MASARYK UNIVERSITY

FACULTY OF INFORMATICS

High-Performance Object Storage in Kubernetes Cluster

Master's Thesis

BC. ONDŘEJ MOLÍK

Brno, Fall 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

High-Performance Object Storage in Kubernetes Cluster

Master's Thesis

BC. ONDŘEJ MOLÍK

Advisor: RNDr. Daniel Tovarňák, Ph.D.

Department of Computer Systems and Communications

Brno, Fall 2023


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Ondřej Molík

Advisor: RNDr. Daniel Tovarňák, Ph.D.

Acknowledgements

I would like to thank RNDr. Daniel Tovarňák, Ph.D for his guidance and professional advice, which were crucial for the creation of this thesis. I am also really grateful to all my colleagues, who provided me with valuable feedback and their cooperation. Further, I want to thank my family and friends for their support throughout my entire studies.

Abstract

This thesis aims to design, deploy and evaluate a MinIO-based object storage in a Kubernetes cluster. Particular attention is paid to the stability, maintainability and overall production readiness of the result.

Firstly, the motivation and technologies used for the implementation are introduced. Secondly, the software and hardware assets allocated for the object storage and supporting infrastructure are presented along with the technical limits they present. Based on the selected technologies and the assets available, four architectural designs are created, each for a different infrastructure layer. The layers are the Kubernetes, object storage, monitoring and logging. Following the designs, the cluster provisioning process is implemented. The provisioned cluster is then evaluated, and the results are compared to reference designs and expectations. Secondary results of the testing include processes for the maintenance and troubleshooting of the object storage and supporting infrastructure. The practical result consists of the object storage cluster and the collection of artefacts for its deployment.

The work concludes with suggestions for future development that would increase the performance of the object storage and further improve the observability of the underlying infrastructure.

Keywords

CSIRT-MU, Data Lake, Data LakeHouse, data warehouse, Kubernetes, MinIO

Contents

List of Tables

List of Figures

List of Listings

Acronyms

ACME Automatic Certificate Management Environment. 17, 31

AGPL GNU Affero General Public License. 9

API Application Program Interface. 3–5, 8, 9, 11, 14, 15, 17, 20, 23, 24, 27, 29, 31–34, 42, 43, 45, 46, 48

