image into the Harbor registry. Once the image was pushed, as the difference in the artifacts with respect to the iteration 1 to the images pushed in the next iterations are very less, Docker daemon re-used the layers already existing in the registry and pushed only the layers that have changed. Hence, further push times were relatively lower when compared to the first iteration. In comparison with the Ceph cluster storage infrastructure, the local storage infrastructure took significantly more time to push the image in each iteration although both infrastructures took relatively more time for their respective first iterations.

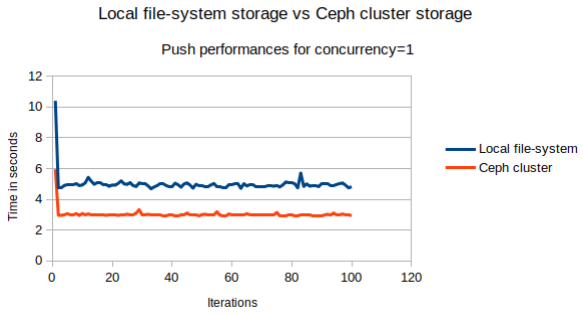


Figure 4.2: Push performances (concurrency=1)

For the local storage infrastructure, iteration 1 required more than 10 seconds to pull the image from the Harbor registry and further after the artifacts were populated in the local repository, it took consistently lesser time compared to the first iteration. But conversely, the Ceph cluster storage with single concurrency has higher performances consistently and was below the 3 second mark through out the experiment. This proved to be significantly more efficient than the Harbor registry with a local storage infrastructure.

This efficiency is possible with distributed storage mechanism on the cluster as the local storage infrastructure uses a specific node's disk space to provide storage and the data is transferred over the network if the storage does not exist on the same node. Hence the latency increases which may lead to higher timelines, this provides good observations as to why the Ceph cluster storage infrastructures are better for storing these Docker images on the Kubernetes cluster when compared to the local storage infrastructure.

Previous theoretical research were also in accordance to the results obtained, leaning towards the Ceph cluster implementations over the local-storage

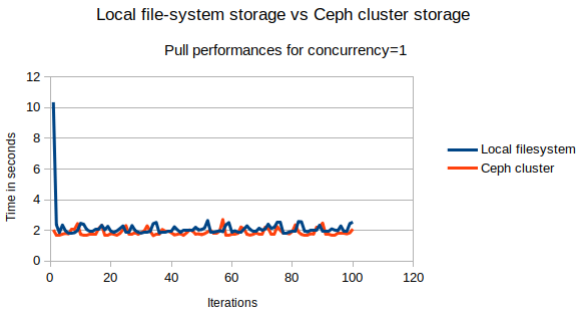


Figure 4.3: Pull performances (concurrency=1)

implementations, as these might cause a single point of failure, where, if the node on which the registry storage exists fail, the system fails to function although a replication is made available on another node. This again does not make the system self healing and limits the powers of Kubernetes as a self healing and fault tolerant orchestrator.

4.2 Qualitative Results

The result of the manual tests performed against the integrated platform using the cloud-based testing tool TestLodge are presented in this section. The tests were performed in batches with test cases for each test suite executed separately. Each test suite consisted of 2 to 4 test cases that were to be executed to complete the test suite run. As a result of the tests performed, a total of 18 test cases were executed across the 6 test suites and a total of 13 test cases passed with the remaining failing these tests.

Test suites	Test cases executed	Passed	Failed
6	18	13	5

Table 4.5: Test cases passed vs failed

The failed test cases were evaluated closely, all the test cases that failed were a part of the meta data mismatch across the two integrated platforms. The Python scripts called through the Hopsworks code to regulate the functionality in Harbor is checked for a successful execution, if return values from the scripts denotes a success, only then the functionality is allowed to complete. If the Python scripts returns a non-zero value, then the Hopsworks code

