Terentev Nikita

DESIGN AND DEPLOYMENT OF A TEST INFRASTRUCTURE FOR SERVICE MIGRATION AND TELEMETRY ENHANCEMENT

Technology and Communication

2025

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Terentev Nikita
Title	Design and Deployment of a Test Infrastructure for Service Migration and Telemetry Enhancement
Year	2025
Language	English
Pages	85 + 0 Appendices
Name of Supervisor	Matti Tuomaala

This thesis investigates the implementation of a cloud-native, open-source infrastructure stack deployed on a bare-metal Kubernetes cluster. The study addresses the need for scalable, resilient, and observable platforms capable of supporting complex data and application workflows without relying on proprietary or cloud-hosted services.

The theoretical framework is based on principles of distributed systems, declarative infrastructure management, and automation. Emphasis is placed on Infrastructure as Code, templating, and declarative configuration to ensure reproducibility and maintainability of system components. The research draws from best practices in DevOps and Site Reliability Engineering (SRE), with a focus on fault tolerance, eventual consistency, and modular service design.

The results show that a fully open-source stack can be integrated into a coherent platform supporting storage, authentication, monitoring, logging, and metadata governance. While not aiming to compare directly with managed cloud services, the implementation demonstrates that open infrastructure is viable for complex workflows in controlled environments. Its successful orchestration highlights the practical and cost-efficient benefits of using open-source tools to build customizable, maintainable systems on bare-metal hardware.

Keywords Kubernetes, DevOps, open-source, infrastructure

CONTENTS

ABSTRACT 2
1 INTRODUCTION 8
1.1 Motivation 8
1.2 Scope 8
1.3 Objectives and Company-Specific Requirements 9
2 CONTAINERIZATION FUNDAMENTALS 11
2.1 Virtual Machines vs. Containers: An Architectural Perspective 11
2.2 Container Orchestration Systems 13
3 KUBERNETES FUNDAMENTALS 14
3.1 Cluster Architecture 14
3.1.1 Master Node (Control Plane) 14
3.1.2 Worker Node 16
3.2 Key Concepts 17
3.2.1 Pods, Replica Sets, Deployments, Namespaces, Selectors 17
3.3 Kubernetes Networking 22
3.3.1 ClusterIP, NodePort, Ingress, Ingress Controller, LoadBalancer 22
3.4 Configuration and Secret Management 26
3.4.1 ConfigMaps 27
3.4.2 Secrets 27
3.5 Kubernetes Storage 28
3.6 Key Scheduling Concepts 29
3.7 Operators, CRDs 30
4 LOAD BALANCERS 32
4.1 Load Balancing Overview 32
4.2 MetalLB 33
5 MONITORING 35
5.1 Metrics 35
5.2 Logging 37
6 AUTOMATION AND TEMPLATING 38

6.1 Ansible 38
6.2 Helm 40
7 AUTHENTICATION 41
7.1 OpenID Connect (OIDC) 41
7.2 Identity Providers 42
8 DATAHUB 45
9 OBJECT STORAGE (MINIO) 47
10 SYSTEM ARCHITECTURE 49
11 SYSTEM IMPLEMENTATION 50
11.1 Kubernetes Cluster Setup with Ansible 50
11.2 Longhorn and MinIO Installation 52
11.3 Nginx Ingress Controller Installation 63
11.4 Keycloak configuration 64
11.5 MetalLB deployment 66
11.6 DataHub deployment 68
11.7 Prometheus operator deployment 72
11.8 Loki deployment 75
11.9 Logging operator deployment 77
12 CONCLUSION AND FUTURE WORK 80
REFERENCES 82

FIGURES

Figure 1. Virtualization vs Containerization (ByteByteGo, 2022)....... 12
Figure 2. Cluster Architecture (CNCF, 2024)................................ 15
Figure 3. Pod internals 18
Figure 4. ReplicaSet 19
Figure 5. Deployment 20
Figure 6. Namespaces... 21
Figure 7. Label-Selector based trafic routing 22
Figure 8. ClusterIP 23
Figure 9. NodePort 24
Figure 10. Ingress (CNCF, 2024) 25
Figure 11. ConfigMaps and Secrets 26
Figure 12. Dynamic storage provisioning with Longhorn 28
Figure 13. Logging workflow 37
Figure 14. Longhorn requirements setup 39
Figure 15. inventory.ini used for the setup.................................. 39
Figure 16. Authentication process with OIDC (openid.net, n.d.) 42
Figure 17. Recipe configuration (docs.datahub.com) 45
Figure 18. DataHub architecture (docs.datahub.com) 46
Figure 19. System Architecture 49
Figure 20. k8s prerequisites setup with ansible, k8s-cluster.yaml..... 50
Figure 21. Cluster initialization and network setup, kubeadm-create.yaml 51
Figure 22. Longhorn prerequisites installation with Ansible, Longhorn-prereqs.yaml 53
Figure 23. Longhorn values file 54
Figure 24. docker-compose.yml, part 1 55
Figure 25. docker-compose.yml, part 2 56
Figure 26. nginx.conf, part 1 57
Figure 27. nginx.conf, part 2 58
Figure 28. nginx.conf, part 3 59
Figure 29. MinIO access token management page 60
Figure 30. MinIO credentials secret.. 60

Figure 31. Backup target configuration 61
Figure 32. Backup target status 61
Figure 33. Longhorn volume management interface 62
Figure 34. Longhorn volume backup history 62
Figure 35. Longhorn backup management panel 62
Figure 36. MinIO bucket contents 62
Figure 37. values.yaml for nginx ingress controller 63
Figure 38. Datahub realm 64
Figure 39. Keycloak client configuration 65
Figure 40. Keycloak federation configuration 65
Figure 41. List of users imported from Active Directory 66
Figure 42. l2-adv.yaml 67
Figure 43. address-pool.yaml 67
Figure 44. ns.yaml 68
Figure 45. values.yaml authentication section 69
Figure 46. OIDC k8s secret 69
Figure 47. values.yaml Ingress configuration section 70
igure 48. TLS key and certificate k8s secret 70
Figure 49. Keycloak authentication page 71
Figure 50. DataHub main page after Keycloak authentication 71
Figure 51. grafana-service.yaml 73
Figure 52. dashboard-tls.yaml 73
Figure 53. grafana-ingress.yaml 74
Figure 54. Grafana cluster compute resources dashboard using Prometheus source 74
Figure 55. Loki values.yaml 76
Figure 56. logging.yaml 78
Figure 57. flow.yaml 78
Figure 58. output.yaml 79
Figure 59. Grafana with Loki source 79

ABBREVIATIONS

ARP	Address Resolution Protocol
BGP	Border Gateway Protocol
CNCF	Cloud Native Computing Foundation
CRD	Custom Resource Definition
CSI	Container Storage Interface
IP	Internet Protocol K8s Kubernetes
OIDC	OpenID Connect
OP	OpenID Provider
OSS	Open-source Software
PV	Persistent Volume
PVC	Persistent Volume Claim
RP	Relying Party
SSO	Single Sign-On
VM	Virtual Machine

1 INTRODUCTION

1.1 Motivation

In today's data-centric technological environment, data science teams depend heavily on scalable and flexible infrastructure to conduct meaningful research, manage metadata, and store large volumes of data, including image and video materials. Building and maintaining such infrastructure manually is often labor-intensive, error-prone, and inefficient. The emergence of modern DevOps practices, containerization, and infrastructure-as-code tools has transformed the way complex environments are provisioned and maintained. With technologies like Kubernetes and Ansible, it is now possible to automate the setup of distributed systems, dynamically allocate storage, and configure internal and external traffic routing with minimal manual intervention.

Additionally, the growing ecosystem of open-source tools—such as Longhorn, MetalLB, and DataHub—provides powerful building blocks for creating tailored, production-like environments even in testing scenarios. This thesis is motivated by the need to create a reproducible, modular, and extensible test environment specifically designed for data science workflows. By integrating key services such as object storage, logging, metrics collection, and metadata analysis tools, the infrastructure aims to significantly reduce the time required to deploy and manage essential services, while increasing observability and scalability.

1.2 Scope

This thesis focuses on the design and implementation of an infrastructure platform to support the needs of a data science team. The solution leverages infrastructure-as-code, container orchestration, and modern DevOps practices to deploy and manage essential services, such as

metadata platforms, observability stacks, and storage systems within a Kubernetes cluster. On the infrastructure engineering side, the thesis includes analysis and planning of key components, such as dynamic storage provisioning, internal and external traffic routing, user authentication through SSO, and integration of metadata analysis tools.

These components are deployed with the goal of building a modular, scalable, and maintainable architecture that meets the operational demands of data-driven workflows. The DevOps aspect involves evaluating and configuring orchestration technologies (primarily Kubernetes), provisioning automation tools like Ansible, and integrating supporting services such as Longhorn, MetalLB, and DataHub. Additionally, the thesis covers the setup of monitoring and logging tools—including Prometheus, Grafana, FluentBit, Kafka, and MinIO—ensuring observability, backup support, and reliable data handling withing the system.

1.3 Objectives and Company-Specific Requirements

This thesis was made in collaboration with Ilim Group. The company specializes in the pulp and paper industry, leveraging AI to automate warehouse and logistics operations. The company employs 17,000 people and is the largest pulp and paper enterprise in Russia. Ilim Group ranks third in Europe and tenth in the world in terms of pulp production volume. It produces 75% of all market pulp in Russia, 20% of the country’s cardboard, and 10% of its paper. Ilim is well-known not only in Russia — the company is among the top 10 most in-demand pulp producers in China.

The primary objective of this thesis is to design and implement a fully automated, open-source test infrastructure for deploying and managing services used by a data science team. The infrastructure should support core services, such as metadata management, monitoring, logging, and object storage, while ensuring flexibility, scalability, and maintainability. A key requirement from the company is that the entire solution must be

based on open-source and cloud-native technologies. Due to cost constraints and security policies, the infrastructure must run entirely on bare-metal servers in an isolated environment, without reliance on commercial cloud platforms or paid third-party services.

The solution must include the deployment of a Kubernetes cluster using Ansible and integrate supporting tools such as Longhorn for persistent storage provisioning, MetalLB for load balancing in bare-metal environments, and DataHub for metadata visualization. Additionally, the system must provide logging and monitoring capabilities through tools like FluentBit, FluentD, Kafka, Prometheus, and Grafana. The object storage for backup and research data must be handled via MinIO. By meeting these goals, the project aims to deliver a reliable and reusable environment for experimentation, service testing, and data analysis workflows—aligned with the company's operational limitations and technical aspirations.

2 CONTAINERIZATION FUNDAMENTALS

In today's cloud-native world, containerization has become a popular and efficient alternative to traditional virtualization. This section explains the basic ideas behind containerization, how it differs from virtual machines, and why it is now the preferred way to deploy and scale modern applications.

2.1 Virtual Machines vs. Containers: An Architectural Perspective

In modern computing, both virtual machines (VMs) and containers serve the purpose of resource isolation and application deployment. However, they differ significantly in architecture, performance, and operational complexity. A virtual machine emulates an entire physical computer system, including its own operating system kernel. VMs run on a hypervisor—a software layer that enables multiple operating systems to run concurrently on a single physical machine. The hypervisor can be of two types: Type 1 (bare-metal), which runs directly on hardware (e.g., VMware ESXi), and Type 2 (hosted), which runs on a host operating system (e.g., VirtualBox) (Canonical Ltd, January 18, 2023).

Each VM includes a full guest operating system, virtualized hardware (CPU, memory, storage), application code and dependencies. While VMs offer strong isolation and are ideal for legacy applications or multi-tenant infrastructure, they are resource-intensive. Each VM consumes significant system resources due to its independent OS instance and overhead introduced by the hypervisor (Canonical Ltd, January 18, 2023).

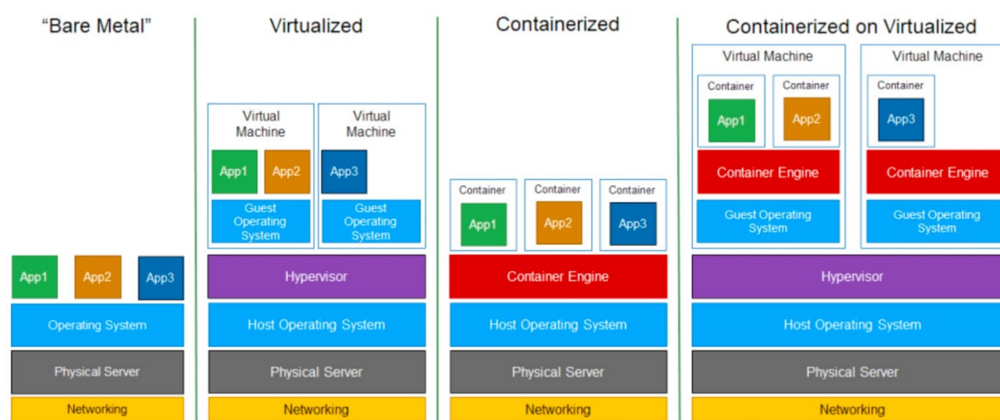

Figure 1. Virtualization vs Containerization (ByteByteGo, 2022)

In contrast, as shown on the Figure 1, containers share the host operating system's kernel and isolate applications at the process level, rather than at the hardware emulation level. Containers do not require a hypervisor or a separate guest OS, making them much lighter and faster to start than VMs (Canonical Ltd., January 18, 2023). Containers achieve isolation and resource control through two primary Linux kernel features: namespaces (Docker Inc, n.d.) and cgroups (Datadog, May 23, 2023).

Containers represent a resource-efficient paradigm for application deployment, wherein each application operates within its own isolated environment along with all necessary dependencies. This model eliminates the potential for dependency conflicts and ensures consistent behavior across different computing environments. In contrast to virtual machines—which are typically more resource-intensive and often host multiple applications—containers leverage lightweight abstraction and shared operating system kernels. This distinction makes containers par-

ticularly well-suited for microservice-based architectures, where modularity, reproducibility, and scalability are critical to system reliability and maintainability.

2.2 Container Orchestration Systems

In modern computing environments, applications are often composed of numerous loosely coupled services, each packaged within its own container. Managing these containers manually becomes impractical at scale, especially in dynamic cloud-native architectures. Container orchestration refers to the automation of deploying, managing, scaling, and networking containers throughout their entire lifecycle, enabling consistent software deployment across various environments and at scale (RedHat, March 21, 2025).

Orchestration systems, such as Kubernetes, Docker Swarm, and Apache Mesos, provide mechanisms to maintain the desired state of applications by handling container scheduling, load balancing, health monitoring, rolling updates, and resource allocation automatically. These systems continuously monitor application performance and can reschedule containers in the event of node failures, ensuring fault tolerance and high availability. Moreover, they facilitate service discovery and network management by abstracting complex infrastructure interactions, allowing developers to focus on application logic rather than underlying deployment details (RedHat, March 21, 2025). As a result, container orchestration is a critical component in the implementation of scalable, resilient, and portable microservice architectures.

3 KUBERNETES FUNDAMENTALS

The following chapter provides an overview of the main Kubernetes concepts, the understanding of which is essential for the design of the system and technologies used in this work. Kubernetes (k8s) is a powerful open-source platform originally developed by Google. Its history is rooted in Google's internal system called Borg, which had been used for managing large-scale containerized applications for over a decade. Drawing on this experience, Google introduced Kubernetes in 2014 as a more accessible and extensible system for container orchestration. In 2015, Kubernetes was donated to the newly formed Cloud Native Computing Foundation (CNCF), a part of the Linux Foundation. This move helped promote community-driven development and adoption across the tech industry. Since its release, Kubernetes has rapidly become the de facto standard for managing containerized workloads and services. It has been embraced by major cloud providers and enterprises due to its flexibility, scalability, and strong ecosystem of tools and extensions.

The k8s architecture is based on a cluster model consisting of control plane components (Master Nodes) and Worker Nodes. Together, these components ensure the system is highly available, fault-tolerant, and scalable. Each worker node is a physical or virtual machine tasked with running containerized applications, while the Control Plane handles scheduling and orchestration. The Control Plane can be deployed on a dedicated host or run alongside worker node components on the same machine. (CNCF, October 20, 2024)

3.1 Cluster Architecture

3.1.1 Master Node (Control Plane)

The master node is responsible for the global decisions about the cluster (e.g., scheduling), detecting and responding to cluster events, and exposing the Kubernetes API. Its core components are explained next.