ARP Address Resolution Protocol. 34

AWS Amazon Web Services. 8, 9, 16

BGP Border Gateway Protocol. 30, 34

BIOS Basic Input Output System. 26

BMC Baseboard Management Controller. 19, 26, 27

CA Certification Authority. 33

CIS Center for Internet Security. 6

CNI Container Network Interface. 5, 6, 31

CPU Central Processing Unit. 3, 9, 23, 31, 32, 37

CVE Common Vulnerabilities and Exposures. 6

DNS Domain Name Server. 16–18, 23, 27, 29, 31

ECC Error Correction Code. 19

ECMP Equal-cost multi-path. 34

FIPS Federal Information Processing Standard Publication. 6

HTTP Hypertext Transfer Protocol. 7, 10, 31, 38, 40, 48

ILM Index Life-cycle Management. 20

IP Internet Protocol. 4, 14, 16, 20–23, 26, 27

IPv6 Internet Protocol version 6. 28

JSON Javascript Object Notation. 7

LTS Long Term Support. 27, 46, 47

LVM Logical Volume Manager.. 23, 27

MTU Maximum Transmission Unit. 6, 18, 21, 22, 30

NTP Network Time Protocol. 18, 28

NVMe NVM Express. 19, 23, 24, 27

PBAC Policy Based Access Control. 8

QSFP28 Quad Small Form Pluggable. 18, 19, 21, 22

RAID Redundant Array of Inexpensive Disks. 10, 19, 23, 27

RBAC Role Based Access Control. 3, 4, 12, 31, 38, 39, 43

SAN Subject Alternative Name. 29

SATA Serial AT Attachment. 19, 23, 27

SSD Solid State Drive. 19, 27

SSH Secure Shell. 2, 7, 27, 28, 31

TCP Transmission Control Protocol. 14, 33, 40

TLS Transport Layer Security. ix, x, 3, 17, 32–34, 36, 38, 40

TSIG Transaction Signature. 16, 31

TTFB Time To First Bytes. viii, 46

TTL Time To Live. 23

UDP User Datagram Protocol. 14

URL Uniform Resource Locator. 29, 37, 39

VRL Vector Remap Language. 14, 15

WAL Write Ahead Log. 36, 38

YAML YAML Ain't Markup Language. 2, 8, 12, 14, 40, 43

Introduction

The task aimed to be solved in this thesis is the deployment, operation and maintenance of high-performance object storage on a bare-metal Kubernetes cluster. The main conceptual purpose of this object storage cluster is to be used by a data warehouse solution, which is being developed in the given environment. Currently, the environment does not offer an object storage service with sufficient performance and resiliency characteristics.

The other available options, such as OpenStack Ceph-based object storage or less sophisticated deployments of MiniIO clusters, proved to be unable to satisfy the performance requirements of the data warehouse, especially during read operations. Furthermore, the operational procedures, such as monitoring and audit logging, are not implemented to an adequate degree, making the availability and stability of these services subject to frequent unplanned changes. These findings, as well as the iterative improvements achieved during the testing of these services and the experimentation with the alternatives, were gathered over the last one and a half years and resulted in the formulation of the requirements, which should be fulfilled by the end of this thesis.

The text of the thesis is structured in the following manner. After this introduction, the thesis consists of five other chapters. The first is chapter 1 and introduces the Kubernetes environment, technologies and available options. The chapter 2 describes the available assets, restrictions and architectural limitations that must be considered. It finishes with four abstract architecture proposals for the cluster itself, the object storage with centralized monitoring and logging. The chapter 3, details the deployment process, obstacles encountered during implementation and the usage of the artefacts from the attachment. The second to last chapter, chapter 4, evaluates the performance of the object storage and describes the necessary maintenance and troubleshooting procedures. The thesis ends with chapter 5, which shortly evaluates the fulfilment of the original task and possible future areas of development.

The practical result consists of a bare-metal Kubernetes cluster optimised for MinIO object storage and a suit of benchmark results and operations guidelines for the maintenance of cluster and replication of the results in other environments, existing in parallel with the reference implementation cluster. The description of these procedures is a part of the text of this thesis. The artefacts used by the practical result are stored in the attachment. They consist of the necessary definitions and automations needed for the deployment of the object storage cluster and can serve as a boilerplate for the creation of another similar cluster.

1 Technologies

This chapter introduces the software, technologies and circumstances relevant to the task which is being solved in this thesis. It is divided into seven sections, four of which describe the technologies relevant to the given layer of the overall architecture. The first is Kubernetes, the second is object storage, the third is monitoring, and the fourth is logging. This system of layers is followed in the following chapters as well. The other three sections introduce the containerization principles and other tooling, which is not a clear part of any of the aforementioned layers.

1.1 Ansible

Ansible[1] is an open-source automation tool used for configuration management and infrastructure provisioning. It is being developed by Redhat, Inc. and was initially released in 2012. The automations are defined in YAML files, and the values in the files can be templated using Jinja2[2].

The basic unit in Ansible is called a `task`; multiple tasks, along with related files, templates and variables, form an Ansible role. Roles usually describe an independent unit which can be used by itself, e.g. installation and configuration of a web server. Playbooks are used to combine host-specific variables and configurations with roles and apply them to a particular group of servers. The target hosts and their groups are specified in so-called inventories, which in the case of this thesis is just a single YAML file with specific contents [1]. Usually, Ansible roles are stored each in a separate Git repository. This way, they can be referenced in the `requirements.yaml` file in the root of multiple playbook Git repositories. While roles are typically fully parameterized and do not contain secrets or other system-specific information, the playbooks can contain secrets encrypted by Ansible Vault. This allows the administrator just to clone the playbook repository, use `ansible-galaxy` command to fetch necessary dependencies, add the file with the password to the vault and use the playbook. Ansible mainly uses SSH to connect to the hosts, which are being provisioned, and copy files or edit configuration. All production-grade Ansible roles and playbooks should be idempotent so that they can be executed repeatedly with configuration adjustments and updates.

Ansible was chosen over other infrastructure provisioning options because it is widely used in the environment in which the object storage cluster is deployed, and I am proficient in it [2].