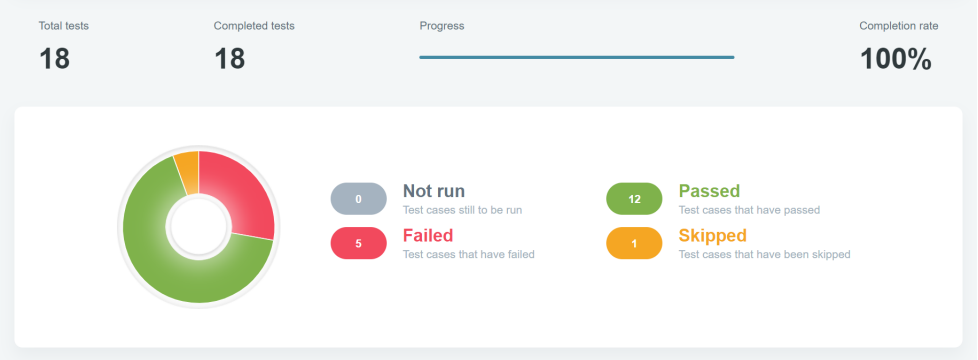


Figure 4.4: Snapshot from the TestLodge tool

execution throws an exception and the functionality does not complete. Considering these failed test cases were due to a forceful deletion of metadata on one of the platforms, these errors are less prone to occur in the actual system. Aforementioned reconciliation jobs designed to check for any mismatch in the metadata across platforms and updating the Harbor metadata accordingly will be used in practice to negate these scenarios. Time intervals for these jobs are critical to avoid such failures or errors in the system.

4.3 Future Work

This section aims to expose possible continuations of this work. These propositions are made mainly based on the limitations of the current implementations. As mentioned in the earlier chapters, this thesis is a proof-of-concept for the integration of Hopsworks platform with the Harbor registry and Python scripts are used to perform the API calls to update the metadata and perform the functionalities on Harbor. If this implementation is to be realised on a production environment, the best practice will be to develop Java based integration calls to the Harbor API to perform the functionality as the current approach uses a less secure password management technique where the credentials needs to be passed to the Python scripts as part of the integration. This can be done in a more elegant, secure way with consistency in the programming languages. Also, a good approach would be to test the same architectures with more testing scenarios. It will then provide more data points for comparison and will allow to better draw clear conclusions from statistical test results. As this thesis was aimed to provide efficiency in-terms of performance of Docker image pull and push to the registry, the performed tests were limited to these functionalities. The security aspects were not really explored with re-

spect to the registry as part of this thesis, where Harbor provides improved security for the images in terms of allowing signed images to be pushed into the registry. This could be a great addition to the registry features to be implemented. Conducting more complex experiments is also very efficient way to spot some potential new issues not detected by this work's testing scenarios. Integration tests performed for this project was manual and limited to the scope of extended functionalities. This might also encourage to extend more sophisticated features present on Hopsworks to be extended onto the registry.

Chapter 5

Conclusion

In this era of big data context, there is always a need for efficient scalable and fault tolerant software architectures in order to process the growing amount of data. Hopsworks is one such platform that provides great features that integrates many big data frameworks under one hood. With that being said, there is also a need for a secure container registry to be implemented on to the platform capable of extending the main multi-tenancy features of the Hopsworks platform.

In this report, two architectures based on a storage mechanism were compared. The implemented infrastructures perform well but differs in terms of fault tolerance and slightly in terms of performance. This is explained with some experimental results that show that the distributed Ceph cluster storage infrastructure performed better on Docker push and pull commands when compared to the local-storage infrastructure. The fact that the data placements when storing these images play a vital role in the performance was determined by these experiments. Also it is evident that the infrastructure with the distributed Ceph cluster approach provides better fault tolerance as the local-storage infrastructure has a single point of failure.The experimental results show that the storage infrastructures opted for the container registry has an important impact on performances of a system as a whole. However, the implemented solutions are rather simple, and some robustness issues occur during experiments. A direct continuation of the current work is to fix these issues and perform more tests, especially testing scenarios with more images. The limitation of the experiments is the computational power of the machines of the testing cluster as most performance metrics are put to test with high performance machines, But, this was also enough to spot the most obvious characteristics of the tested architecture.

The results of this study cannot be used directly to develop a complete integrated architecture for production as it only focuses on a few parameters and was designed to be more of a proof-of-concept implementation. Further experiments are required in order to determine an optimal solution that can be deployed in a production environment. However, the lessons learned from this study surely will be very useful when implementing an integrated Harbor registry for the Hopsworks platform in a production environment.

Bibliography

[1] *Big Data - A critical path to develop new business oppurtunities*. https://www.ie.edu/insights/articles/big-data-critical-path-to-develop-new-business-opportunities/. Accessed: 2020-04-07.

[2] *Apache Hadoop*. https://hadoop.apache.org/. Accessed: 2020-04-07.