The architecture of Kubernetes cluster is shown on the Figure 2. API Server is a central interface for all communication with the cluster that handles RESTful API requests from users, tools, and internal components. (CNCF, 2024) Etcd: A lightweight, distributed key-value store that holds the entire cluster state. It is the source of truth for configuration, metadata, and cluster status. Scheduler assigns newly created Pods to suitable worker nodes based on resource requirements, constraints, affinity/anti-affinity rules, and other scheduling policies. Controller Manager runs a set of controllers that continuously monitor the cluster state and attempt to maintain the desired configuration. (e.g., Deployment Controller, Node Controller, Job Controller). Cloud Controller Manager (optional) is used in cloud environments to manage cloud-specific logic. On bare-metal, this component is usually not used or replaced by third-party integrations. (CNCF, October 20, 2024)

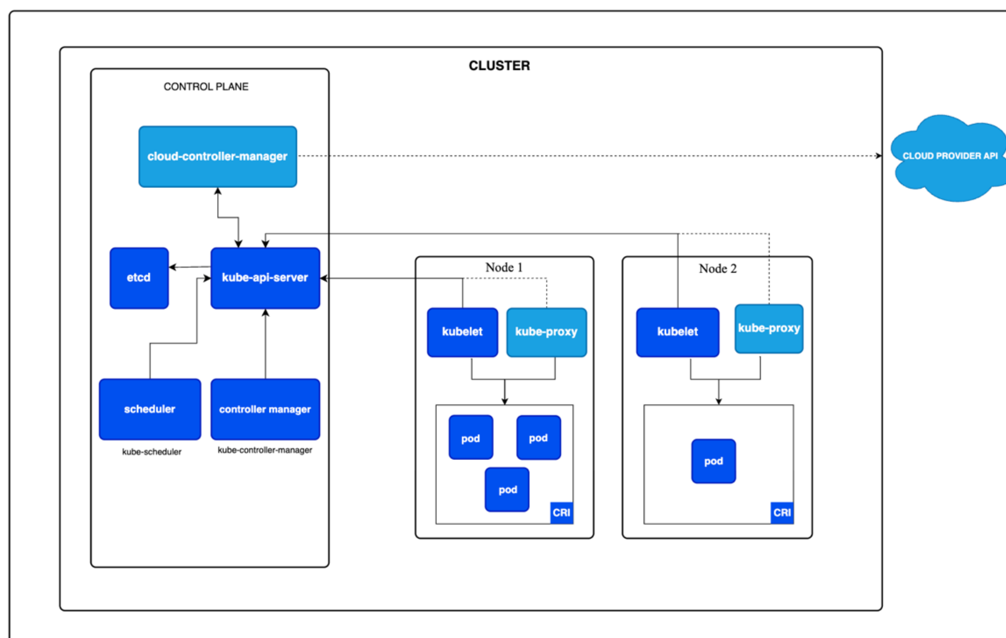

Figure 2. Cluster Architecture (CNCF, 2024)

3.1.2 Worker Node

In Kubernetes architecture, a worker node is a fundamental unit responsible for running application workloads in the form of containerized processes. Each node can be a physical machine or a virtual machine within a cluster and operates under the management of the control plane. While the control plane makes global decisions regarding the cluster (such as scheduling and scaling), it is the worker nodes that execute the tasks. These nodes host the necessary components to manage the execution environment, enforce cluster policies, and ensure application availability. A standard Kubernetes worker node contains three key components that enable it to fulfill its role: the kubelet, the container runtime, and kube-proxy (CNCF, October 20, 2024).

The *kubelet* is an agent that acts as the primary communication interface between the control plane and the node. It receives Pod specifications from the control plane and ensures that the containers defined within those Pods are running and healthy. It also collects and reports the status of Pods and the node back to the control plane. The *container runtime* is the software responsible for pulling container images and managing the lifecycle of containers. Kubernetes supports several runtimes that implement its Container Runtime Interface (CRI), including containerd, CRI-O, and previously Docker. The *kube-proxy* is the network component that maintains network rules and routes traffic to the appropriate Pods and services. It enables seamless communication between Pods within and across nodes, ensuring proper service discovery and load balancing (CNCF, October 20, 2024).

Worker nodes are often grouped together to form a scalable and resilient cluster. The orchestration layer ensures that workloads are distributed efficiently across these nodes, taking into account resource availability,

node health, and application requirements. In the event of a failure, Kubernetes can automatically reschedule workloads to healthy nodes, thereby promoting high availability. By abstracting the complexity of underlying infrastructure, worker nodes in Kubernetes provide the execution environment necessary for running modern, microservices-based applications. This abstraction allows developers to focus on defining application behavior while Kubernetes handles the orchestration and runtime management across a distributed cluster of nodes.

3.2 Key Concepts

To manage containers efficiently across a cluster, Kubernetes introduces several key building blocks. These components help define, scale, and organize workloads in a consistent and automated way. In the scope of this thesis, it is important to understand these basic concepts, as Kubernetes serves as the primary infrastructure environment for the services being deployed. This section introduces the most essential concepts which together form the foundation of Kubernetes.

3.2.1 Pods, Replica Sets, Deployments, Namespaces, Selectors

Kubernetes provides a declarative and highly automated model for managing containerized applications at scale. Among its core abstractions—Pods, ReplicaSets, Deployments, and Namespaces—lies an essential mechanism known as selectors, which enable dynamic and hierarchical control over these objects. Together, these components constitute the operational logic of Kubernetes clusters.

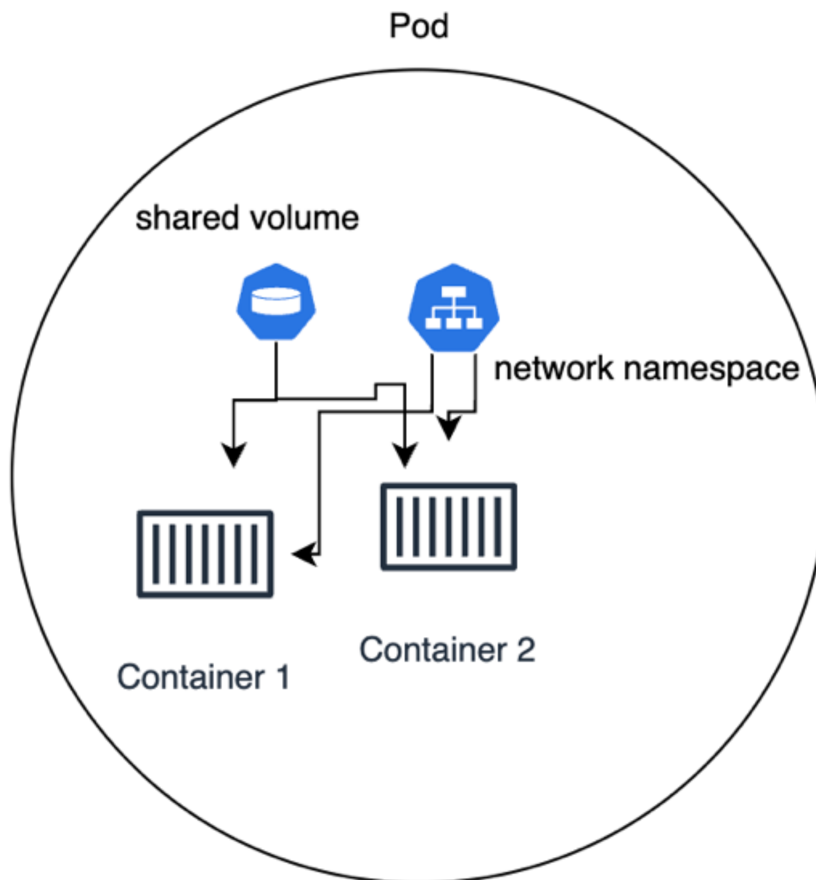

Figure 3. Pod internals

A *Pod* (see Figure 3) is the smallest and most basic deployable object in Kubernetes. It encapsulates one or more tightly coupled containers that share a network namespace and can mount shared storage volumes. All containers in a Pod are scheduled on the same node and can communicate via localhost. Pods are ephemeral by design and are generally not created directly by users in production settings; instead, their lifecycle is managed by higher-level controllers such as ReplicaSets and Deployments. Pods expose metadata—specifically, labels—that make them addressable and manageable by these controllers. (CNCF, April 7, 2024)

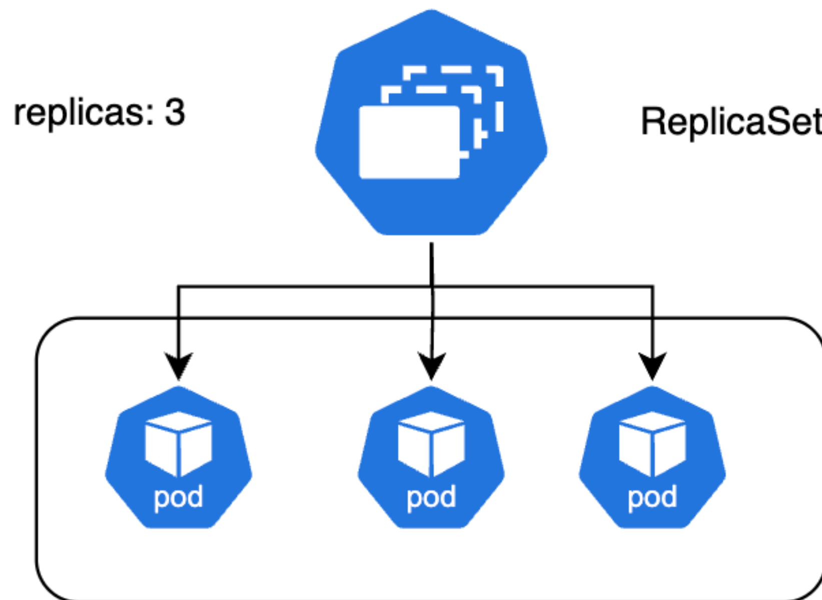

Figure 4. ReplicaSet

A *ReplicaSet* (see Figure 4) ensures that a specific number of Pod replicas are running concurrently. It continuously monitors the actual state of a set of Pods and adjusts it to meet the declared desired state. A key mechanism enabling this control is the label selector. A ReplicaSet identifies and manages Pods based on a label query, ensuring it only targets Pods that match certain criteria. This decoupling between controller and resource allows the system to scale, update, or delete Pods in a flexible and modular way. Without selectors, such fine-grained, dynamic association between Pods and their controllers would be impractical. (CNCF, April 15, 2025)

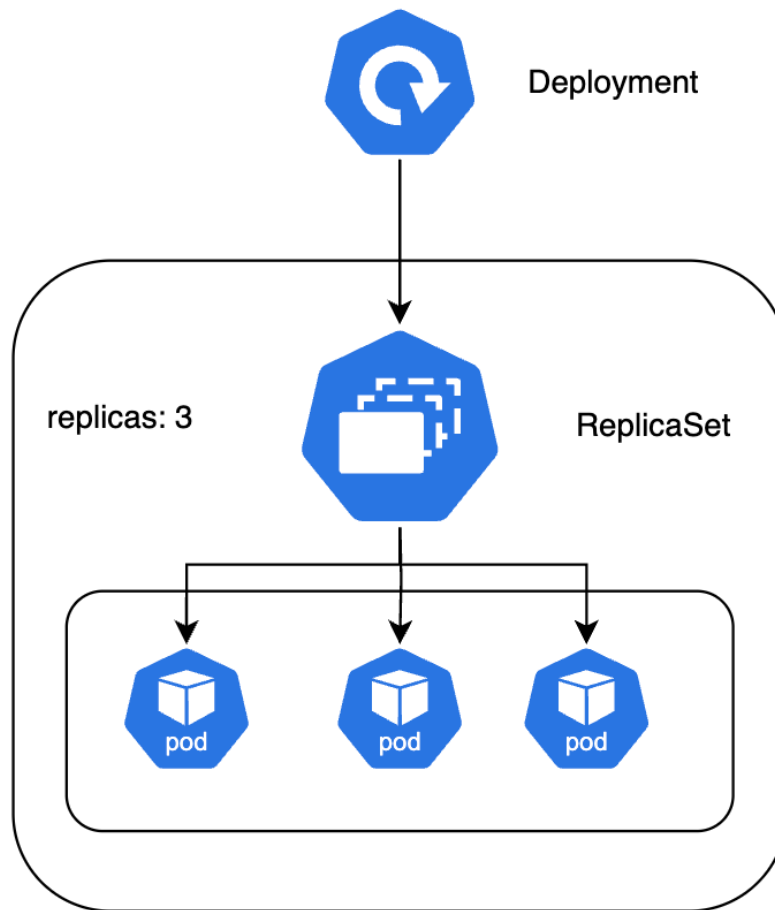

Figure 5. Deployment

A *Deployment* (see Figure 5) builds on the capabilities of ReplicaSets by introducing declarative management for application lifecycle events such as rolling updates, scaling, and rollback. The Deployment controller itself does not interact directly with Pods; instead, it manages ReplicaSets, which in turn manage Pods. Selectors play a critical role here as well: each Deployment specifies a selector to determine which ReplicaSets—and ultimately which Pods—it governs. This hierarchical chaining of selectors ensures a clean separation of responsibilities and enables automated orchestration of highly available applications across the cluster. (CNCF, March 27, 2025)

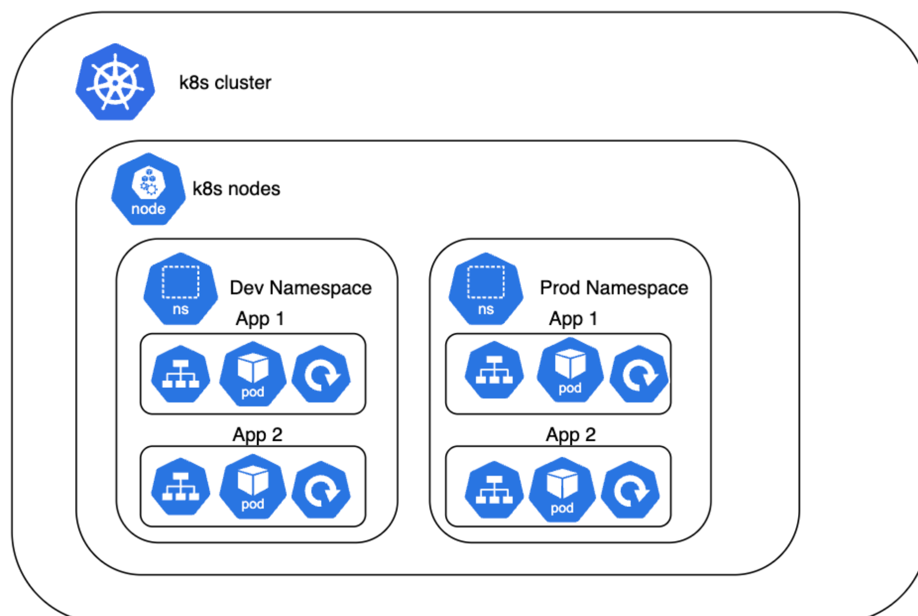

Figure 6. Namespaces

Namespaces (see Figure 6) provide a means to divide cluster resources between multiple users or teams, supporting multi-tenancy, resource quotas, and access control. Namespaces do not inherently restrict controller relationships, but selectors can be scoped to a namespace, enabling precise governance over which resources are managed by which controllers within a defined boundary. (CNCF, September 3, 2024)

Label selectors are fundamental to the Kubernetes control model. They are used to associate a controller—such as a ReplicaSet or Deployment—with the specific set of objects it should manage. This indirect relationship, mediated by metadata (labels), enables high modularity and loose coupling. (CNCF, July 30, 2024)

For instance, a ReplicaSet uses a selector to target all Pods with a certain label, ensuring that the right number of replicas are maintained. A Deployment uses a selector to manage the appropriate ReplicaSet(s), especially during rolling updates or rollbacks. Other resources, such as Services, also use selectors to determine which Pods they should route

traffic to. Selectors support two types of matching: equality-based (e.g., app=nginx) and set-based (e.g., env in (prod, staging)), which provide expressive power to manage complex application topologies. Without selectors, Kubernetes would lack a mechanism to dynamically bind resources to controllers, and the declarative model would not scale effectively.

3.3 Kubernetes Networking

3.3.1 ClusterIP, NodePort, Ingress, Ingress Controller, Load-Balancer

A Service in Kubernetes is an abstraction that defines a logical set of Pods and a policy by which to access them (see Figure 7), typically via a stable network endpoint. While Pods are ephemeral and can change IP addresses upon restart, Services ensure consistent communication by automatically routing requests to healthy Pods using selectors and labels. Services support different types such as ClusterIP, NodePort, LoadBalancer, and ExternalName, enabling internal and external access to applications deployed in the cluster.

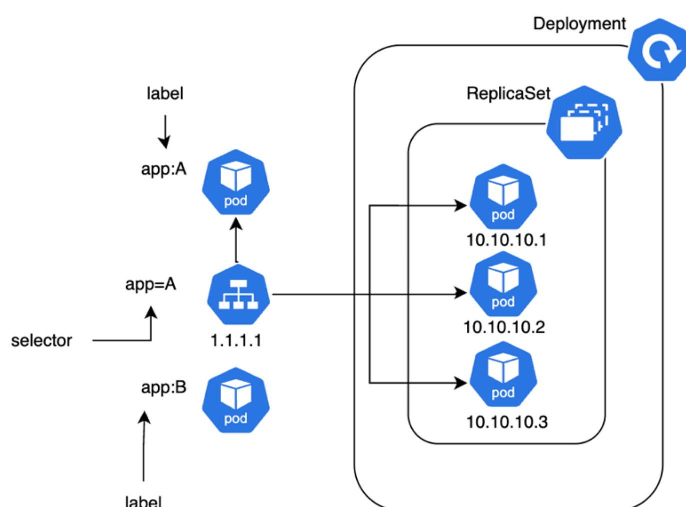

Figure 7. Label-Selector based trafic routing

The Kubernetes networking stack facilitates communication between internal components and external clients through a layered approach. The principal constructs at the core of this architecture are: *ClusterIP, NodePort, Ingress, Ingress Controller, and LoadBalancer*. These abstractions collectively enable the exposure, routing, and load balancing of applications both within and outside the cluster.

ClusterIP (see figure 8) is the Foundation of Internal Communication ClusterIP is the default service type in Kubernetes and represents the most basic level of service exposure. When a service is defined as ClusterIP, it receives a virtual IP address that is routable only within the cluster network. This allows workloads, such as pods in different namespaces or deployments, to communicate securely and efficiently without being accessible from the external network. This abstraction is essential for maintaining encapsulation and internal microservice communication patterns. The internal DNS system in Kubernetes resolves service names to their ClusterIP, enabling simple and stable service discovery mechanisms.