1.2 Containers

Since the Kubernetes is a platform for containerized applications, an introduction to containerization is in order. The Linux containers are an alternative to the concept of virtual machines, the main difference being that the containers share the same kernel with each other and the host operating system. Each container is a running instance of a container image, which is essentially an archive of a root file system for the given

1. `https://www.ansible.com`
2. `https://jinja.palletsprojects.com`

service. The container images are built using a layered file system, allowing the reuse of the layers between images, leading to lower storage requirements. Ideally, each container should contain a single running process, but typically this is not upheld. The process typically has a separate network and user namespace, along with restrictions on resources it can use, e.g. memory, CPU or network throughput. The containers can also mount files, directories, and block devices from their host; these mounted structures are called volumes [3].

Out of many container options in the Linux ecosystem, Docker[3] became the most popular one [4]. Kubernetes originally used Docker as its container run-time, but in recent versions, it was replaced by more purpose-built CRI-O[4].

The massive deployment of containers facilitated new solutions to the already known system administration issues. These issues include but are not limited to the management of TLS certificates, service discovery, resource allocation and scheduling, distributed persistent storage, etc... This led to more complex environments, which in turn necessitated the existence of management tools and platforms such as Kubernetes and new concepts of storage like Object Storage.

1.3 Kubernetes

Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized applications. It was originally developed by Google Inc. for the purposes of their Google Container Engine platform in 2014[5].

The ecosystem of Kubernetes is in many ways similar to the ecosystem of Linux-based operating systems. There are many different distributions specializing in different use cases and many inter-changeable components which implement the same API with some additional features. The differences in the additional features can be quite large, but the core API and resources remain always the same. For the purposes of this thesis, only the distributions developed by Rancher Labs were considered.

Kubernetes provides an API, which controls everything inside the clusters. The entities managed by the API are called resources. The resources or their combination capture the entire state of the cluster. Not all resources represent a running process; some just store information, e.g. configuration. Resources with an underlying process are called workloads. A workload resource in the cluster can connect to this API and instrument other resources. The access is controlled via RBAC[6]. An introduction of the most common resources relevant to the rest of the thesis follows.

- **Namespace** – *Namespace* is not a workload resource, an empty *Namespace* does not consume any resources. Other resources must be placed in some *Namespace*, unless they are cluster-scope, in which case their names are usually prefixed with `Cluster` keyword.
- **Pod** – It is the most basic workload resource in Kubernetes. Everything that is running inside a cluster is running in some *Pod*. It consists of one or more containers, which share the network namespace and the access to Kubernetes API.

3. `https://www.docker.com`
4. `https://cri-o.io`

- **Deployment** – This resource is an abstraction built on top of *Pods*. It is usually used to manage replicas of stateless applications, such as micro-services. The volumes of a *Deployment* are shared by all its replicas.
- **StatefulSet** – This resource is similar to *Deployment*, but is intended for stateful applications. There are two significant differences: the first is that it is possible to reference a single particular *Pod*, and the second is that it is possible to define so-called *volumeClaimTemplates*, which provision a separate volume for each replica of the *StatefulSet*. The shared volumes are still available, as on *Deployments*.
- **DaemonSet** – Used to managed *Pods* of applications, which require exactly one *Pod* on each node in the cluster. It is usually reserved for core cluster components such as log-collecting agents, load balancers, etc...
- **Service** – *Service* is not a workload resource; it is essentially a floating IP address from the cluster-service IP range (e.g. `10.43.0.0/16`) with predictable domain name (e.g. nameofservice.nameofnamespace.svc). The target *Pod* is chosen by a label selector.
- **ConfigMap** – Contains one or more key-value items which can be mounted to *Pods* as files.
- **Secret** – Similar to *ConfigMap*, but the contents are base64 encoded and can be encrypted on API of the cluster, so that it is only stored in the encrypted form not as a plain-text.
- **Ingress** – *Ingress* is essentially reverse proxy configuration, which references a *Service* port by the name of the service and name or number of the port. The reverse proxy implementation varies between different Kubernetes distributions, but the most common, which is used by RKE2 is Nginx Ingress Controller[5].
- **PersistentVolume** – A *PV* resource represents physical storage devices with clearly defined capacity, such as mounted drives or network shares.
- **PersistentVolumeClaim** – This resource purpose is to claim or allocate storage in *PersistentVolumes*. These claims are, in turn, mounted to *Pods*. A *PVC* can be either created dynamically along with the underlying *PV* by a *StorageClass* or statically by the administrator, who needs to define both *PVC* and the corresponding *PV*.
- **StorageClass** – As stated above, this resource represents essentially a group of volumes with a common provisioning method and underlying storage.
- **CRD** – Custom resource definition allows the administrator of the cluster to extend the core API with additional resources. These resources usually require a so-called operator, which creates, modifies and deletes *Pods* and other resources based on the *CRDs* state. Essentially creating another abstraction layer on top of the core API. The operator uses the Kubernetes API and RBAC roles given to it to carry out these changes.