[3] *Hopsworks Documentation 0.9*. https://hopsworks.readthedocs.io/en/0.9/overview/introduction/what-hopsworks.html. Accessed: 2020-04-09.

[4] *How to build a collaborative data science environment*. https://blogs.oracle.com/datascience/how-to-build-a-collaborative-data-science-environment. Accessed: 2020-04-09.

[5] *Conda environment administration*. URL: https://hopsworks.readthedocs.io/en/1.2/admin_guide/conda.html.

[6] Alan R. Hevner, Salvatore T. March, Jinsoo Park, Sudha Ram. "Design Science in Information Systems Research". In: *Management Information Systems Quarterly* 28 (2004), p. 75.

[7] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, Samir Chatterjee. "A Design Science Research Methodology for Information Systems Research". In: *Journal of Management Information Systems* 24 Issue 3 (2007-08), pp. 45–78.

[8] *IEEE Code of Ethics*. https://www.ieee.org/about/corporate/governance/p7-8.html. Accessed: 2020-04-12.

[9] *5 things about data science*. https://www.kdnuggets.com/2018/02/5-things-about-data-science.html. Accessed: 2020-04-20.

[10] *Data science*. https://www.binghamton.edu/transdisciplinary-areas-of-excellence/data-science/index.html. Accessed: 2020-04-20.

[11] Michael Brodie. "Understanding Data Science: An Emerging Discipline for Data-Intensive Discovery". In: Sept. 2015, pp. 33–51.

[12] *What is Data Science? Transforming data into value*. https://www.cio.com/article/3285108/what-is-data-science-a-method-for-turning-data-into-value.html. Accessed: 2020-04-12.

[13] Kristopher Overholt Christine Doig. *White paper: Productionizing and Deploying Secure and Scalable Data Science Projects*. Tech. rep. Continuum Analytics, June 2017.

[14] *What is big data?* https://www.oracle.com/big-data/what-is-big-data.html. Accessed: 2020-04-25.

[15] D. P. Acharjya, Kauser Ahmed P. "A Survey on Big Data Analytics: Challenges, Open Research Issues and Tools". In: *International Journal of Advanced Computer Science and Applications* 7 Issue 2 (2016), pp. 511–515.

[16] Olshannikova, E., Ometov, A., Koucheryavy, Y. et al. "Visualizing Big Data with augmented and virtual reality: challenges and research agenda". In: *Journal of Big Data2* 22 (2015).

[17] *How Hadoop helps solve Big Data Problem*. https://www.techopedia.com/2/30166/technology-trends/big-data/how-hadoop-helps-solve-the-big-data-problem. Accessed: 2020-04-15.

[18] *What is Hops?* https://hopsworks.readthedocs.io/en/latest/overview/introduction/what-hops.html. Accessed: 2020-04-15.

[19] *Ceph (software)*. URL: https://en.wikipedia.org/wiki/Ceph_(software).

[20] *INTRO TO CEPH*. URL: https://docs.ceph.com/docs/master/start/intro/.

[21] *What is virtualization?* URL: https://opensource.com/resources/virtualization#:~:text=Virtualization.

[22] R. Morabito, J. Kjällman, and M. Komu. “Hypervisors vs. Lightweight Virtualization: A Performance Comparison”. In: *2015 IEEE International Conference on Cloud Engineering*. 2015, pp. 386–393.

[23] *Containers and Cloud: From LXC to Docker to Kubernetes*. URL: http://www.ce.uniroma2.it/courses/sdcc1617/articoli/bernstein_cc2014.pdf.

[24] *Open container project*. URL: https://www.docker.com/blog/open-container-project-foundation/.

[25] D. Bernstein. “Containers and Cloud: From LXC to Docker to Kubernetes”. In: *IEEE Cloud Computing* 1.03 (Sept. 2014), pp. 81–84. ISSN: 2372-2568. DOI: 10.1109/MCC.2014.51.

[26] *Docker Overview*. URL: https://docs.docker.com/get-started/overview/.

[27] *Borg,Omega, and Kubernetes*. URL: https://dl.acm.org/doi/pdf/10.1145/2890784.

[28] *Kubernetes Components*. URL: https://kubernetes.io/docs/concepts/overview/components/.

[29] *Overview of kubectl*. URL: https://kubernetes.io/docs/reference/kubectl/overview/.