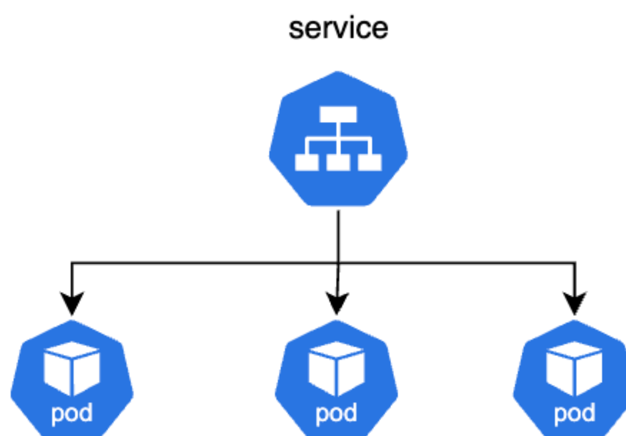

Figure 8. ClusterIP

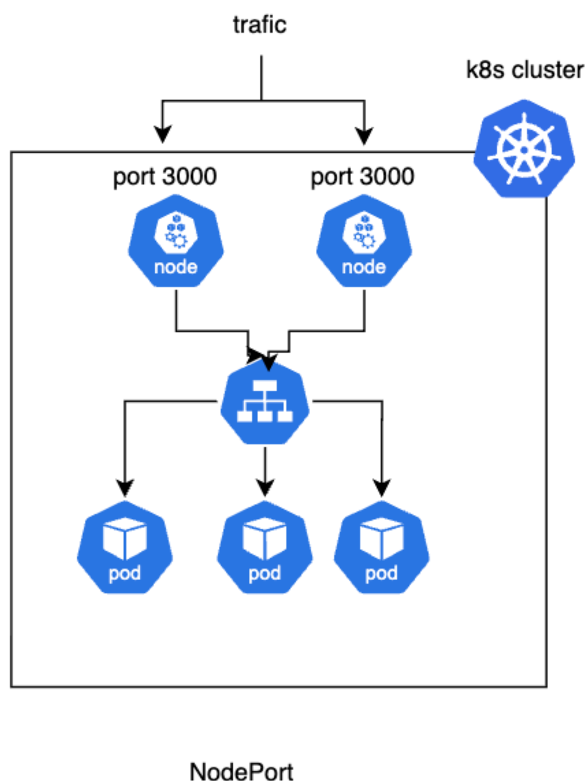

Figure 9. NodePort

While ClusterIP is sufficient for intra-cluster communication, applications often require exposure to external clients—for instance, for development, monitoring, or direct access to services. The NodePort service type (see figure 9) allows external access by opening a fixed TCP/UDP port on each node in the cluster. Incoming requests to this port on any node are forwarded to the corresponding service. However, NodePort is limited in terms of scalability and flexibility. It is bound to a specific port range (usually 30000–32767), and traffic distribution is rudimentary compared to more advanced load-balancing solutions. Therefore, while useful in certain cases, it is typically considered a low-level mechanism for exposing services. (CNCF, September 18, 2024)

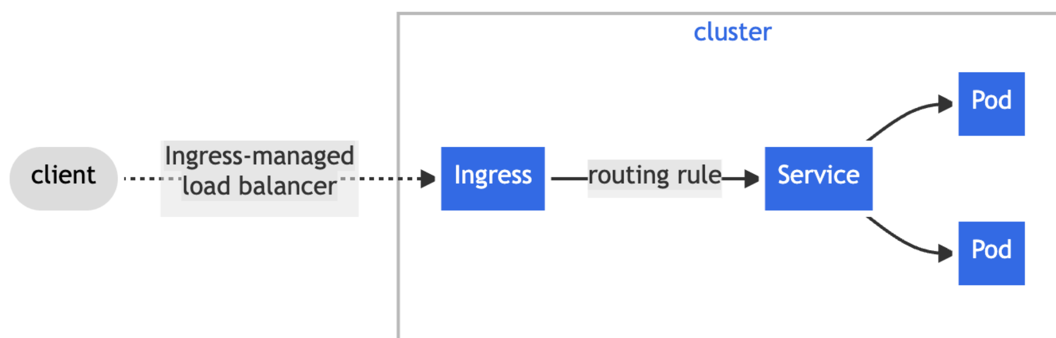

Figure 10. Ingress (CNCF, 2024)

The LoadBalancer service type (see figure 10) enhances Kubernetes ability to expose services externally by provisioning a cloud provider's native load balancer to route traffic to the service. This is accomplished via integration with the Kubernetes cloud controller manager. The LoadBalancer service automatically assigns a stable external IP address and routes traffic to the underlying pods. The cloud provider decides how it is load balanced. (CNCF, September 18, 2024)

Although LoadBalancer and NodePort provide basic service exposure, more complex routing requirements—such as domain-based routing, TLS termination, or multi-path dispatch—require the use of Ingress (see figure 10). The Ingress resource is a high-level abstraction that defines a set of rules for HTTP(S) traffic routing to multiple services based on hostnames and paths. This allows administrators and developers to expose many services through a single external endpoint, centralizing traffic control and reducing the number of required external load balancers. The power of Ingress lies in its declarative model, allowing infrastructure-as-code approaches to managing HTTP traffic. However, the Ingress object itself does not perform any networking function; rather, it acts as a contract to be implemented by an Ingress Controller. (CNCF, September 18, 2024)

An Ingress Controller is a runtime component that interprets and enforces the rules declared in Ingress resources. Common Ingress Controllers include NGINX, Traefik, and HAProxy, each capable of handling

request routing, TLS termination, request filtering, and more. The decoupling of the declarative interface (Ingress) from its implementation (Ingress Controller) enables flexibility and customization. Different controllers can be deployed to meet different performance, feature, or compliance requirements. The design also supports pluggability: organizations can implement custom controllers or choose open-source ones that align with their architectural and operational priorities. (CNCF, September 18, 2024)

3.4 Configuration and Secret Management

Kubernetes provides several ways to manage configuration data — through environment variables, ConfigMaps, and Secrets. Secrets and ConfigMaps are stored in etcd and transmitted to Pods over a network connection, both of which are secured by the Kubernetes API server (see Figure 11). Kubernetes also supports encryption at rest for Secret data, adding an extra layer of security.

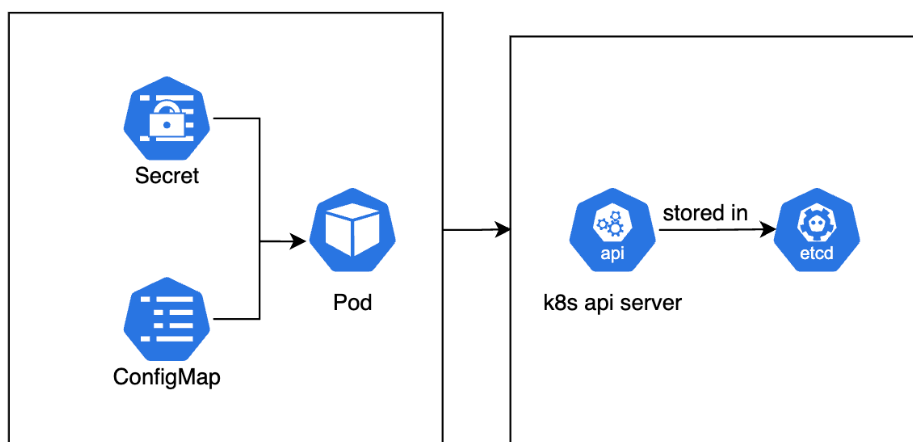

Figure 11. ConfigMaps and Secrets

3.4.1 ConfigMaps

A *ConfigMap* is a Kubernetes API resource designed to store non-sensitive configuration data as key-value pairs. This data can be injected into Pods in various ways, including as environment variables, command-line arguments, or mounted as files within a volume. The primary purpose of a ConfigMap is to separate configuration details from the application's container image, promoting greater portability and flexibility across different environments.

This design supports easier updates and deployments without the need to rebuild container images for environment-specific settings. It is important to note that ConfigMaps are *not intended for sensitive information*, as they do not offer encryption or secrecy. For storing confidential data, Kubernetes provides the *Secret* resource, or one may employ external tools that offer enhanced security and encryption mechanisms. (CNCF, June 16, 2021)

3.4.2 Secrets

A *Secret* in Kubernetes is a specialized resource designed to securely store sensitive information, such as passwords, tokens, or cryptographic keys. By using Secrets, confidential data can be managed separately from Pod specifications or container images, thereby reducing the risk of accidental exposure within application code or deployment files. Since Secrets are created independently from the Pods that consume them, the likelihood of leaking sensitive information during routine operations—such as creating, inspecting, or modifying Pods—is minimized.

Furthermore, Kubernetes and cluster applications can enforce additional security measures, such as avoiding persistent storage of secret data. While Secrets and ConfigMaps are functionally similar in that they both store key-value pairs and provide configuration data to applications, Secrets are specifically intended for confidential content and are handled with greater care to maintain data privacy and integrity. It is important

to note that Secrets, by default, are stored in plaintext within the API server's backing datastore, etcd. This means that anyone with direct access to etcd or with sufficient API permissions can view or alter these Secrets. Moreover, any user authorized to create Pods in a namespace can potentially gain access to all Secrets in that namespace, either directly or through indirect means such as creating a Deployment that mounts the Secret. (CNCF, June 16, 2021)

3.5 Kubernetes Storage

The key concepts related to storage in Kubernetes are PVs, PVCs, CSI and StorageClass. To manage persistent storage, Kubernetes introduces the Persistent Volume (PV) and Persistent Volume Claim (PVC) abstractions. A *Persistent Volume (PV)* represents a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using a StorageClass. *A Persistent Volume Claim (PVC)* is a request for storage by a user, which specifies the desired capacity and access mode. This decoupling of storage provisioning from storage consumption facilitates greater flexibility and portability across environments. (CNCF, March 25, 2025)

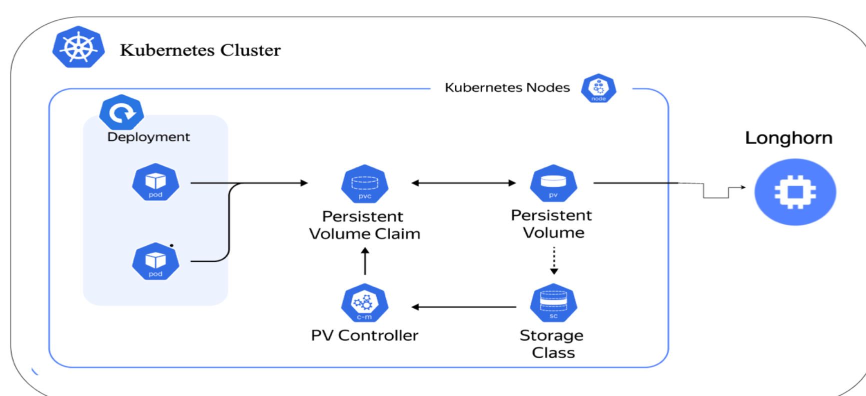

Figure 12. Dynamic storage provisioning with Longhorn

The StorageClass resource defines the provisioner, parameters, and reclaim policy for dynamic volume provisioning (see figure 12). It acts as

a template for creating PVs automatically when PVCs are issued. In this work, Longhorn is employed as the storage provider. Longhorn is a cloud-native, distributed block storage system built specifically for Kubernetes. It integrates seamlessly via the Container Storage Interface (CSI*)*, a standard designed to enable portable and extensible volume drivers across different orchestrators and storage backends. Container Storage Interface enables third-party storage vendors to develop plugins without requiring changes to the core Kubernetes codebase. CSI-compliant drivers, such as Longhorn, manage the lifecycle of volumes—provisioning, attachment, detachment, and deletion—using standardized APIs. (CNCF, March 25, 2025)

3.6 Key Scheduling Concepts

In Kubernetes, taints and tolerations is a core mechanism for control over the placement of workloads across a cluster. Conceptually, they implement a negative scheduling constraint, allowing cluster operators to restrict certain nodes from accepting arbitrary workloads unless explicitly permitted. A *taint* is a property assigned to a node, signaling that the node is reserved for specific workloads and should repel all pods by default. This repulsion is realized through the use of a key-value pair and an associated effect (NoSchedule, PreferNoSchedule, or NoExecute). For example, a node tainted with key=dedicated, value=infra, effect=NoSchedule will reject all pods unless they carry a corresponding toleration. A toleration, by contrast, is applied to a pod. It acts as a declarative allowance, signaling that the pod is capable of “tolerating” the taint and is thus eligible for scheduling onto nodes bearing that specific taint. Importantly, tolerations do not guarantee scheduling onto a tainted node; rather, they enable the scheduler to consider the node as a viable candidate. This taint–toleration model supports a variety of operational goals:

•Isolation of infrastructure components (e.g., system daemons, logging agents) on dedicated nodes.

•Enforcement of resource guarantees by repelling best-effort workloads.

•Implementation of eviction policies via the NoExecute effect, ensuring that tainted nodes are cleared of incompatible workloads dynamically.

As part of Kubernetes extensible scheduling architecture, taints and tolerations operate at the filtering stage of the scheduler. Nodes whose taints are not tolerated by a pod are excluded from consideration, thus influencing the set of candidate nodes without dictating a final scheduling decision. In conclusion, taints and tolerations form a declarative, node-centric scheduling primitive that balances flexibility and control. They enable administrators to enforce policy-driven workload placement, ensuring that node resources are allocated according to organizational or infrastructural constraints, while preserving Kubernetes' general-purpose scheduling behavior. (CNCF, February 18, 2025)

3.7 Operators, CRDs

A Kubernetes Operator is a method for packaging, deploying, and managing Kubernetes-native applications. These applications not only run on Kubernetes but are also managed using the Kubernetes API and standard tooling such as kubectl (command line tool for communication with cluster). An Operator functions as an application-specific controller that extends the Kubernetes API to automate the creation, configuration, and lifecycle management of complex applications. Unlike basic controllers, Operators embed domain-specific logic to carry out tasks that would otherwise require manual intervention by a human operator. In Kubernetes, controllers are responsible for maintaining the desired state of the cluster by constantly comparing it to the actual state and taking corrective action when discrepancies arise. An Operator is a custom controller that leverages Custom Resources (CRs) to manage the

lifecycle of an application and its components. Users define high-level configurations in CRs, and the Operator interprets these directives, translating them into low-level operations based on best practices in its codebase. (RedHat, May 11, 2022)

Custom Resource Definitions (CRDs) allows cluster administrators to define new resource types beyond Kubernetes built-in objects (like Pods, Services, etc.). Once registered, these custom resources can be manipulated via kubectl, API requests, or Helm—just like native objects. (CNCF, October 31, 2024) For example, a PostgreSQL Operator may introduce a PostgresCluster resource, which simplifies the creation, backup, and scaling of PostgreSQL databases.

4 LOAD BALANCERS

Load balancing is a fundamental component in the design of scalable and highly available systems. It refers to the process of distributing incoming network traffic across a pool of backend servers or services to optimize resource use, maximize throughput, reduce response time, and ensure fault tolerance. As systems grow in complexity and scale, manual routing of requests becomes infeasible; automated load balancing is therefore essential. Motivation and Use Cases Without load balancing, systems are prone to bottlenecks and single points of failure.

4.1 Load Balancing Overview

A well-designed load balancer ensures that not a single backend instance is overwhelmed, enables horizontal scaling of services, provides redundancy and failover support, enables blue-green and canary deployments through routing policies. In Kubernetes, load balancing is particularly important for exposing services to external clients, distributing traffic among pods, and integrating with infrastructure-level networking components.

Different algorithms are used depending on the performance goals and characteristics of the backend services: Round Robin distributes requests evenly across the pool in a circular order. It is simple and stateless, but not optimal for uneven workloads. Least Connections sends traffic to the server with the fewest active connections. IP Hashing routes requests based on a hash of the client IP, maintaining session stickiness. Random selects a backend at random. Weighted Round Robin/Least Connections accounts for differing server capacities by assigning weights. These algorithms may be implemented at Layer 4 (transport) or Layer 7 (application), depending on the load balancer's capabilities.

4.2 MetalLB

In bare-metal environments, where no cloud provider is available to automatically provision load balancers, Kubernetes supports alternative solutions. These include MetalLB, an open-source load balancer implementation for bare-metal clusters that assigns IPs via Layer 2 (ARP) or Layer 3 (BGP) routing. (MetalLB, n.d.) Such solutions bridge the gap between cloud-native service types and on-premise infrastructure, enabling the use of LoadBalancer semantics even without cloud-native APIs.

MetalLB consists of two core components: a controller and a speaker. The controller is a Kubernetes controller that integrates with the Kubernetes API server. It monitors Service resources and assigns IP addresses from a pre-configured address pool. The speaker is wokrload that runs on each node and is responsible for advertising the assigned IPs via network protocols such as ARP (Layer 2) or BGP (Layer 3). (MetalLB, n.d.)

In Layer 2 Mode (ARP-based advertisement) MetalLB uses ARP (for IPv4) or NDP (for IPv6) to advertise a virtual IP address from one node at a time. Only a single node “claims” the IP by responding to ARP requests on the local network. If the node holding the IP fails or becomes unready, MetalLB withdraws the advertisement and another node takes over. This process ensures high availability via fast failover, typically within a few seconds. Since only one node owns the IP at any given time, all external traffic flows through that node. MetalLB in Layer 2 mode does not provide external load balancing across multiple nodes. Thus, Layer 2 mode guarantees availability but not load distribution at the ingress point. Regardless of how traffic enters the cluster, kube-proxy handles service-level balancing internally. It maintains iptables/ipvs rules to distribute traffic from a Service to its backing pods, across all nodes. Even if external traffic enters through a single ingress node, kube-proxy may forward requests to backend pods running on other nodes. (MetalLB, n.d.)

In BGP mode, MetalLB speaks the Border Gateway Protocol with upstream routers or switches. Multiple nodes can simultaneously advertise the same IP address, and if the upstream network supports ECMP (Equal-Cost Multi-Path routing), it can distribute traffic among them. BGP inherently handles path withdrawals, so failover is also fast and reliable. BGP enables true load balancing of incoming traffic across all advertising nodes, assuming appropriate ECMP configuration in the network infrastructure. (MetalLB, n.d.)