5. `https://github.com/kubernetes/ingress-nginx`

There are two other important concepts in the ecosystem: labels and annotations. Labels are usually for the selection of resources by other resources, e.g. *Service* uses a label selector to specify which *Pods* are part of said *Service*. All resources can have labels, and some of the labels are added by default to all resources of a given type. Documentation also provides a recommended label set to use for identification of components of one application [7]. Annotations can also be present on all resources, but instead of selection, they are usually used for side effects [5]. For example, the Nginx Ingress Controller implements the *Ingress* API but also provides further functionality via annotations, e.g. `nginx.ingress.kubernetes.io/whitelist-source-range`.

The Kubernetes API is also used by the administrators to create, modify and remove resources from the cluster. The most popular tool used for this purpose is common for all Kubernetes distributions and is called `kubectl`. It is configured using the `KUBECONFIG` environment variable, which defines where the active configuration file is located. The aforementioned configuration file is typically either downloaded from the Rancher web interface or from one of the master nodes. An example of `kubectl` configuration and usage is shown in Listing 1

```
export KUBECONFIG=~/.kube/config

# list the nodes of the cluster
kubectl get nodes

# apply a Kustomization
kubectl apply -k example/
```

Listing 1: Example of configuration and usage of `kubectl`

The networking between *Pods* on different nodes is implemented by so-called CNI plugins, which typically use packet encapsulation (e.g. VXLAN) to tunnel the traffic through the underlying network infrastructure.

1.3.1 RKE2

Out of the available Kubernetes distributions, the RKE2[6] was selected. Previously, other Kubernetes distributions were tested, including k3s[7] and RKE1[8]. K3s proved to be a hindrance in use cases requiring the exchange of many included components such as CNIs and *Ingress* controllers. The biggest obstacle to using RKE1 was that it still relies on Docker[9] as the container runtime. The final deciding factor of the selection process was that the Managed Kubernetes is also based on RKE2.

It is being developed by Rancher Labs, a company closely associated with SUSE. Among the distinguishing features is the easiness of deployment and removal, which lowers the entry barrier of migrating to unmanaged Kubernetes. Each RKE2 release

6. `https://docs.rke2.io`
7. `https://github.com/k3s-io/k3s`
8. `https://github.com/rancher/rke`
9. `https://docker.io`

is provided with two shell scripts, which can install or uninstall RKE2, do not have any side effects and do not leave any orphaned files in the host operating system after removal.

Unlike k3s distribution from Rancher Labs, which focuses on environments with restricted resources, RKE2 keeps the features of k3s and adds hardening, along with built-in support for more sophisticated CNI plugins [8]. Out of these plugins, we chose Calico; the reasons for this choice are clarified in later chapters.

The above-mentioned hardening is materialized by the following measures. All RKE2 components are subject to CVE scans before the release is published. The documentation provides a CIS v1.23 hardening profile, which is optionally enforced by the RKE2 server. Furthermore, the entire distribution is FIPS 140-2 compliant, allowing for its usage in security-sensitive projects. An improvement over RKE1 is that Docker is no longer used as the default container run-time because it was replaced with a more modern CRI-O. The last important feature is that RKE2 is closely integrated with the Rancher web interface, which can be used to manage access to the Kubernetes cluster.