[30] *Persistent Volumes*. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[31] *Understanding Kubernetes storage basics*. URL: https://cloud.ibm.com/docs/containers?topic=containers-kube_concepts.

[32] *Do you know what’s in Helm 3?* URL: https://developer.ibm.com/technologies/containers/blogs/kubernetes-helm-3/#:~:text=Helm.

[33] *Charts*. URL: https://helm.sh/docs/topics/charts/#:~:text=Helm.

[34] *Hopsworks Documentation 1.2*. https://hopsworks.readthedocs.io/en/1.2/overview/introduction/what-hopsworks.html. Accessed: 2020-05-15.

[35] *Trow*. URL: https://github.com/ContainerSolutions/trow.

[36] *Trow Architecture*. URL: https://github.com/ContainerSolutions/trow/blob/master/docs/ARCHITECTURE.md.

[37] *Harbor*. URL: https://goharbor.io/.

[38] *Rook - Open-Source, Cloud-Native Storage for Kubernetes?* https://rook.io/. Accessed: 2020-08-10.

[39] *Rook - Ceph Storage*. https://rook.io/docs/rook/v1.4/ceph-storage.html. Accessed: 2020-08-10.

[40] *Harbor authentication*. https://goharbor.io/docs/1.10/administration/configure-authentication/. Accessed: 2020-08-12.

[41] *How we secure your data with Hopsworks*. https://www.logicalclocks.com/blog/how-we-secure-your-data-with-hopsworks. Accessed: 2020-08-18.

[42] *Managing Users*. https://goharbor.io/docs/1.10/administration/managing-users/. Accessed: 2020-08-18.

Appendix A

Additional Infrastructure Design

A.1 Persistent Storage with HopsFS

Hopsworks utilizes HopsFS for it's data and meta data storage. So with HopsFS already being a part of the technology stack in the Hopsworks platform, the current architecture proposes utilizing HopsFS as a medium to store the Docker image manifests.

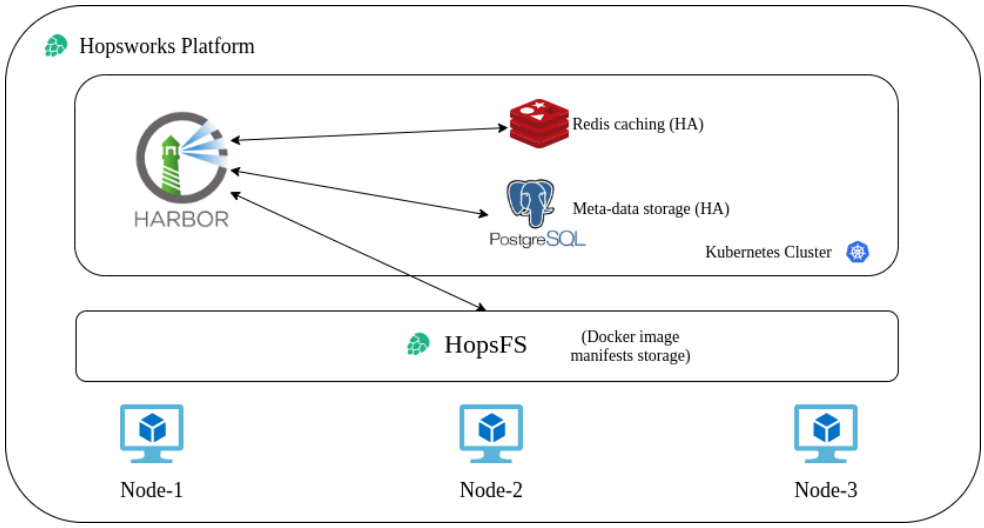


Figure A.1: HopsFS Storage Architecture

- Harbor utilizes PostgreSQL, Redis for metadata storage and caching purposes. The image manifests are simply stored in the file-system on the

pod it is running on. This image manifest data is persisted if the data on the pod is persisted, if not, they are destroyed with the pod as well.

- As a replacement, the current architecture uses the existing HopsFS distributed file-system to store the image manifests to persist the data even after the completion of the pod life-cycle.