In the context of a small-scale, internal Kubernetes deployment, the use of MetalLB in Layer 2 mode represents an efficient approach to providing LoadBalancer services in bare-metal environments. This operational mode leverages standard Ethernet protocols (ARP for IPv4, NDP for IPv6) to advertise service IPs directly on the local subnet, thereby eliminating the need for complex external routing infrastructure or BGP configuration. Layer 2 mode exhibits several properties that make it well-suited for internal services in constrained or isolated environments. First, its simplicity and low operational overhead reduce the burden on cluster administrators, making it an accessible solution for teams without deep networking expertise. Second, failover is effectively managed through MetalLB’s speaker election mechanism, ensuring high availability without requiring additional hardware or protocols. While Layer 2 mode inherently lacks load distribution across multiple ingress nodes, Kubernetes internal mechanisms such as kube-proxy help mitigate single-node saturation by redistributing traffic at the pod level once it enters the cluster.

5 MONITORING

Monitoring is a process of gathering, analyzing, summarizing, and visualizing real-time quantitative metrics from a system—such as the number and types of queries, error rates, response times, and server uptimes (Beyer et al., 2016). Monitoring serves several essential purposes in system management. It enables the analysis of long-term trends—such as tracking database growth or user engagement metrics—and supports comparisons over time or between experimental configurations. For instance, engineers might evaluate the performance of different database technologies or measure the impact of adding cache nodes on response times. Dashboards, typically incorporating key performance metrics like the "four golden signals" (latency, traffic, errors, and saturation), offer a visual summary of system health. Additionally, monitoring facilitates ad hoc investigations, helping teams correlate issues like sudden latency increases with other system events. Beyond technical diagnostics, monitoring can support business analytics and security investigations. (Beyer et al., 2016).

5.1 Metrics

Metrics are quantitative measurements used to observe and evaluate the performance and health of systems. In the context of distributed applications and infrastructure, metrics provide insights into aspects such as resource usage, request latency, error rates, and throughput. They form the foundation for effective monitoring, alerting, and capacity planning. By continuously collecting and analyzing metrics, system operators can detect issues early, improve reliability, and optimize system behavior. For example, a web server may expose metrics such as request latency or request count, while a database may track active connections or query execution times. These metrics help identify and diagnose performance issues—if a web application becomes slow under

heavy load, analyzing request count metrics can reveal bottlenecks, guiding actions such as scaling the number of instances.

In our system Prometheus was used for metrics collection and Grafana for visualization using dashboards. Prometheus is an open-source systems monitoring and alerting toolkit originally developed by SoundCloud. It collects and stores metrics as time series data—each data point includes a timestamp and optional key-value pairs known as labels, which enable a rich, multi-dimensional data model. Prometheus provides a powerful query language, PromQL, that allows users to aggregate and analyze these metrics flexibly. It operates without reliance on distributed storage; each Prometheus server is a self-contained unit. Metrics are typically scraped (pulled) over HTTP from instrumented targets, though pushing is also possible via the PushGateway. Targets can be registered either statically or discovered automatically using service discovery mechanisms. Prometheus integrates easily with visualization tools, offering multiple options for graphing and dashboarding. (Prometheus, n.d.)

Grafana Open-Source Software (OSS) is a powerful observability platform that enables users to query, visualize, alert on, and explore metrics, logs, and traces from diverse data sources. Through its extensive plugin architecture, Grafana supports integrations with time-series databases such as Prometheus and CloudWatch, logging systems like Loki and Elasticsearch, and relational or NoSQL databases including PostgreSQL. By leveraging these data sources, Grafana OSS facilitates the construction of live dashboards, offering insightful and interactive visualizations to support monitoring, troubleshooting, and decision-making in real time. (Grafana Labs, n.d.)

5.2 Logging

Logging is a foundational component of software systems, enabling developers, operators, and analysts to observe and interpret system behavior over time. Logs offer a chronological trace of events, including errors, warnings, and informational messages generated during application execution. They serve as critical tools for debugging, performance analysis, security auditing, and maintaining system reliability. In the context of complex, distributed architectures, comprehensive logging is indispensable for diagnosing faults, identifying anomalies, and supporting operational transparency.

The logging pipeline (see Figure 13) demonstrates a modular approach to centralized log collection in a Kubernetes environment. Each service running in the cluster produces logs, which are collected locally by lightweight Fluent Bit agents. These agents are deployed on each node and are responsible for forwarding the logs to a central Fluentd instance. Fluentd acts as an aggregator and processor, where logs can be filtered, parsed, and enriched before being routed to the backend. (CNCF, May 29, 2025) The processed logs are then sent to Loki, a log aggregation system designed by Grafana Labs. Loki indexes and stores the logs, making them searchable and ready for visualization and analysis through Grafana dashboards. This workflow ensures reliable, structured, and resource-efficient log management across the entire system.

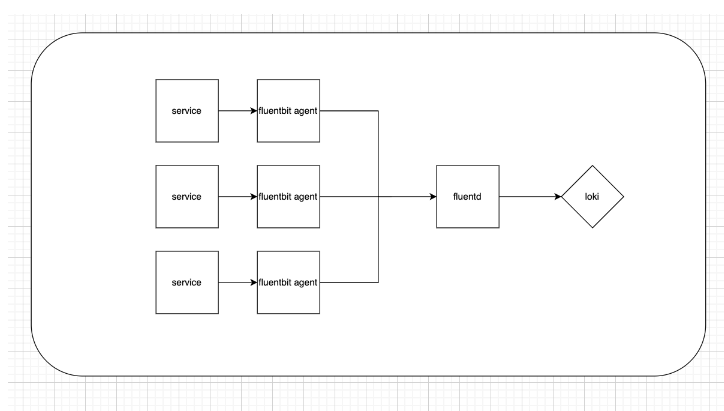

Figure 13. Logging workflow

6 AUTOMATION AND TEMPLATING

As infrastructure grows in complexity, manual management becomes unsustainable and difficult to reproduce. Automation tools address these challenges by enabling repeatable, consistent, and scalable workflows for configuring systems and deploying applications. In Kubernetes environments, tools like Ansible and Helm make it possible to automate everything from cluster setup to application deployment, bringing structure and control to infrastructure management.

6.1 Ansible

Ansible is an open-source automation tool known for its simplicity and minimal setup requirements. One of its key advantages is that it is agentless—no additional software needs to be installed on the managed hosts. The only prerequisites are a Python interpreter and pre-configured SSH key access to the target machines. One of the most important concepts in Ansible is the playbook (see Figure 14). A playbook is a file that defines a series of tasks to be executed on managed hosts, describing the desired state of a system in a simple, human-readable format. (Ansible, May 15, 2025a)

Instructions that are defined in the task are executed by the targets in an inventory file (see Figure 15). In Ansible, an inventory is a file or script that specifies the hosts and their groupings for task execution. It serves as the central reference point that tells Ansible which machines to manage. Inventories can be defined in formats such as INI, or YAML. Grouping hosts allows users to efficiently target specific subsets of infrastructure during automation tasks. (Ansible, May 15, 2025b)

```yaml
---
- name: Prepare Kubernetes nodes for Longhorn
  hosts: all
  become: yes

  tasks:
    - name: Install open-iscsi and NFS client
      apt:
        name:
          - open-iscsi
          - nfs-common
        state: present
        update_cache: true

    - name: Enable and start iscsid service
      service:
        name: iscsid
        enabled: true
        state: started

    - name: Install system utilities
      apt:
        name:
          - bash
          - curl
          - util-linux
          - grep
          - gawk
          - lsof
        state: present

    - name: Ensure findmnt and lsblk are available
      shell: |
        which findmnt >/dev/null && which lsblk >/dev/null
      changed_when: false

    - name: Check filesystem type of root partition
      command: findmnt -n -o FSTYPE /
      register: fs_check
      changed_when: false

    - name: Fail if filesystem is not ext4 or xfs
      fail:
        msg: "Unsupported filesystem type for Longhorn: {{ fs_check.stdout }}"
      when: fs_check.stdout not in ['ext4', 'xfs']

    - name: Print success message
      debug:
        msg: "Node is ready for Longhorn (Filesystem: {{ fs_check.stdout }})"

```

Figure 14. Longhorn requirements setup

```ini
[master]
master1 ansible_host=10.138.10.18

[workers]
worker1 ansible_host=10.138.10.17

[all:vars]
ansible_user=ubuntu
```

Figure 15. inventory.ini used for the setup

6.2 Helm

Helm is a package manager for Kubernetes that simplifies the deployment and management of applications within a cluster. It allows users to define, install, and upgrade even the most complex Kubernetes applications using a packaging format called charts. These charts are reusable and configurable, making Helm especially useful for automating deployments and maintaining consistency across different environments. (CNCF, n.d.)

By abstracting the complexity of writing raw Kubernetes manifests, Helm reduces manual effort and the risk of human error. It supports versioning, rollback, and dependency management, enabling teams to manage infrastructure as code more effectively. Additionally, Helm's templating system allows for reusable and customizable configurations, making deployments more consistent across environments.

7 AUTHENTICATION

In modern software systems, authentication is a critical security mechanism that verifies the identity of users or services before granting access to resources. It ensures that only legitimate users can interact with protected systems, safeguarding sensitive data and operations from unauthorized access.

7.1 OpenID Connect (OIDC)

Modern authentication systems rely on standardized protocols that define how identities are verified and how secure tokens are exchanged. OpenID Connect is a widely adopted authentication protocol built on top of the OAuth 2.0 framework (IETF RFC 6749 and 6750). It streamlines the process of verifying a user's identity by leveraging an Authorization Server and enables the secure retrieval of user profile information in a consistent, RESTful way (OpenID n.d.).

OIDC (OpenID Connect) authentication enables users to sign in to one application and gain access to another without creating separate credentials. For instance, when a user visits a news website and chooses to sign up using Facebook instead of registering a new account, they are utilizing OIDC authentication. In this scenario, Facebook acts as the OpenID Provider (OP), managing the authentication process and requesting the user's permission to share specific information—like their profile details—with the news site, which functions as the Relying Party (RP). (Microsoft, n.d.-a)

The whole flow of the authentication is shown in Figure 16. The end user accesses a website or web application through their browser. Upon selecting the sign-in option, the user enters their username and password.

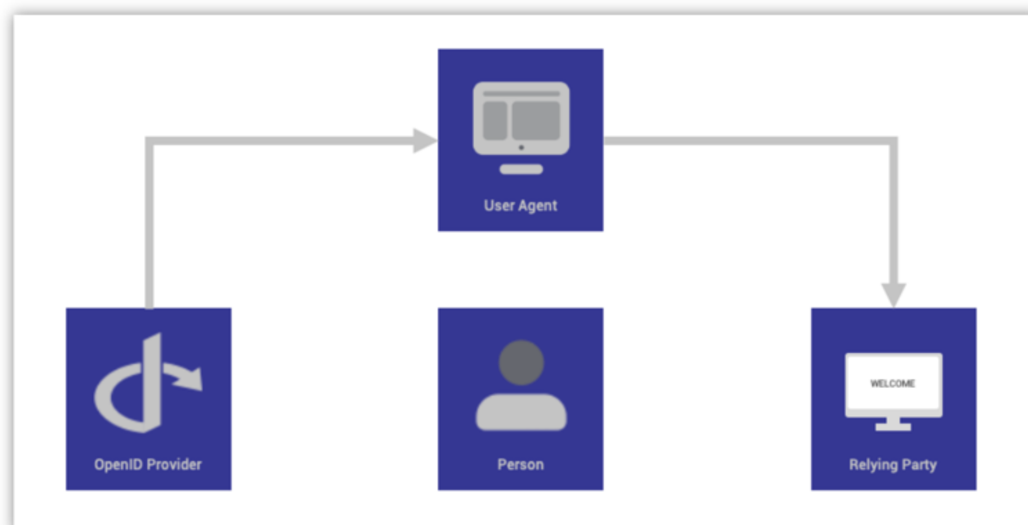

Figure 16. Authentication process with OIDC (openid.net, n.d.)

The Relying Party (RP), also known as the client application, sends an authentication request to the OpenID Provider (OP). The OP verifies the user's credentials and obtains the necessary authorization. After successful authentication, the OP returns an ID Token (and typically an Access Token) to the RP. The RP can use the Access Token to make a secure request to the UserInfo Endpoint on the OP. The UserInfo Endpoint responds with verified identity details (claims) about the end user. (OpenID, n.d.)

7.2 Identity Providers

Identity Providers (IdPs) are systems responsible for authenticating users and issuing identity information to relying parties (RPs), such as web applications or services. In modern infrastructure IdPs play a crucial role in centralizing authentication, enforcing access control, and integrating with Single Sign-On (SSO) and federation standards like OpenID Connect or SAML. Identity Providers can be enterprise-grade solutions, open-source platforms, or cloud-native services. In this section, we consider two popular IdPs commonly used in private, cloud-free environments: Active Directory and Keycloak.

Active Directory (AD) is a directory service developed by Microsoft for Windows domain networks. It is widely adopted in enterprise environments and provides a centralized mechanism for authenticating and authorizing users and devices. Key features of Active Directory include:

- LDAP support with AD as LDAP-compliant directory, allowing integration with various identity-aware applications (Microsoft, n.d.-b).
- Kerberos-based authentication to authenticate users securely. Group Policy management for policies across users and computers (Microsoft, n.d.-b).
- SSO support with AD serving as an IdP in federated environments using ADFS (Active Directory Federation Services) or integration with OpenID Connect providers via extensions (Microsoft, n.d.-b).

Active Directory is often used in on-premises infrastructure and integrates well with Windows-based systems, but it can also be used in mixed environments with Linux systems via LDAP or Kerberos. In our infrastructure, Active Directory is maintained and administered by the organization's cybersecurity department. Within the scope of this thesis, it serves as an external user directory integrated with Keycloak. Keycloak is configured to federate users from Active Directory, allowing the system to import and authenticate users based on existing enterprise credentials. This setup ensures centralized identity management while enabling flexible role-based access control through Keycloak.

Keycloak is an open-source Identity and Access Management (IAM) solution developed by Red Hat. It provides SSO, user federation, identity brokering, and support for standard protocols such as OpenID Connect, OAuth 2.0, and SAML 2.0. Keycloak is particularly suited for cloud-native and containerized environments due to its flexibility and RESTful management API. Keycloak features a web-based admin interface for man-

aging users, roles, and clients, supports Single Sign-On and identity federation, allowing users to authenticate once and access multiple applications. It also integrates with existing directories like LDAP and Active Directory and enables access control through role-based permissions. (Keycloak, n.d.) In our system, Keycloak is managed and maintained by the organization's cybersecurity department. As part of this arrangement, the administrative account and preconfigured LDAP connection credentials (login and password) for integrating user import from Active Directory were set up in advance.

8 DATAHUB

DataHub is a data catalog platform that simplifies metadata management, data discovery, and governance. It empowers users to explore and understand datasets, monitor data lineage, perform profiling, and define data contracts. Designed to support fast-evolving data environments, DataHub provides developers with extensibility and data professionals with tools to maximize the value of organizational data. (DataHub, n.d.)

There are several concepts that are crucial for understanding how DataHub works. DataHub uses connectors and ingestions for connection and data sync. A connector is a plugin that allows DataHub to establish a connection with a particular data source. Ingestion is a process of data retrieval, it is configured using recipe. A recipe is a set of rules, such as source (data source) and sink (ingestion destination). (Figure 17)

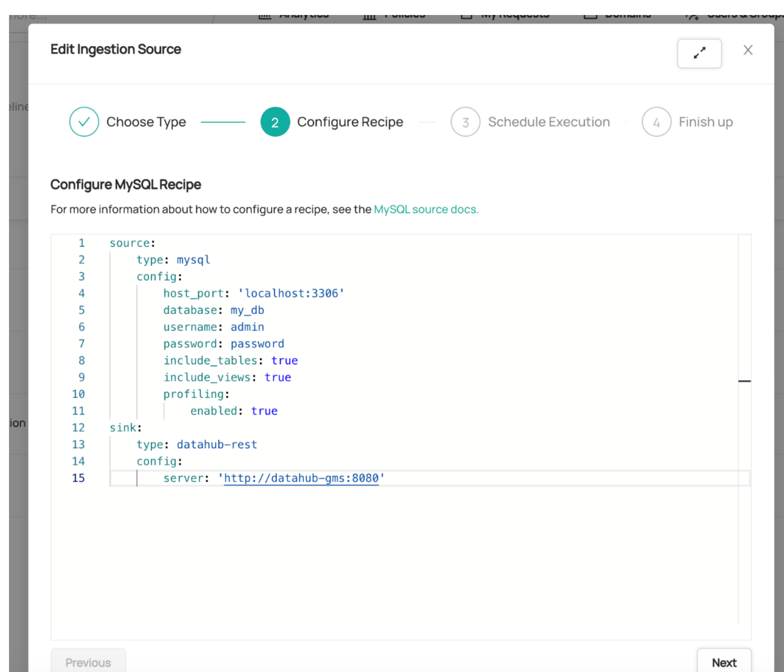

Figure 17. Recipe configuration (docs.datahub.com)

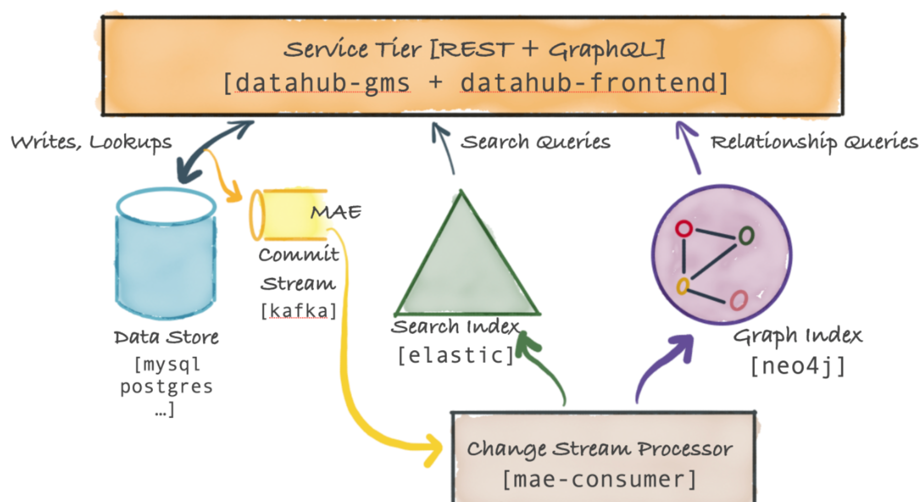

Figure 18. DataHub architecture (docs.datahub.com)

DataHub architecture is shown in Figure 18. The service tier (datahub-gms + datahub-frontend) is the entry point for users and applications, supporting both REST and GraphQL APIs. It handles user interactions, metadata ingestion, and query requests. Data store persists metadata in a traditional database (e.g., MySQL or PostgreSQL). All write and lookup operations go through this layer. Any changes to metadata are emitted as Metadata Audit Events (MAEs) to a Kafka topic in a commit stream component. This enables asynchronous processing and system decoupling. Change Stream Processor (mae-consumer) listens to the Kafka stream and updates downstream indices accordingly, ensuring that the search and relationship layers reflect the latest metadata state. Search Index (Elasticsearch/Opensearch) receives updates from the change stream and serves search queries, enabling fast and flexible text-based metadata discovery. Graph Index (Neo4j) also updated by the stream processor, so it is possible to build a specific snapshot of the graph in time by replaying MAEs up to that point. (DataHub, n.d.)