1.3.2 Calico

Calico is a CNI plugin developed by Tigera, Inc. and offers several advanced features such as multi-cluster networking, FIPS compliant mode and extended implementation of *NetworkPolicies*. Furthermore, it seems to perform well in network throughput testing with larger MTU "Overview of kubernetes cni plugins performance"[9], which is something that proved to be a significant issue during the previous iterations of object storage clusters.

For three major reasons, it was chosen over the other available CNI plugins. Firstly, it is one of the most popular plugins, offering large community support. Secondly, it can be configured to run without packet encapsulation, which can often lead to network performance degradation. Lastly, Calico implements additional *CRDs*, which can be used to configure the host firewall from Kubernetes.

- **HostEndpoint** – This resource represents a single network interface of one node in the cluster. It should always have a label applied to itself so that it can be selected by *GlobalNetworkPolicies*.

- **GlobalNetworkPolicy** – It provides a way to configure a Calico-based Kubernetes-compatible firewall on the cluster nodes. Each resource can contain multiple rules; the rules affect only the interfaces that match the given label selector. Multiple resources are merged in an additive manner, allowing everything that is allowed in at least one of the rules.

1.3.3 Kustomization

Kustomization is the more native and simpler method of organizing multiple Kubernetes resources into one coherent service stack. It does not require any additional tooling because Kubernetes command-line utility contains the `kubectl kustomization` command. The main benefits of this method are the simplicity and the fact that all changes and configurations are typed and adhere to the known JSON schema of the resources.

A Git repository with a Kubernetes service deployed by Kustomization must contain `kustomization.yaml` file; any other structure or files in the repository are optional and depend on the developer's choice. Kustomization also allows referencing of remote manifest files served over HTTP(s) or other Kustomizations in Git repositories served over SSH.

```yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

namespace: minio

resources:
  - https://congnito.inc/smth.yaml
  - resources/*.yaml

secretGenerator:
  - name: minio-external-certificate
    type: kubernetes.io/tls
    files:
      - tls.crt=files/tls.crt
      - tls.key=files/tls.key
  - name: minio-root
    literals:
      - MINIO_ROOT_USER=root
      - MINIO_ROOT_PASSWORD=changeme

generatorOptions:
  disableNameSuffixHash: true
```

Listing 2: Example of Kustomization file for MinIO

1.3.4 Helm

Another way to manage services deployed in Kubernetes is Helm. Compared to the aforementioned Kustomization, it is more flexible and suitable for the portable definition of an application stack, e.g. some micro-service with database, caches and authentication server. The basic unit in Helm is called a chart. Each chart consists of the following files, with additional options for more complex cases irrelevant to this thesis.

- `Chart.yaml` – Contains metadata of the chart such as name, version and the maintainers.

- `values.yaml` – Stores variables with completely optional structure and types. These are combined with templates to render the resource manifests, which are then applied to the cluster.

- `templates/*.yaml` – The templates are written in Golang templating language, even though the suffix would suggest that the files contain YAML. The templating language is not required. It is possible to place the resource manifests here without any parametrization. Each file in this directory usually contains the definition for one resource.

However, Helm has three major disadvantages, which result in the general preference of Kustomization over Helm in situations where parametrization and portability are not required.

The first is the lack of an inner structure of the values file; it has no standard or agreed-upon common structure. This leads to an anti-pattern where the authors of more complex charts parametrize almost everything and essentially re-implement the API of the templated resources with some minor but frequent deviations. The result is that the chart values look similar across various charts, but the values are slightly different and incompatible, making any issue extremely difficult to find and resolve.

The second was already stated: the templates are not structured or typed, and everything inside the files is a plain-text string, resulting in an increased amount of hard-to-diagnose bugs in more complex templates.

The third is that many charts are either written by a third party, which needs to be trusted as well as the developer of the software which is being deployed, or by the developer, who does not have sufficient Kubernetes background to keep up with the best practices and hardening measures.

The practical results of this thesis contain no Helm charts because the specificity of the deployment environment did not require parametrization or portability. Only used charts are the components of RKE2, which required configuration modification from the default values.

1.4 Object Storage

Objects storage is a relatively new type of storage, which differs from the more traditional block storage in the internal organization. It does not use files and directories which are located on a particular storage device. Rather, it uses objects organized in buckets with prefixed paths. These objects can be versioned and are subject to centrally managed PBAC, which can, for example, give the user access only to objects with particular prefixes or suffixes even when the bucket contains a mix of various suffixes.