- *Replace Overlay2 with an HDFS compliant storage driver* : HopsFS is built on top of HDFS, storage driver programs that are compatible with HDFS that helps to read and write data into them are used to populate the image manifests into HopsFS on pushing the image onto the Harbor registry. Overlay2 is the standard storage driver used by Docker to store images onto the local file-system using an OverlayFS as part of the CentOS Linux Distribution. This storage driver is replaced with an HDFS storage driver enabling the docker daemon to directly store the image manifests into HDFS [1].

- *NFS gateway to mount HDFS file-system on Kubernetes Pods* : In this approach, an NFS gateway is used in combination with the existing file-system for the clients to mount the HopsFS onto the system and interact with the file-system with the help of NFS as if it were part of their local file-system. This NFS gateway can be installed on any DataNode or Name-Nodes on the HopsFS system, it requires an NFS server to be setup and running on one of these machines. Post mounting the HopsFS, users can browse the file-system through the local file-system on an NFSv3 client compatible operating system. The gateway also allows the users to upload and download files directly from and to the local file-system from HopsFS. It also allows to stream data directly through the mount point [2].

- As a result of the lack of support for HDFS storage with Docker distribution, as no stable drivers made available for the purpose, this method is not ideal to be implemented. On the other hand, the incompatibility of Hopsworks with the NFS gateway based on previous research, urges the author to not use this storage implementation as part of this thesis.

[1] Docker Storage Drivers - https://docs.docker.com/storage/storagedriver/select-storage-driver/

[2] HDFS NFS Gateway - https://hadoop.apache.org/docs/r2.8.0/hadoop-project-dist/hadoop-hdfs/HdfsNfsGateway.html

Appendix B

Dockerfile for Infrastructure Test

Listing B.1: Dockerfile used to perform the tests

```
FROM ubuntu:14.04
RUN apt-get install -y nginx
EXPOSE 80
CMD /usr/sbin/nginx -g 'daemon off;'
```

Simple Dockerfile having functionality to run an Nginx web server and expose the port 80 on an Ubuntu 14.04 operating system was developed to use in the test scenarios.

Appendix C

Integration Test-case Examples

Below are a few example snapshots of successful test cases run in the cloud based testing tool Test lodge.