9 OBJECT STORAGE (MINIO)

The object storage is a type of data storage architecture designed for managing unstructured data by organizing it into units called objects. Each object contains the data itself, associated metadata, and a unique identifier, enabling straightforward access and retrieval by applications. This approach uses a flat structure, rather than a traditional hierarchical file system, making it ideal for scalable and distributed storage environments. (Google, n.d)

The object storage keeps all the data blocks of a file bundled together as a single object, along with its metadata and a unique identifier, and stores it within a storage pool. When accessing data, the system uses this identifier and metadata to locate the desired object—such as an image or audio file. Access to these objects is typically handled via RESTful APIs or over HTTP/HTTPS, allowing users to efficiently query metadata and retrieve data. Because objects reside in a global storage pool, finding the right data is quick and scalable. The flat architecture also makes it easy to expand storage capacity by simply adding more devices, even across various locations. Thanks to its scalability and flexibility, object storage has become a preferred solution for managing unstructured data in modern cloud environments. (Google, n.d)

Many object storage solutions provide their own APIs to interact with data, such as OpenStack Swift, Azure Blob Storage API, or Google Cloud Storage APIs. However, the Amazon S3 API has become the de facto standard in the industry due to its widespread adoption and ecosystem support. As a result, most modern object storage systems like MinIO or Ceph aim for S3 API compatibility to ensure interoperability with popular tools and cloud-native applications. MinIO is an object storage platform that offers full compatibility with the Amazon S3 API, supporting all essential S3 features. It is designed for flexibility, capable of running across a wide range of environments including public and private clouds,

bare-metal servers, container orchestration platforms, and edge infrastructure. (MinIO, n.d) The lightweight design of MinIO and its ability to run without complex dependencies make it ideal for managing bare metal infrastructure.

10 SYSTEM ARCHITECTURE

It is essential to understand the system’s workflow, beginning with the underlying infrastructure. The system architecture is depicted on the Figure 19. Longhorn is deployed across all Kubernetes nodes to provision and manage persistent storage. MinIO, running outside the Kubernetes cluster, acts as the backup target for Longhorn volumes.

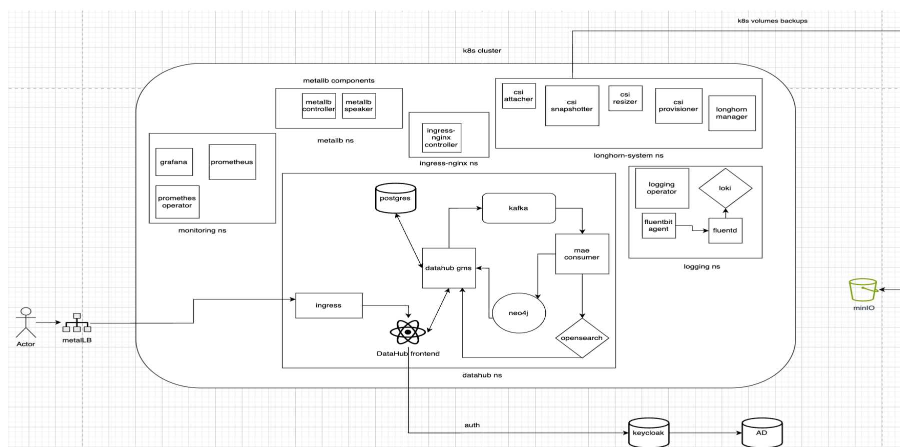

Figure 19. System Architecture

With the Longhorn storage class in place, we proceed to deploy the Prometheus Operator, Loki, and the Logging Operator. The Prometheus Operator collects metrics from all nodes, which are visualized through Grafana, while Loki handles centralized log access. The Logging Operator deploys Fluent Bit agents on each node to collect logs, which are routed through Fluentd for filtering before being forwarded to Loki.

To expose internal services externally, an NGINX Ingress Controller is deployed, enabling centralized traffic routing. MetalLB is configured in Layer 2 mode, with speaker components running on each node to provide failover. DataHub components are deployed in a dedicated namespace, with frontend access exposed via a MetalLB-assigned IP and secured through Keycloak using OpenID Connect (OIDC). Keycloak authenticates users via integration with Active Directory. Data received by the frontend is sent to Kafka and persisted across Neo4j, PostgreSQL, and OpenSearch.

11 SYSTEM IMPLEMENTATION

11.1 Kubernetes Cluster Setup with Ansible

The base infrastructure consists of a multi-node Kubernetes cluster deployed on bare-metal servers using Ansible. A dedicated Ansible playbook (see Figure 20) automates the operating configuration and dependencies installation.

```yaml
- name: Kubernetes Cluster Setup
  hosts: all
  become: true
  tasks:
    - name: Disable swap
      ansible.builtin.command: swapoff -a
      when: ansible_swaptotal_mb > 0

    - name: Comment out swap in /etc/fstab
      ansible.builtin.replace:
        path: /etc/fstab
        regexp: '^([^#].*swap)'
        replace: '# \1'

    - name: Enable br_netfilter module
      ansible.builtin.shell: |
        modprobe br_netfilter
        echo 'br_netfilter' > /etc/modules-load.d/k8s.conf
        echo 'net.bridge.bridge-nf-call-ip6tables = 1' >> /etc/sysctl.d/k8s.conf
        echo 'net.bridge.bridge-nf-call-iptables = 1' >> /etc/sysctl.d/k8s.conf
        sysctl --system

    - name: Enable IPv4 forwarding
      ansible.builtin.sysctl:
        name: net.ipv4.ip_forward
        value: '1'
        sysctl_set: true
        state: present
        reload: yes

    - name: Install required packages
      apt:
        name:
          - apt-transport-https
          - ca-certificates
          - curl
          - gnupg
          - lsb-release
        state: present
        update_cache: yes

    - name: Install containerd
      apt:
        name:
          - containerd
        state: present
        update_cache: yes

    - name: Configure containerd
      ansible.builtin.shell: |
        mkdir -p /etc/containerd
        containerd config default > /etc/containerd/config.toml
        sudo sed -i '/SystemdCgroup/s/false/true/' /etc/containerd/config.toml
        systemctl restart containerd
        systemctl enable containerd

    - name: Create keyrings directory
      ansible.builtin.file:
        path: /etc/apt/keyrings
        state: directory
        mode: '0755'

    - name: Add Kubernetes apt key
      ansible.builtin.shell: |
        curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.32/deb/Release.key | sudo gpg \
        --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg

    - name: Add Kubernetes apt repository
      ansible.builtin.shell: |
        echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.32/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list


    - name: Install kubelet, kubeadm, kubectl
      apt:
        name:
          - kubelet
          - kubeadm
          - kubectl
        state: present
        update_cache: yes

    - name: Hold kubelet, kubeadm, kubectl at current version
      ansible.builtin.shell: |
        apt-mark hold kubelet kubeadm kubectl
```

Figure 20. k8s prerequisites setup with ansible, k8s-cluster.yaml

The next phase (see Figure 21) automates the Kubernetes control plane initialization using kubeadm, applies Flannel CNI for networking, and

securely extracts the kubeadm join command with token and CA hash. Worker node then automatically joins the cluster by executing the generated join script, enabling a fully functional multi-node Kubernetes environment.

```yaml

- name: Initialize control plane
  hosts: master
  become: true
  vars:
    join_command: ""
  tasks:
    - name: Run kubeadm init (if not already done)
      shell: kubeadm init --pod-network-cidr=10.244.0.0/16
      register: kubeadm_output
      args:
        creates: /etc/kubernetes/admin.conf

    - name: Ensure .kube directory exists
      file:
        path: /home/ubuntu/.kube
        state: directory
        owner: ubuntu
        group: ubuntu
        mode: '0700'

    - name: Copy admin.conf to user's kube config
      copy:
        remote_src: true
        src: /etc/kubernetes/admin.conf
        dest: /home/ubuntu/.kube/config
        owner: ubuntu
        group: ubuntu
        mode: '0600'

    - name: Apply Flannel CNI from user home
      become_user: ubuntu
      shell: kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
      environment:
        KUBECONFIG: "/home/ubuntu/.kube/config"

    - name: Extract join command from kubeadm init
      set_fact:
        join_command: "{{ kubeadm_output.stdout_lines | select('search', 'kubeadm join') | list | join(' ') }}"
      when: kubeadm_output is changed

    - name: Extract token from join command
      set_fact:
        kube_token: "{{ join_command | regex_search('--token\\s+(\\S+)', '\\1') }}"
      when: kubeadm_output is changed

    - name: Show join command (only if defined)
      debug:
        msg: "{{ kube_token }}"

    - name: Get CA cert hash
      shell: |
        openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | \
        openssl rsa -pubin -outform DER 2>/dev/null | \
        sha256sum | awk '{print $1}'
      register: ca_cert_hash

    - name: Show join command (only if defined)
      debug:
        msg: "{{ kube_token }}, CA cert hash: {{ ca_cert_hash }}"

    - name: Set join command fact
      set_fact:
        join_command: "kubeadm join {{ ansible_default_ipv4.address }}:6443 --token {{ kube_token[0] }} --discovery-token-ca-cert-hash sha256:{{ ca_cert_hash.stdout }}"

    - name: Show join command (only if defined)
      debug:
        msg: "{{ join_command }}"
      when: join_command is defined and join_command | length > 0

- name: Join worker nodes to cluster
  hosts: workers
  become: true
  tasks:
    - name: Create join script
      copy:
        content: "{{ hostvars.master1.join_command }}"
        dest: /tmp/join.sh
        mode: '0755'
      when: hostvars.master1.join_command is defined

    - debug:
        msg: "{{ hostvars.master1.join_command }}"
    - name: Execute join script
      shell: bash /tmp/join.sh
```

Figure 21. Cluster initialization and network setup, kubeadm-create.yaml

Commands used:

```
ansible-playboook k8s-cluster.yaml

ansible-playbook kubeadm-create.yaml
```

11.2 Longhorn and MinIO Installation

Before Longhorn deployment the prerequisites are installed on cluster nodes using ansible playbook (see Figure 22). Longhorn is installed in the cluster with helm using values file (see Figure 23). There we declare tolerations for longhorn components to run on the control plane node. We also add a selector, so the components can be scheduled only on nodes labelled with longhorn=true.

The commands used are:

```
ansible-playbook longhorn-prereqs.yaml

helm repo add longhorn https://charts.longhorn.io

helm repo update

kubectl label nodes spspamain01 longhorn=true

kubectl label nodes spspadev01 longhorn=true
helm install longhorn longhorn/longhorn --namespace long-
horn-system --create-namespace --version 1.8.1 -f val-
ues.yaml

docker-compose up -d
```

```yaml
---
- name: Prepare Kubernetes nodes for Longhorn
  hosts: all
  become: yes

  tasks:
    - name: Install open-iscsi and NFS client
      apt:
        name:
          - open-iscsi
          - nfs-common
        state: present
        update_cache: true

    - name: Enable and start iscsid service
      service:
        name: iscsid
        enabled: true
        state: started

    - name: Install system utilities
      apt:
        name:
          - bash
          - curl
          - util-linux
          - grep
          - gawk
          - lsof
        state: present

    - name: Ensure findmnt and lsblk are available
      shell: |
        which findmnt >/dev/null && which lsblk >/dev/null
      changed_when: false

    - name: Check filesystem type of root partition
      command: findmnt -n -o FSTYPE /
      register: fs_check
      changed_when: false

    - name: Fail if filesystem is not ext4 or xfs
      fail:
        msg: "Unsupported filesystem type for Longhorn: {{ fs_check.stdout }}"
      when: fs_check.stdout not in ['ext4', 'xfs']

    - name: Print success message
      debug:
        msg: "Node is ready for Longhorn (Filesystem: {{ fs_check.stdout }})"
```

Figure 22. Longhorn prerequisites installation with Ansible, Longhorn-prereqs.yaml

```yaml

global:

  nodeSelector:

    longhorn: "true"

        # -- Toleration for nodes allowed to run user-deployed components such as Longhorn Manager, Longhorn UI, and Longhorn Driver Deployer.
  tolerations:
  - key: "node-role.kubernetes.io/control-plane"

    operator: "Exists"

    effect: "NoSchedule"


csi:

  attacherReplicaCount: 2

  provisionerReplicaCount: 2

  resizerReplicaCount: 2

  snapshotterReplicaCount: 2

defaultSettings:

  replicaSoftAntiAffinity: false
  replicaAutoBalancw: ~
  taintToleration: node-role.kubernetes.io/control-plane:NoSchedule


longhornCsi:

  nodeSelector:
    longhorn: "true"

  tolerations:

    - key: "node-role.kubernetes.io/control-plane"

      operator: "Exists"

      effect: "NoSchedule"



longhornManager:

  nodeSelector:
    longhorn: "true"

  tolerations:

    - key: "node-role.kubernetes.io/control-plane"

      operator: "Exists"

      effect: "NoSchedule"
```

Figure 23. Longhorn values file

MinIO is deployed using Docker Compose (see Figures 24, 25) with four instances and an Nginx load balancer (see Figures 26, 27, 28). This setup ensures high availability by distributing traffic across all instances. If one container fails, the remaining instances continue to serve requests, maintaining cluster accessibility. In our deployment, we utilize the official MinIO Docker Compose configuration, which provisions two

volumes per instance. This setup simulates a distributed, multi-drive architecture typically found in production-grade environments. By assigning two volumes per container, MinIO is able to implement erasure coding across drives, thereby ensuring data redundancy and resilience to individual disk failures. This configuration also enhances performance through parallel I/O operations, as data reads and writes are distributed over multiple storage paths. Although our deployment may run on limited physical infrastructure, this volume layout effectively emulates a fault-tolerant storage system commonly used in bare-metal or cloud-based object storage solutions.

```yaml
# Settings and configurations that are common for all containers

x-minio-common: &minio-common

  image: quay.io/minio/minio:RELEASE.2025-04-03T14-56-28Z

  command: server --console-address ":9001" http://minio{1...4}/data{1...2}

  expose:

    - "9000"

    - "9001"

  # environment:

    # MINIO_ROOT_USER: minioadmin

    # MINIO_ROOT_PASSWORD: minioadmin

  healthcheck:

    test: ["CMD", "mc", "ready", "local"]

    interval: 5s

    timeout: 5s

    retries: 5



# starts 4 docker containers running minio server instances.

# using nginx reverse proxy, load balancing, you can access

# it through port 9000.

services:

  minio1:

    <<: *minio-common

    hostname: minio1

    volumes:

      - data1-1:/data1

      - data1-2:/data2



  minio2:

    <<: *minio-common

    hostname: minio2

    volumes:

      - data2-1:/data1

      - data2-2:/data2

```

Figure 24. docker-compose.yml, part 1

```
73    minio3:
74
75      <<: *minio-common
76
77      hostname: minio3
78
79      volumes:
80
81        - data3-1:/data1
82
83        - data3-2:/data2
84
85
86
87    minio4:
88
89      <<: *minio-common
90
91      hostname: minio4
92
93      volumes:
94
95        - data4-1:/data1
96
97        - data4-2:/data2
98
99
100
101   nginx:
102
103     image: nginx:1.19.2-alpine
104
105     hostname: nginx
106
107     volumes:
108
109       - ./nginx.conf:/etc/nginx/nginx.conf:ro
110
111     ports:
112
113       - "9000:9000"
114
115       - "9001:9001"
116
117     depends_on:
118
119       - minio1
120
121       - minio2
122
123       - minio3
124
125       - minio4
126
127
128
129 ## By default this config uses default local driver,
130
131 ## For custom volumes replace with volume driver configuration.
132
133 volumes:
134
135   data1-1:
136
137   data1-2:
138
139   data2-1:
140
141   data2-2:
142
143   data3-1:
```

Figure 25. docker-compose.yml, part 2

```
user  nginx;

worker_processes  auto;



error_log  /var/log/nginx/error.log warn;

pid        /var/run/nginx.pid;



events {

    worker_connections  4096;

}



http {

    include       /etc/nginx/mime.types;

    default_type  application/octet-stream;



    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '

                      '$status $body_bytes_sent "$http_referer" '

                      '"$http_user_agent" "$http_x_forwarded_for"';



    access_log  /var/log/nginx/access.log  main;

    sendfile        on;

    keepalive_timeout  65;



    # include /etc/nginx/conf.d/*.conf;



    upstream minio {

        server minio1:9000;

        server minio2:9000;

        server minio3:9000;

        server minio4:9000;

    }



    upstream console {

        ip_hash;

        server minio1:9001;

        server minio2:9001;

        server minio3:9001;

        server minio4:9001;
```

Figure 26. nginx.conf, part 1

```
67        server minio1:9001;
68
69        server minio2:9001;
70
71        server minio3:9001;
72
73        server minio4:9001;
74
75    }
76
77
78
79    server {
80
81        listen       9000;
82
83        listen  [::]:9000;
84
85        server_name  localhost;
86
87
88
89        # To allow special characters in headers
90
91        ignore_invalid_headers off;
92
93        # Allow any size file to be uploaded.
94
95        # Set to a value such as 1000m; to restrict file size to a specific value
96
97        client_max_body_size 0;
98
99        # To disable buffering
100
101       proxy_buffering off;
102
103       proxy_request_buffering off;
104
105
106
107       location / {
108
109           proxy_set_header Host $http_host;
110
111           proxy_set_header X-Real-IP $remote_addr;
112
113           proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
114
115           proxy_set_header X-Forwarded-Proto $scheme;
116
117
118
119           proxy_connect_timeout 300;
120
121           # Default is HTTP/1, keepalive is only enabled in HTTP/1.1
122
123           proxy_http_version 1.1;
124
125           proxy_set_header Connection "";
126
127           chunked_transfer_encoding off;
128
129
130
131           proxy_pass http://minio;
132
133       }
134
135   }
136
137
138
139   server {
140
141       listen       9001;
142
143       listen  [::]:9001;
144
145       server_name  localhost;
146
147
148
149       # To allow special characters in headers
150
151       ignore_invalid_headers off;
152
153       # Allow any size file to be uploaded.
```

Figure 27. nginx.conf, part 2

```
            proxy_pass http://minio;

        }

    }


    server {

        listen       9001;

        listen  [::]:9001;

        server_name  localhost;


        # To allow special characters in headers

        ignore_invalid_headers off;

        # Allow any size file to be uploaded.

        # Set to a value such as 1000m; to restrict file size to a specific value

        client_max_body_size 0;

        # To disable buffering

        proxy_buffering off;

        proxy_request_buffering off;


        location / {

            proxy_set_header Host $http_host;

            proxy_set_header X-Real-IP $remote_addr;

            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

            proxy_set_header X-Forwarded-Proto $scheme;

            proxy_set_header X-NginX-Proxy true;


            # This is necessary to pass the correct IP to be hashed

            real_ip_header X-Real-IP;


            proxy_connect_timeout 300;


            # To support websocket

            proxy_http_version 1.1;

            proxy_set_header Upgrade $http_upgrade;

            proxy_set_header Connection "upgrade";


            chunked_transfer_encoding off;


            proxy_pass http://console;

        }

    }

}
```

Figure 28. nginx.conf, part 3

After deploying MinIO, we generate an access token (see Figure 29) specifically for Longhorn to enable secure interaction with the designated backup bucket. This involves creating a Kubernetes secret that stores the base64-encoded access key, secret key, and the MinIO endpoint address (see Figure 30). In the Longhorn UI, a new backup target is configured using the MinIO URL and the name of the previously created secret containing the necessary credentials (see Figure 31).