The original and most popular object storage standard is called S3 and was developed by AWS. Most object store implementations can also emit notifications about the object creation, modification or deletion. The object's life-cycle management can remove objects based on their age or move them to a slower tier of storage. Since it abstracts away any

direct relation with physical drives, it allows for much better scalability than distributed block storage.

It is most commonly used in cloud-native environments for the following purposes. Among the simpler use cases are backups and archival of raw data such as logs or network flow records, which usually utilize automatic expiration and write-only users but typically do not require high performance. Object storage can also be used to serve static assets for websites or even complete websites. However, this thesis focuses on the third and most complex use case, data lakes and data warehouses.

These are usually read-heavy because the data is continually written as it is being generated and collected but can be queried simultaneously by several parallel users or processes. The majority of data of these platforms consists of objects with similar size, e.g. 100 MiB, which contain compressed Parquet or ORC files and much smaller storage for metadata, which is usually represented by a standard relational database such as PostgreSQL. The data chunk size is homogeneous because it allows the platform to optimize the access to it better. The architecture of one such platform is described in "Cloud Native Data Platform for Network Telemetry and Analytics"[10].

The choice of the particular implementation of hosted S3 compatible storage was pre-determined by the circumstances of the environment in which I worked on the thesis.

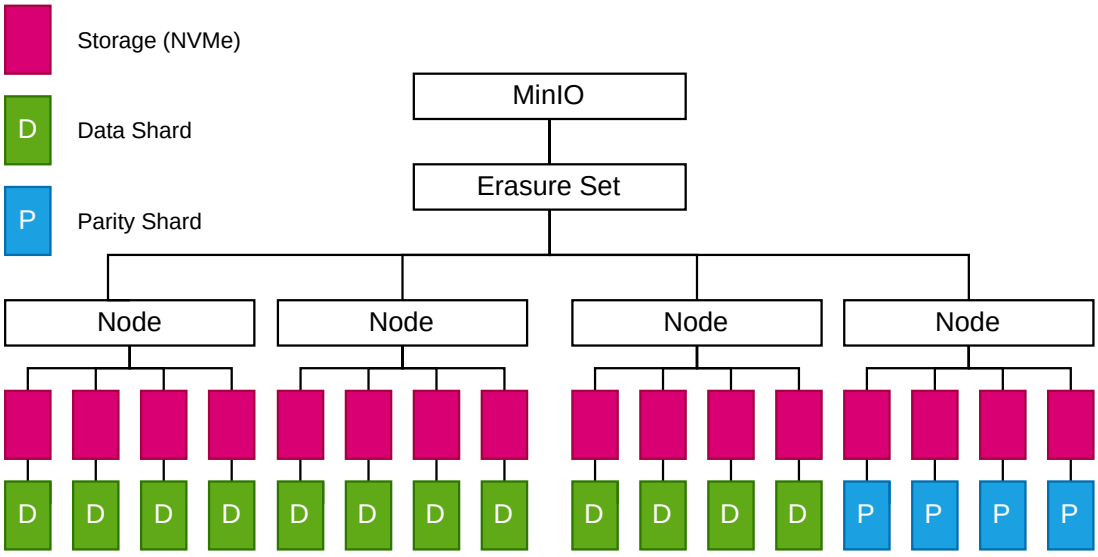

Figure 1.1: MinIO erasure coding EC:8 diagram [11]

1.4.1 MinIO

MinIO is an open-source implementation of the S3 object storage API, licensed under a AGPL. It provides a feature set similar to the original S3 API implementation by AWS. However, some very particular requirements need to be fulfilled during deployment. Otherwise, the speed, features and stability of storage might be severely reduced.

All nodes inside a MinIO cluster should be homogeneous, e.g. have the same network connectivity, the same CPU and memory configuration and most importantly, have the

same even number of identical drives. The drive restriction is due to the erasure coding, which defines how many drives can be lost without losing the ability to read and write data [12].