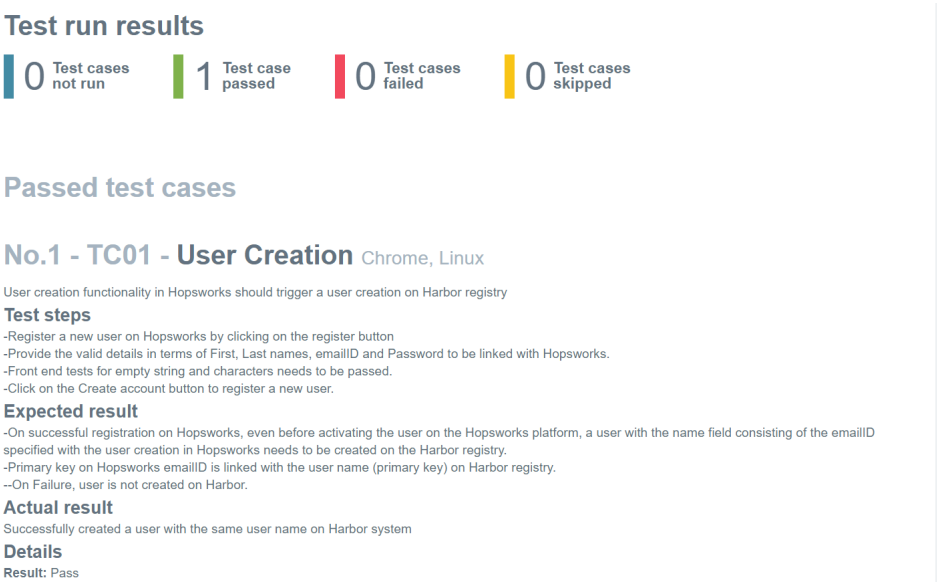


Figure C.1: User creation test case result from the Test Lodge platform

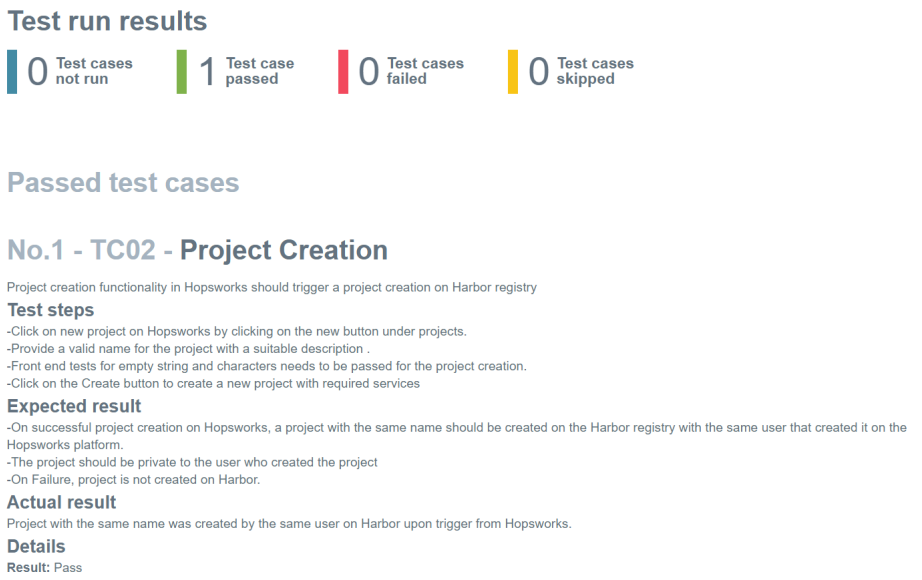


Figure C.2: Project creation test case result from the Test Lodge platform

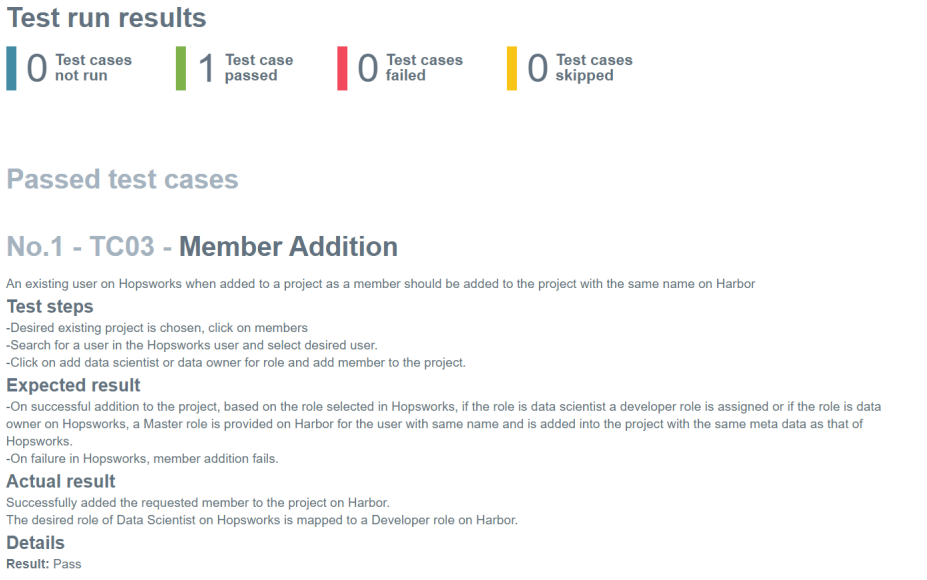


Figure C.3: Member addition test case result from the Test Lodge platform