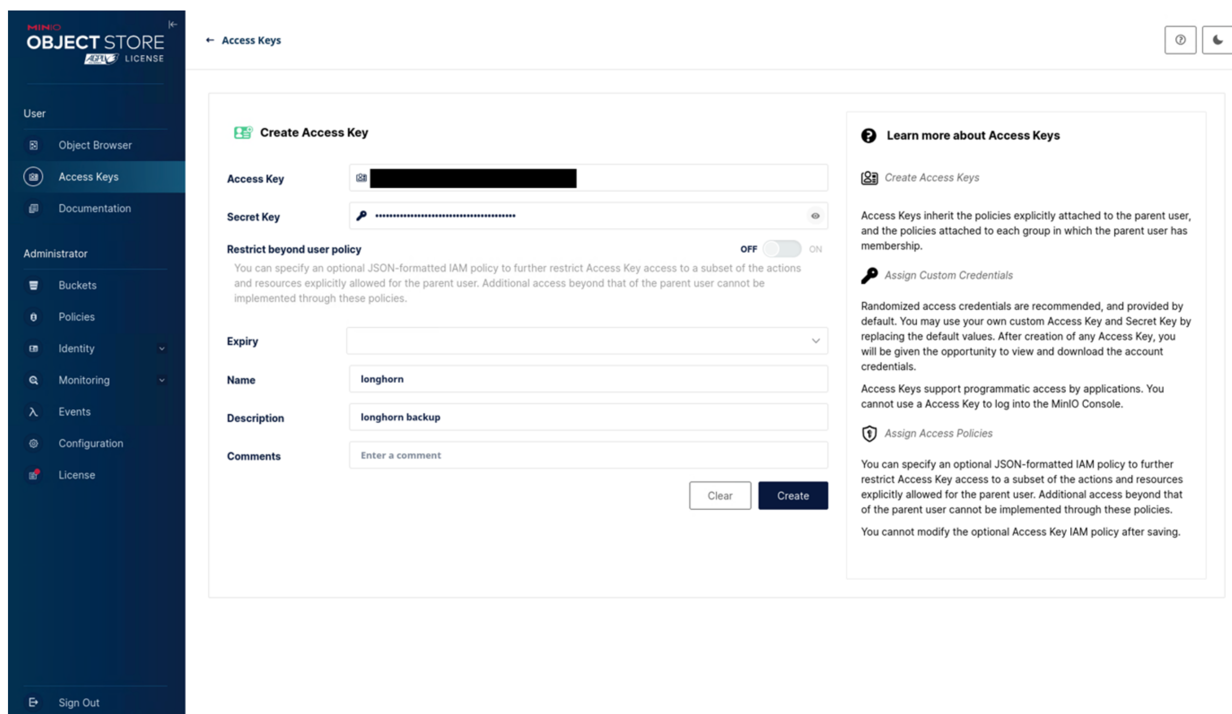

Figure 29. MinIO access token management page

```yaml
1   apiVersion: v1
2   kind: Secret
3   metadata:
4     name: minio-secret
5     namespace: longhorn-system
6   type: Opaque
7   data:
8     AWS_ACCESS_KEY_ID: [illegible]12cUV2NUNWdmVBMko=
9     AWS_SECRET_ACCESS_KEY: [illegible]Q0kxuQ1J4RVRjR0x4ZGRybjJnWWx6SklWSA==
10    AWS_ENDPOINTS: aHR0cDovL3Nwc3BhbWFpbjAxLm9mZmljZS5pcGUuY29ycDo5MDAw
```

Figure 30. MinIO credentials secret

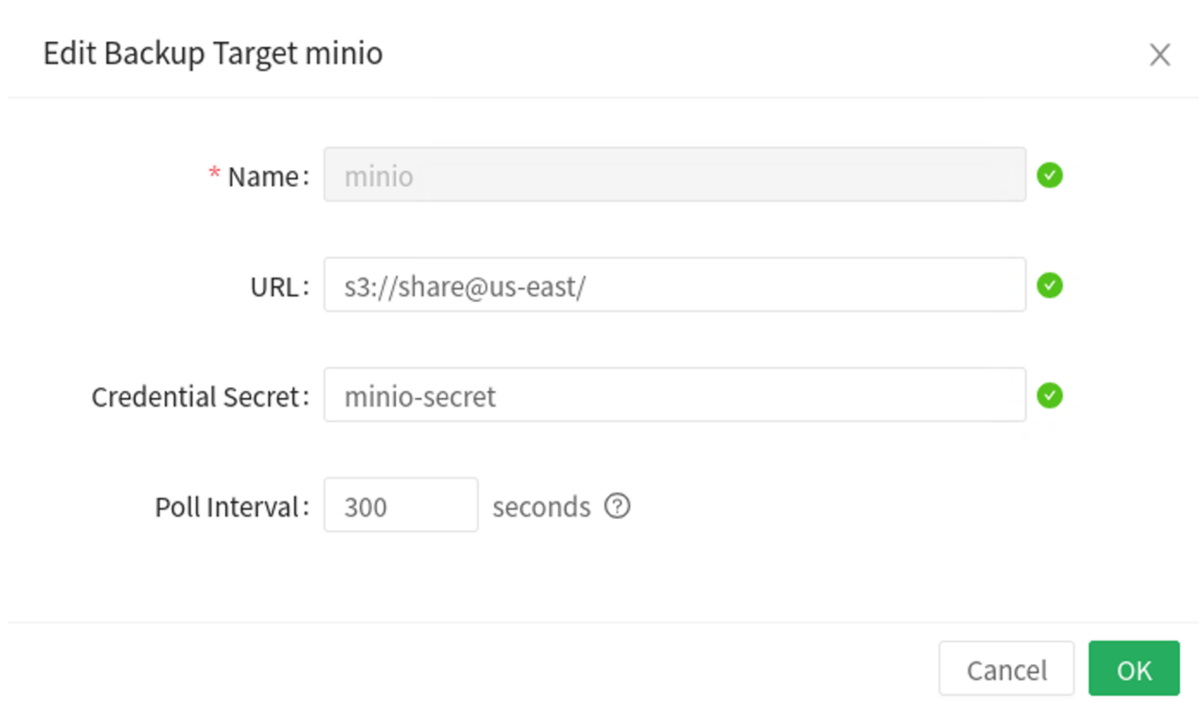

Figure 31. Backup target configuration

Once the backup target becomes available (see Figure 32), we select a Longhorn volume (see Figure 33) and initiate a backup using the configured MinIO destination (see Figure 34). The status of the backup process can be monitored through the Longhorn UI to ensure successful execution (see Figure 35). Finally, we verify the integrity and presence of the backup by inspecting the contents of the specified MinIO bucket (Figure 36).

Figure 32. Backup target status

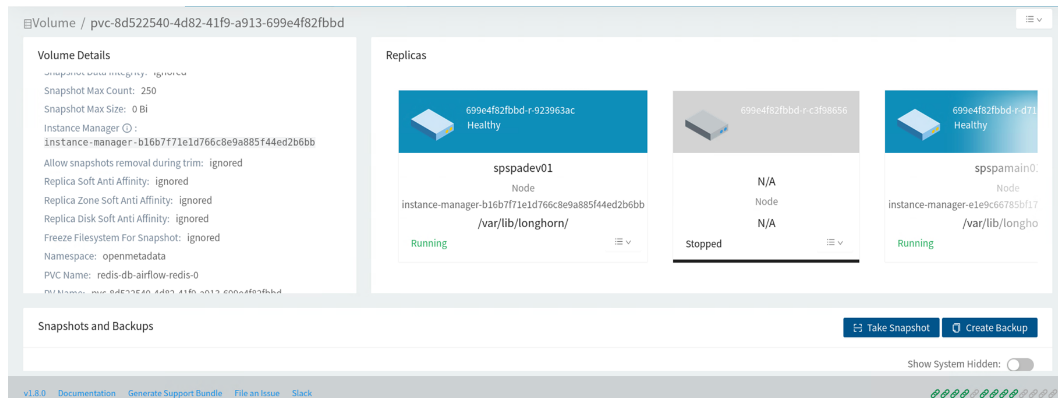

Figure 33. Longhorn volume management interface

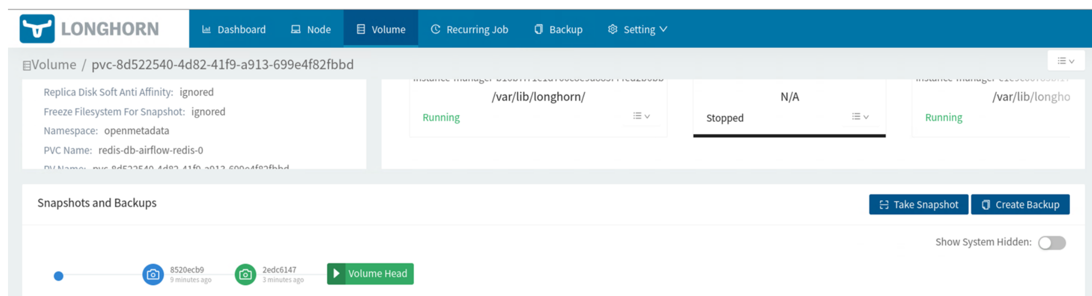

Figure 34. Longhorn volume backup history

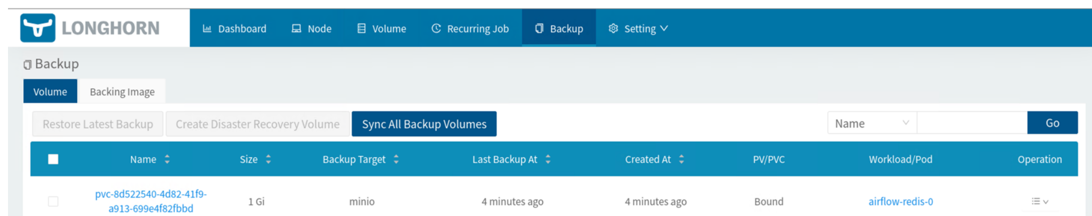

Figure 35. Longhorn backup management panel

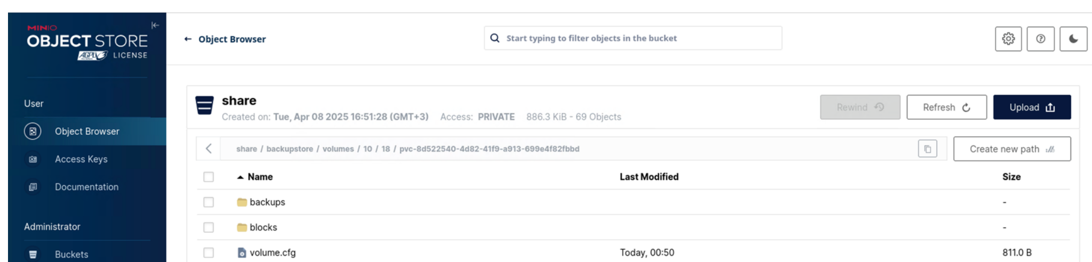

Figure 36. MinIO bucket contents

11.3 Nginx Ingress Controller Installation

The next step involves installing an Ingress controller to manage and define traffic routing rules within the Kubernetes cluster. Initially, we apply the necessary Custom Resource Definition (CRD) manifests required for the NGINX Ingress Controller to function properly. Following this, we deploy the NGINX Ingress Controller using Helm, referencing a custom values.yaml file (see Figure 37) to configure our environment. In this configuration, we specify tolerations to ensure that the Ingress Controller pods can be scheduled on any node regardless of taints.

Commands used:

```
kubectl apply -f https://raw.githubusercontent.com/nginx/ku-
bernetes-ingress/v5.0.0/deploy/crds.yaml

helm repo add ingress-nginx https://kubernetes.github.io/in-
gress-nginx

helm repo update

helm upgrade --install ingress-nginx ingress-nginx/ingress-
nginx --namespace ingress-nginx --create-namespace -f val-
ues.yaml
```

```yaml
1   controller:
2
3     progressDeadlineSeconds: 0
4
5     replicaCount: 2
6
7     tolerations:
8
9       - key: "node-role.kubernetes.io/control-plane"
10
11        operator: "Exists"
12
13        effect: "NoSchedule"
14
```

Figure 37. values.yaml for nginx ingress controller

11.4 Keycloak configuration

Before deploying DataHub, it is necessary to prepare the authentication realm in Keycloak, configure an authentication client, and import users from Active Directory (see Figure 38). In the client configuration section, a new client is created with a defined name and client ID, along with settings for the root URL, home URL, web origins, and redirect URI (see Figure 39). The Redirect URI in the OpenID Connect (OIDC) authentication flow determines the exact URL to which Keycloak is allowed to redirect users after successful login or logout. This mechanism ensures that authorization codes or tokens are only sent to trusted endpoints. After authentication, Keycloak returns the user to the application with the appropriate token or code, and after logout, it redirects the user to a predefined post-logout page.