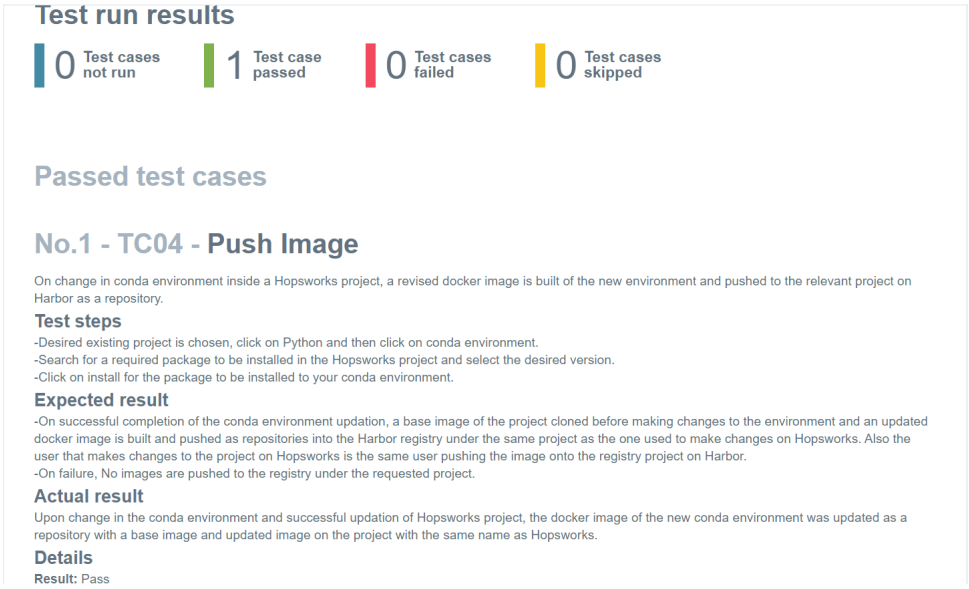


Figure C.4: Docker image push test case result from the Test Lodge platform

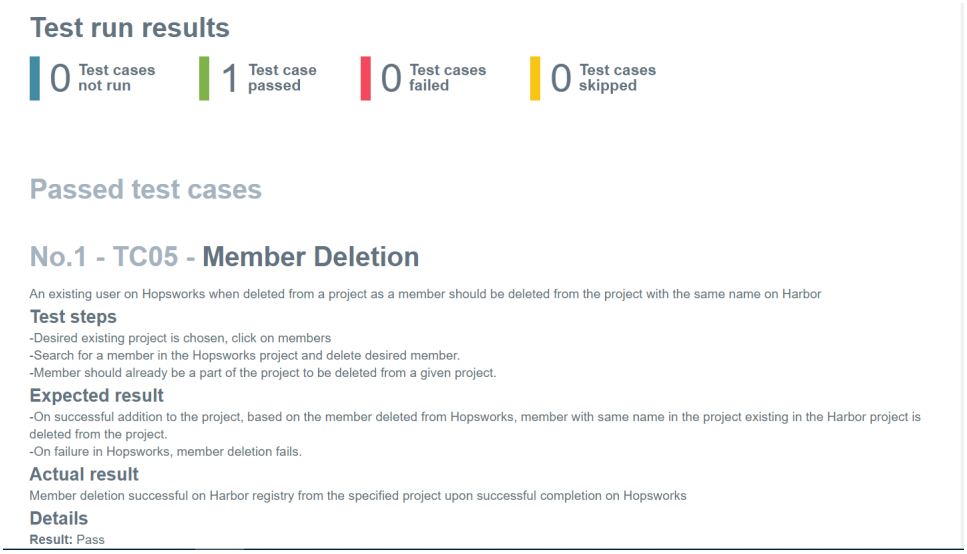


Figure C.5: Member deletion test case result from the Test Lodge platform

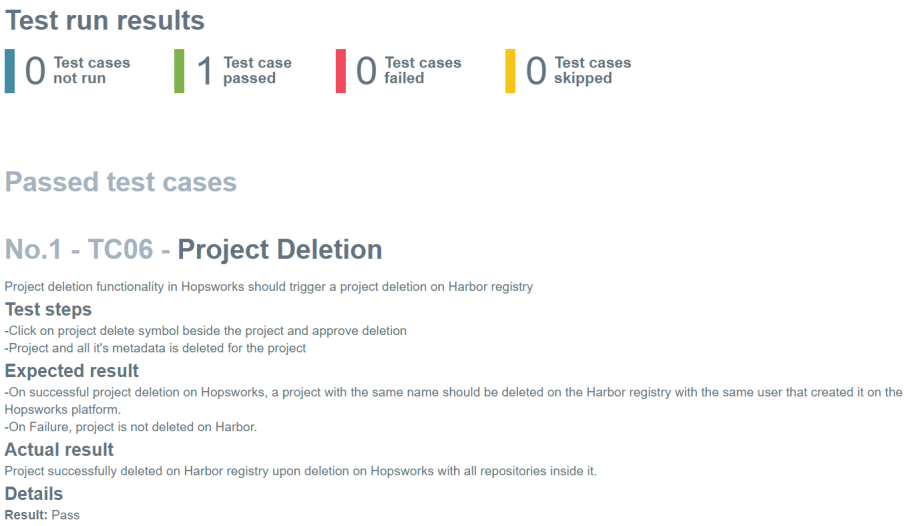


Figure C.6: Project deletion test case result from the Test Lodge platform