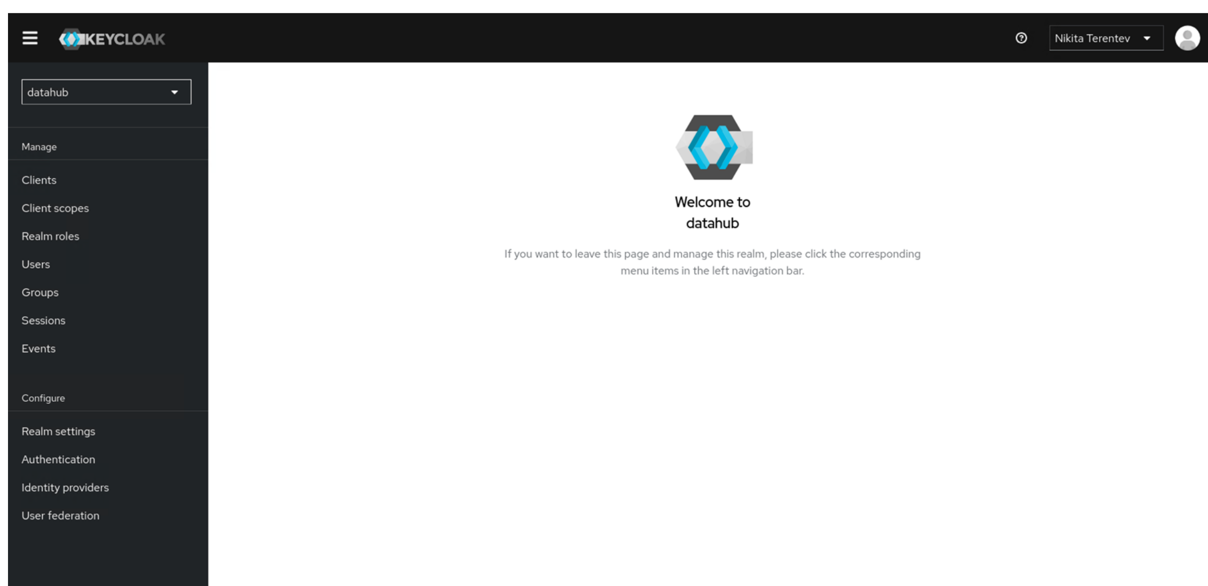

Figure 38. Datahub realm

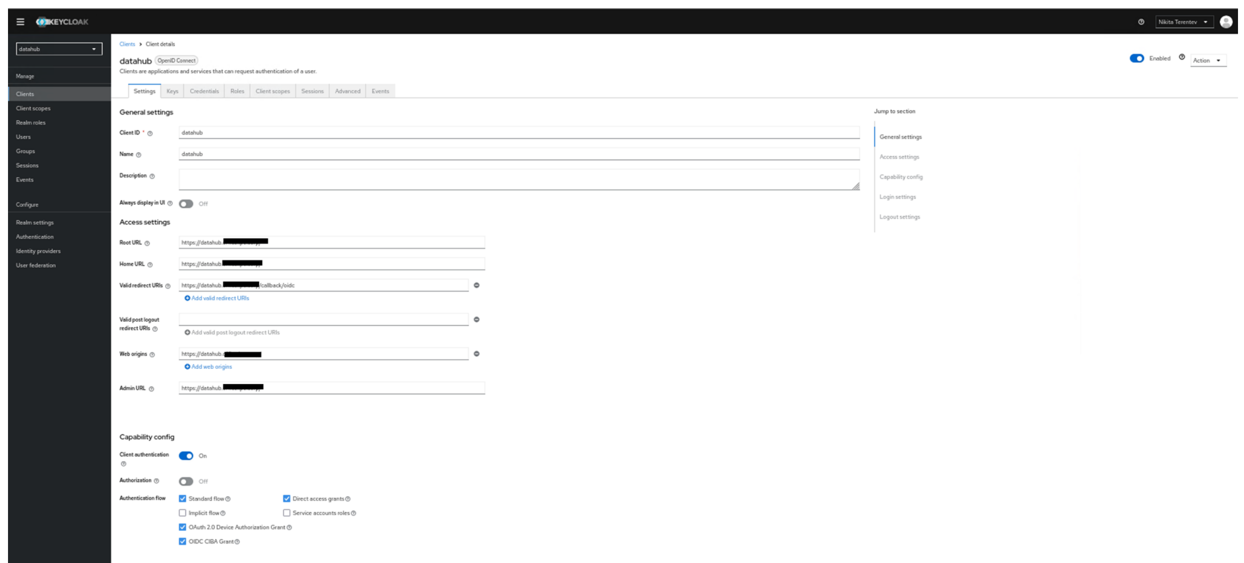

Figure 39. Keycloak client configuration

After completing the client configuration, the next step is to set up user federation in Keycloak. This allows the integration of external identity providers such as Active Directory for centralized authentication and user management. Using the credentials provided by the Cyber Security team, a connection to Active Directory is established, and users are imported based on the specified query in the User LDAP Filter field (see Figures 40, 41).

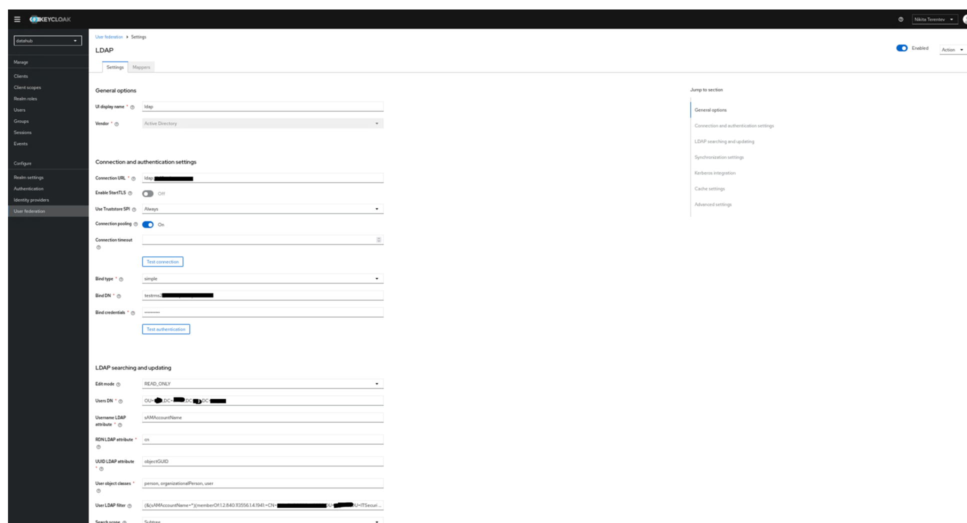

Figure 40. Keycloak federation configuration

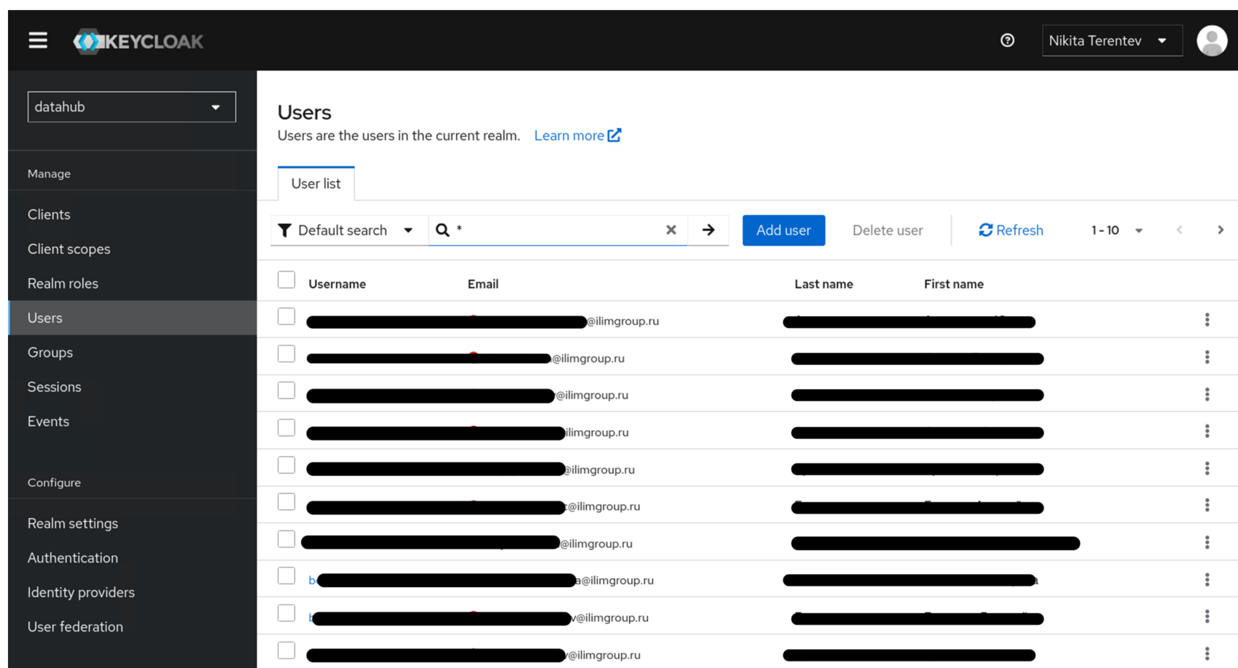

Figure 41. List of users imported from Active Directory

11.5 MetalLB deployment

We create an address pool that MetalLB can use to allocate IP addresses for services of type LoadBalancer. This pool defines the range of IPs that MetalLB is allowed to assign. In our deployment, only a single IP address from this pool is utilized (Figure 43), allowing external access to a specific internal service. To advertise this IP on the local network, MetalLB is configured in Layer 2 mode (see Figure 42). Additionally, MetalLB's speaker component must interact directly with the host network to send ARP responses. For this reason, the metallb-system namespace is created with an annotation that grants the required elevated privileges (see Figure 44).

Commands used:

```
helm repo add metallb https://metallb.github.io/metallb

helm upgrade --install metallb metallb/metallb -n metallb-system -create-namespace -f values.yaml
```

```
kubectl apply -f ns.yaml l2-adv.yaml address-pool.yaml
```

```yaml
apiVersion: metallb.io/v1beta1

kind: L2Advertisement

metadata:

  name: default

  namespace: metallb-system

```

Figure 42. l2-adv.yaml

```yaml
apiVersion: metallb.io/v1beta1

kind: IPAddressPool

metadata:

  name: main-pool

  namespace: metallb-system

spec:
  addresses:

  - 10.138.10.20-10.138.10.20

```

Figure 43. address-pool.yaml

```yaml
apiVersion: v1

kind: Namespace

metadata:

  name: metallb-system

  labels:

    pod-security.kubernetes.io/enforce: privileged

    pod-security.kubernetes.io/audit: privileged

    pod-security.kubernetes.io/warn: privileged
```

Figure 44. ns.yaml

11.6 DataHub deployment

With Longhorn, the Ingress-NGINX controller, and Keycloak properly configured, the necessary infrastructure components are in place for deploying DataHub. The deployment is carried out using Helm, with customizations specified in the values file. Authentication is set up using the OpenID Connect (OIDC) protocol (Figures 45 and 46), and ingress access is secured using a TLS secret (Figures 47 and 48). As a result of this configuration, accessing DataHub triggers a redirect to the Keycloak login page (Figure 49), where users can authenticate using their Keycloak credentials (Figure 50).

Commands used:

```
helm repo add datahub https://helm.datahubproject.io

kubectl apply -f datahub-tls.yaml datahub-oidc-secret.yaml
```

```
helm upgrade --install datahub datahub/datahub -n datahub –
create-namespace -f values.yaml
```

```
167  auth:
168    sessionTTLHours: "24"
169  # OIDC auth based on https://datahubproject.io/docs/authentication/guides/sso/configure-oidc-react
170  oidcAuthentication:
171    enabled: true
172    provider: other # <- choose only one
173
174    clientId: "datahub"
175  #   # clientSecret: your-client-secret
176  #   # only needed if you would like to store the client secret in secret
177    clientSecretRef:
178      secretRef: datahub-oidc-secret
179      secretKey: secret
180  #   # only needed if provider is `other`
181    discoveryUri: "https://sp/realms/datahub/.well-known/openid-configuration"
182    #discoveryUri: "
183
184    # only needed if provider is `okta`
185    # oktaDomain: your-okta-domain.com
186
187
188    # only needed if provider is `azure`
189    # azureTenantId: your-azure-tenant-id
190    # if needed, it should set meaningful defaults from provider
191    # scope: "openid profile email"
192
193    # The attribute that will contain the username used on the DataHub platform.
194    user_name_claim: "email" #was email
```

Figure 45. values.yaml authentication section

```
1   apiVersion: v1
2   kind: Secret
3   metadata:
4     name: datahub-oidc-secret
5     namespace: datahub
6     uid: 4eacf419-c564-46d9-84b6-b4032a0104ae
7     resourceVersion: '20323935'
8     creationTimestamp: '2025-04-10T06:28:04Z'
9     managedFields:
10      - manager: kubectl-create
11        operation: Update
12        apiVersion: v1
13        time: '2025-04-10T06:28:04Z'
14        fieldsType: FieldsV1
15        fieldsV1:
16          f:data:
17            .: {}
18            f:secret: {}
19          f:type: {}
20    selfLink: /api/v1/namespaces/datahub/secrets/datahub-oidc-secret
21  type: Opaque
22  data:
23    secret:
```

Figure 46. OIDC k8s secret

```
149  ingress:
150    className: "nginx"
151    enabled: true
152    extraLabels: {}
153    annotations: {}
154      # kubernetes.io/ingress.class: nginx
155      # kubernetes.io/tls-acme: "true"
156    hosts:
157      - host: datahub.
158        paths:
159          - /
160        # redirectPaths: []
161    tls:
162    - secretName: datahub-tls
163      hosts:
164        - datahub.
165    pathType: ImplementationSpecific
```

Figure 47. values.yaml Ingress configuration section

```
1   apiVersion: v1
2   kind: Secret
3   metadata:
4     name: datahub-tls
5     namespace: datahub
6     uid: 36fb2632-b853-4957-babb-31f632abd67d
7     resourceVersion: '20092033'
8     creationTimestamp: '2025-04-09T12:41:59Z'
9     managedFields:
10      - manager: kubectl-create
11        operation: Update
12        apiVersion: v1
13        time: '2025-04-09T12:41:59Z'
14        fieldsType: FieldsV1
15        fieldsV1:
16          f:data:
17            .: {}
18            f:tls.crt: {}
19            f:tls.key: {}
20          f:type: {}
21    selfLink: /api/v1/namespaces/datahub/secrets/datahub-tls
22  type: kubernetes.io/tls
23  data:
24    tls.crt: >-
25
26    tls.key: >-
27
```

igure 48. TLS key and certificate k8s secret

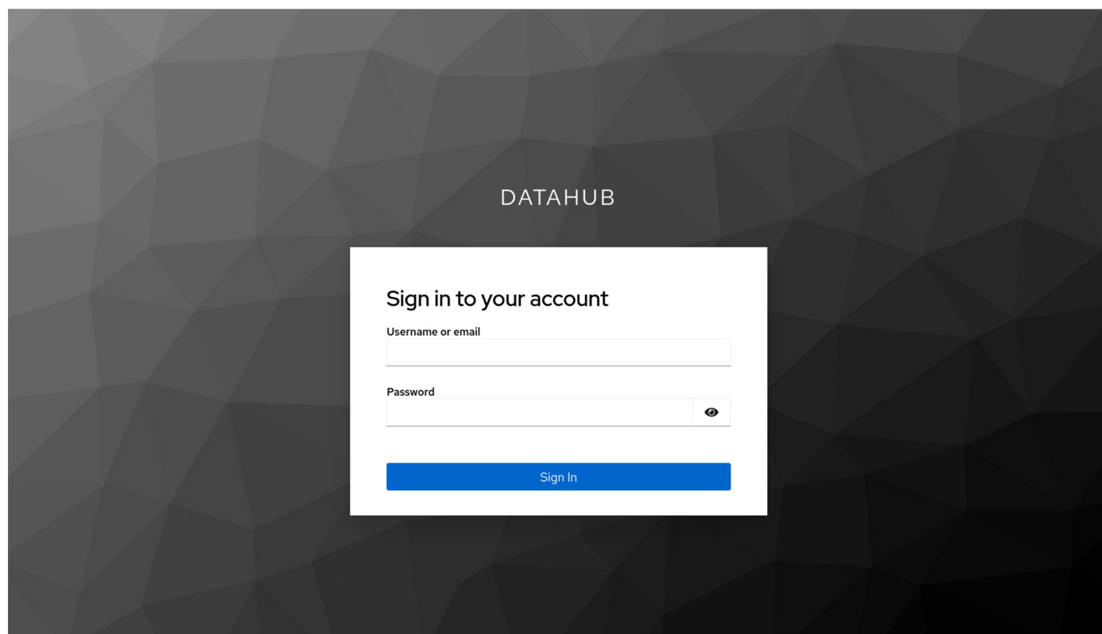

Figure 49. Keycloak authentication page

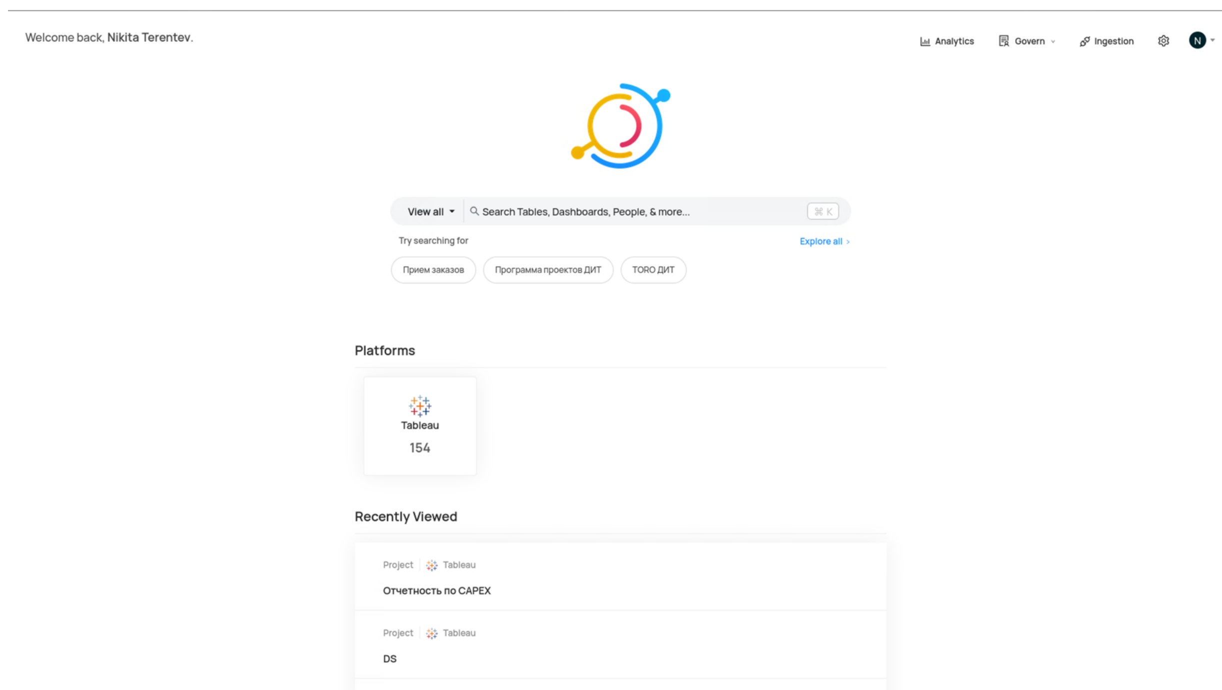

Figure 50. DataHub main page after Keycloak authentication

11.7 Prometheus operator deployment

After successfully deploying DataHub, the next step is to deploy the Prometheus Operator to enable monitoring of the Kubernetes cluster. The Prometheus Operator simplifies the deployment and management of Prometheus instances and facilitates the collection of metrics from nodes, pods, and services. These metrics are visualized in Grafana, providing real-time insights into the system's performance and overall health. Following the operator installation via Helm, additional Kubernetes manifests were applied to create a Service (Figure 51), Ingress (Figure 52), and TLS secret (Figure 53) for secure access to the Grafana interface. Once Prometheus was configured as a data source, Grafana dashboards became available to monitor cluster compute resources (Figure 54).

Commands used:

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm repo update

helm upgrade --install kube-prometheus-stack prometheus-community/kube-prometheus-stack -n monitoring --create-namespace

kubectl apply -f grafana-service.yaml dashboard-tls.yaml grafana-ingress.yaml
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: grafana
  namespace: monitoring
spec:
  type: ClusterIP
  ports:
  - name: web
    port: 3000
    protocol: TCP
    targetPort: 3000
  selector:
    app.kubernetes.io/name: grafana
```

Figure 51. grafana-service.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: dashboard-tls
  namespace: monitoring
type: Opaque
data:
  tls.crt: >-

  tls.key: >-

```

Figure 52. dashboard-tls.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: grafana-ingress
  namespace: monitoring
  annotations:
    # type of authentication
    #    nginx.ingress.kubernetes.io/auth-type: basic
    # prevent the controller from redirecting (308) to HTTPS
    nginx.ingress.kubernetes.io/ssl-redirect: 'false'
    # name of the secret that contains the user/password definitions
    #nginx.ingress.kubernetes.io/auth-secret: basic-auth
    # message to display with an appropriate context why the authentication is required
    #nginx.ingress.kubernetes.io/auth-realm: 'Authentication Required '
    # custom max body size for file uploading like backing image uploading
    nginx.ingress.kubernetes.io/proxy-body-size: 10000m
spec:

  ingressClassName: nginx
  tls:
    - hosts:
      - dashboard.
      secretName: dashboard-tls
  rules:
  - host: dashboard.
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: grafana
            port:
              number: 3000
```

Figure 53. grafana-ingress.yaml

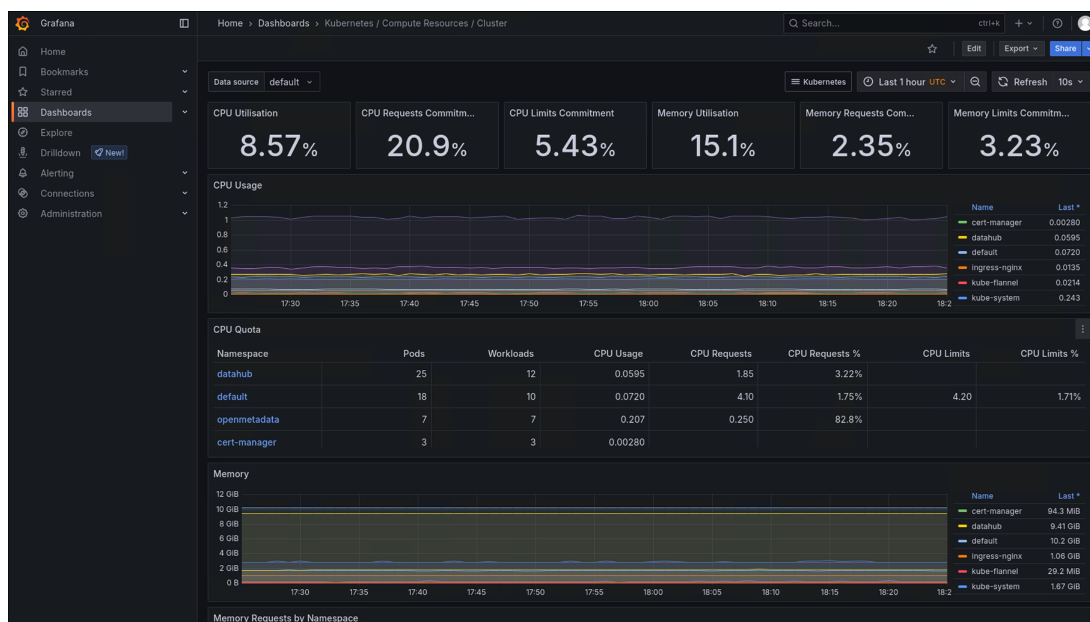

Figure 54. Grafana cluster compute resources dashboard using Prometheus source

11.8 Loki deployment

Loki is deployed using Helm in single-binary mode, meaning that all core components—such as the ingester, querier, and index gateway—run within a single unified process. This approach simplifies deployment and resource management. Once deployed, Loki serves as the backend for log aggregation. It receives log data from a log router, stores the logs, and makes them queryable via Grafana. Replication factor is configured to be 2 with 2 Loki replicas, it means that each log entry will be stored on both replicas. This ensures high availability and fault tolerance—if one replica fails, the other can continue serving logs without data loss. Additionally, Kubernetes tolerations are configured to allow each Node to run a Loki instance, ensuring proper pod scheduling across nodes. (see Figure 55)

```
kubectl create ns loki

helm repo add grafana https://grafana.github.io/helm-charts

helm repo update helm upgrade --install loki grafana/loki --
namespace loki -f values.yaml
```

```yaml
loki:
  commonConfig:
    replication_factor: 2
  schemaConfig:
    configs:
      - from: "2024-04-01"
        store: tsdb
        object_store: s3
        schema: v13
        index:
          prefix: loki_index_
          period: 24h
  pattern_ingester:
      enabled: true
  limits_config:
    allow_structured_metadata: true
    volume_enabled: true
  ruler:
    enable_api: true

minio:
  enabled: true

deploymentMode: SingleBinary

singleBinary:
  replicas: 2
  tolerations:
    - key: "node-role.kubernetes.io/control-plane"
      operator: "Exists"
      effect: "NoSchedule"

# Zero out replica counts of other deployment modes
backend:
  replicas: 0
read:
  replicas: 0
write:
  replicas: 0

ingester:
  replicas: 0
querier:
  replicas: 0
queryFrontend:
  replicas: 0
queryScheduler:
  replicas: 0
distributor:
  replicas: 0
compactor:
  replicas: 0
indexGateway:
  replicas: 0
bloomCompactor:
  replicas: 0
bloomGateway:
  replicas: 0
```

Figure 55. Loki values.yaml

11.9 Logging operator deployment

The final step in the observability stack is the deployment of the Logging Operator, which enables centralized log collection and processing. After deploying Loki with Helm to serve as the log storage backend, the Logging Operator is configured to launch Fluent Bit agents on each node. These lightweight agents collect logs from all running containers and forward them to Fluentd, which applies filters and transforms the data. The processed logs are then sent to Loki.

The Logging resource defines the overall configuration of the logging system (Figure 56). It specifies the logging architecture and the target namespaces where the logging components are deployed. The ClusterFlow resource defines the rules for log selection and processing (Figure 57), including filters and routing logic. The ClusterOutput resource specifies the endpoints to which the logs are sent (Figure 58). After successful configuration of these resources, logs can be queried and visualized in Grafana (Figure 59).

Commands used:

```
helm upgrade --install --wait --create-namespace --namespace logging logging-operator oci://ghcr.io/kube-logging/helm-charts/logging-operator

kubectl apply -f logging.yaml flow.yaml output.yaml
```

```yaml
apiVersion: logging.banzaicloud.io/v1beta1
kind: Logging
metadata:
  name: default-logging-namespaced
  namespace: logging
spec:
  fluentd: {}
  fluentbit: {}
  controlNamespace: logging
  watchNamespaces:
    - monitoring
    - kube-system
    - datahub
```

Figure 56. logging.yaml

```yaml
apiVersion: logging.banzaicloud.io/v1beta1
kind: ClusterFlow
metadata:
  name: test-log-flow
  namespace: logging
spec:
  filters:
    - tag_normaliser:
        format: ${namespace_name}.${pod_name}.${container_name}
    - parser:
        key_name: message
        parse:
          type: json
        remove_key_name_field: true
        reserve_data: true
        emit_invalid_record_to_error: false
  match:
  globalOutputRefs:
    - test-log
```

Figure 57. flow.yaml

```yaml
apiVersion: logging.banzaicloud.io/v1beta1
kind: ClusterOutput
metadata:
  name: test-log
  namespace: logging
spec:

  loki:
    url: http://loki-gateway.monitoring.svc.cluster.local:80
    buffer:
      timekey: 1m
      timekey_wait: 30s
      timekey_use_utc: true
    tenant: "1"
```

Figure 58. output.yaml

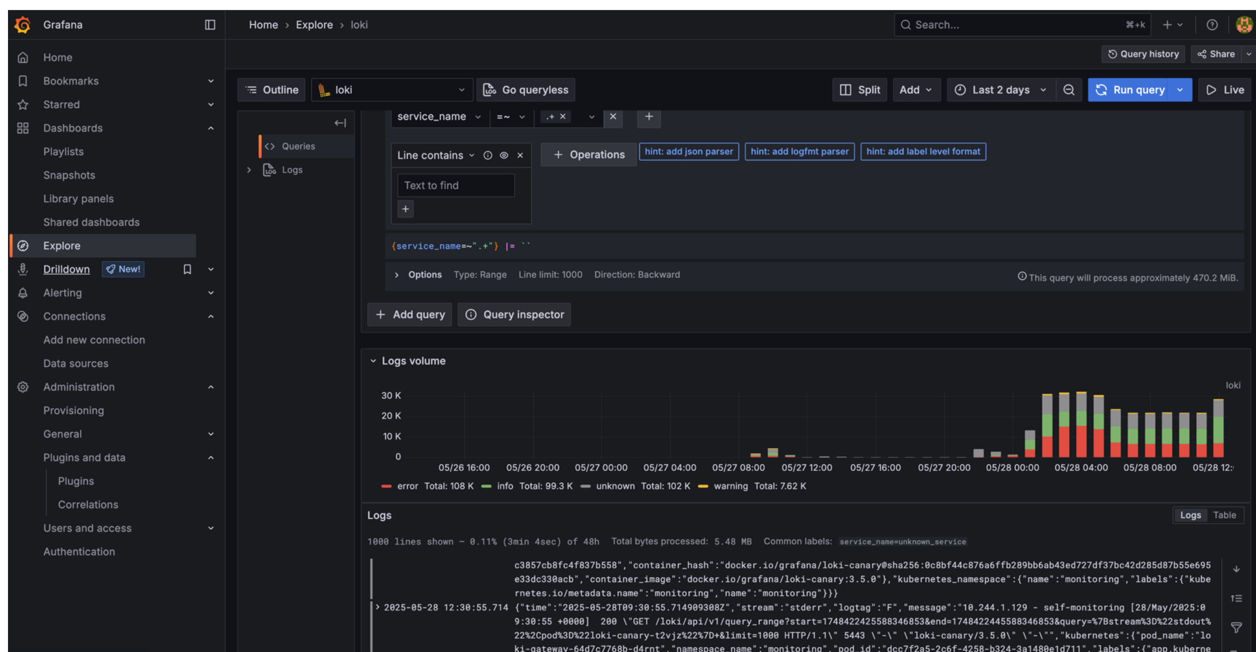

Figure 59. Grafana with Loki source

12 CONCLUSION AND FUTURE WORK

This thesis explored the deployment of a fully open-source infrastructure stack on a bare-metal Kubernetes cluster. The aim was to build a cloud-native environment capable of supporting various services such as authentication, storage, logging, monitoring, and metadata governance—without relying on managed or proprietary solutions. One of the main achievements of this thesis is the demonstration that open-source tooling—when thoughtfully combined—can offer a stable and functional platform suitable for hosting complex applications. The use of tools like Longhorn, Grafana, Prometheus, Loki, and DataHub created a complete ecosystem for observability and data management.

However, there are areas where the system could be further improved. One such area is performance evaluation—the current setup has not been tested under high load, and future work could focus on benchmarking resource usage, latency, and fault recovery. Another important aspect is policy enforcement. Tools like Gatekeeper, which implements Open Policy Agent (OPA) as a Kubernetes admission controller, could be added to define and enforce rules on resource usage, security labels, or container configurations. This would add an additional layer of governance and help prevent misconfigurations at deployment time. A third potential improvement is the adoption of centralized secret management. While Kubernetes supports secrets natively, integrating tools like HashiCorp Vault would allow for more secure handling of sensitive data, better access control, auditing capabilities, and secret rotation. This becomes especially important as the system grows and hosts multiple applications with varying levels of sensitivity.

In conclusion, this thesis has demonstrated that a fully open-source and self-managed infrastructure can effectively support cloud-native workloads on bare-metal environments. While certain aspects such as security, scalability, and performance require further refinement, the implemented system provides a solid foundation for future expansion. It

shows that, with careful design, open-source tools can enable the creation of reliable, flexible, and extensible platforms without relying on proprietary solutions.

REFERENCES

Ansible. (May 15, 2025a). *Ansible playbooks.* Retrieved May 27, 2025, from https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html

Ansible. (May 15, 2025b). *How to build your inventory.* Retrieved May 27, 2025, from https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html#inventory-basics-formats-hosts-and-groups

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (Eds.). (2016). *Site reliability engineering: How Google runs production systems* (Chapter 6). O'Reilly Media. https://sre.google/books/

Canonical Ltd. (January 18, 2023). *Containerization vs. Virtualization: understand the differences*. Retrieved May 13, 2025, from https://ubuntu.com/blog/containerization-vs-virtualization

Cisco. (n.d.). *What is Single Sign–On?* Retrieved May 21, 2025, from https://www.cisco.com/site/us/en/learn/topics/security/what-is-single-sign-on-sso.html

CNCF (October 20, 2024) *Cluster Architecture.* Retrieved May 13, 2025, from https://kubernetes.io/docs/concepts/architecture/

CNCF. (June 16, 2021a). *Storage*. Retrieved May 14, 2025, from https://kubernetes.io/docs/concepts/storage/

CNCF. (June 16, 2021b). *Configuration*. Retrieved May 22, 2025, from https://kubernetes.io/docs/concepts/configuration/

CNCF. (March 27, 2025). *Deployments*. Retrieved May 13, 2025, from https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

CNCF. (April 7, 2024). *Pods*. Retrieved May 13, 2025, from https://kubernetes.io/docs/concepts/workloads/pods/

CNCF. (July 30, 2024). *Labels*. Retrieved May 13, 2025, from https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

CNCF. (September 18, 2024) *Ingress.* Retrieved May 13, 2025, from https://kubernetes.io/docs/concepts/services-networking/in-gress/

CNCF. (September 3, 2024). *Namespaces*. Retrieved May 13, 2025, from https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

CNCF. (October 20, 2024). *Cluster Architecture*. Retrieved May 13, 2025, from https://kubernetes.io/docs/concepts/architecture/

CNCF. (October 31, 2024). *Custom Resources*. Retrieved May 22, 2025, from https://kubernetes.io/docs/concepts/extend-kuber-netes/api-extension/custom-resources/#adding-custom-re-sources

CNCF. (March 25, 2025). *Persistent Volumes*. Retrieved May 22, 2025, from https://kubernetes.io/docs/concepts/storage/persistent-volumes/

CNCF. (July 16, 2024). *Operator Pattern*. Retrieved May 22, 2025, from https://kubernetes.io/docs/concepts/extend-kuber-netes/operator/

CNCF. (April 15, 2025). *ReplicaSets*. Retrieved May 13, 2025, from https://kubernetes.io/docs/concepts/workloads/controllers/repli-caset/

CNCF. (September 18, 2024). *Services, Load Balancing and Network-ing*. Retrieved May 14, 2025, from https://kuber-netes.io/docs/concepts/services-networking/

CNCF. (February 18, 2025). *Taints and Tolerations.* Retrieved May 25, 2025, from https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/

CNCF. (n.d.). *Using Helm.* Retrieved May 29, 2025, from https://helm.sh/docs/intro/using_helm/

CNCF. (May 29, 2025). *Logging Operator.* Retrieved May 29,2025, from https://kube-logging.dev/docs/

Datadog. (May 23, 2023). *Container security fundamentals part 4: Cgroups*. Retrieved May 12, 2025, from https://securitylabs.datadoghq.com/articles/container-security-fundamentals-part-4/#cgroups-basics

Docker Inc. (n.d.). *What is Docker?* Retrieved May 12, 2025, from https://docs.docker.com/get-started/docker-overview/#the-underlying-technology

DataHub. (n.d.) *UI Based Ingestions.* Retrieved May 23, 2025, from https://docs.datahub.com/docs/ui-ingestion

DataHub (n.d.) *DataHub Serving Architecture.* Retrieved May 24, 2025, from https://docs.datahub.com/docs/architecture/metadata-serving

Grafana Labs. (n.d.) *Grafana documentation.* Retrieved May 29, 2025, from https://grafana.com/docs/grafana/latest/

Google. (n.d.). *What is Object storage?* Retrieved May 23, 2025, from https://cloud.google.com/learn/what-is-object-storage

Keycloak. (n.d.). *Keycloak docs*. Retrieved May 21, 2025, from https://www.keycloak.org/

MetalLB. (n.d.). *MetalLB docs*. Retrieved May 14, 2025, from https://metallb.io/

Microsoft. (n.d.-a). *What is OIDC?* Retrieved May 21, 2025, from https://www.microsoft.com/en-us/security/business/security-101/what-is-openid-connect-oidc

Microsoft. (n.d.-b). *Identity and Access documentation*. Retrieved May 24, 2025, from https://learn.microsoft.com/en-us/windows-server/identity/identity-and-access

MinIO. (n.d.). *MinIO object storage for Linux*. Retrieved May 24, 2025, from https://min.io/docs/minio/linux/index.html

OpenID. (n.d.). *How OpenID Connect Works?* Retrieved May 21, 2025, from https://openid.net/developers/how-connect-works/

Prometheus. (n.d.). *What is Prometheus?* Retrieved May 24, 2025, from https://prometheus.io/docs/introduction/overview/

RedHat. (March 21, 2025). *What is container orchestration?* Retrieved May 12, 2025, from https://www.redhat.com/en/topics/containers/what-is-container-orchestration

RedHat. (May 11, 2022). *What is a Kubernetes operator?* Retrieved May 22, 2025, from https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-operator

Xu, A. (April 8, 2022) *Virtualization vs. Containerization.* Retrieved May 13, 2025, from https://blog.bytebytego.com/p/what-are-the-differences-between