Erasure coding used on objects in MinIO is similar to the parity used on block device RAID, e.g. RAID6 is equivalent to parity 2. MinIO uses Solomon-Reed error correction coding with parity ranging from `EC:2` to `EC:8` or the limit of the particular disk pool [13]. The coding provides protection from data loss in case a drive or node fails or from bit rot caused by drive errors and potential bugs in their firmware [11]. The parity on the object level also significantly reduces healing times as the contents of the entire disk do not have to be rewritten. One server can run multiple disk pools, each with different parity settings and disk characteristics. The only restriction is that each pool must be definable in an expression with both nodes and disks having consecutive indexes, e.g.: `https://minio-0...1.minio-hl.minio.svc/disk0...1`.

MinIO[10] also provides a Kubernetes operator for managing the object store cluster. This operator was not deemed suitable for our production usage; concrete reasons are explained later in this thesis.

1.4.2 Warp

Warp is an open-source S3 benchmarking tool developed by the same team as MinIO. The main benefit offered by this tool is the possibility of distributed benchmarks, where one Warp server instruments several Warp clients in order to saturate the total bandwidth of the S3 cluster.

The benchmark is divided into sections according to the HTTP method used, e.g. GET, PUT or STAT. There is one additional category called `mixed`, which basically is a sum of all other categories. The results are expressed in two numbers: the throughput of objects per second and the throughput of bytes per second. The data from the printed results are computed and automatically stored in a file in the client's current working directory, allowing for additional interpretation later.

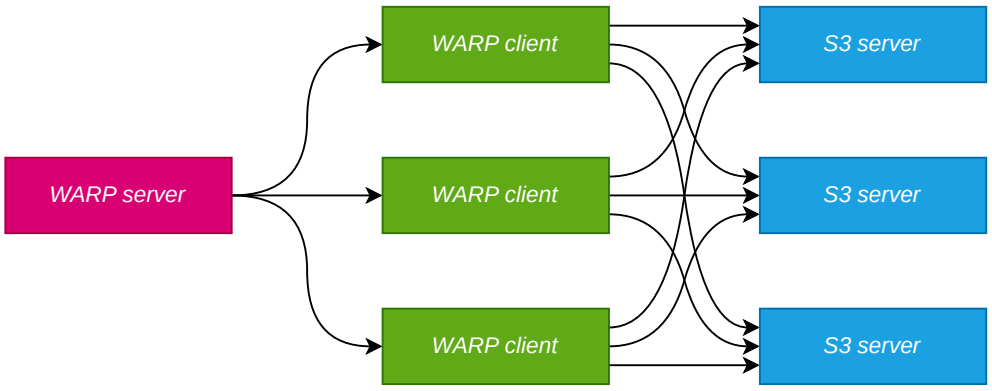

Figure 1.2: Distributed Warp diagram [14]

10. `https://min.io`

1.5 Monitoring

The purpose of monitoring is to be able to continuously evaluate the performance, availability and other characteristics of deployed services and proactively notify the administrators of potential problems and anomalies. The data inputted into a monitoring stack can include the following types[11].

- **Logs** – Logs are an infinite append-only sequence of individual log records, which consist of the serialized message and its context [15].
- **Metrics** – A periodically collected numerical measure value in the system, usually identified by a set of string labels. Metrics are to be viewed as continuous.
- **Traces** – Traces consist of many so-called spans and capture the journey of a request or user through the system. A different component usually generates each of the spans.

The umbrella term for the three categories of data listed above is observability. While it is common to combine all observability data into one stack in cloud-native environments, this thesis uses a more traditional approach, where the monitoring stack deals only with metrics and the log collection is implemented separately. The trace collection options is not elaborated further, as the components used to implement the object storage cluster do not have trace instrumentation built in. The stack for the collection of metrics and alerting uses the three components described in the following sections.

1.5.1 Prometheus

Prometheus[12] is an open-source monitoring and alerting toolkit. Since its inception in 2012, it has become a de-facto standard of monitoring for Kubernetes and other cloud-native environments.

It is designed primarily to scrape metrics from metric endpoints exposed by the monitored application. However, this model requires the applications to have built-in instrumentation for metrics. Applications without built-in instrumentation for metrics in Prometheus format can have an adjacent exporter process. The exporter uses either another API provided by the application or the log generated by the application to generate metrics and publish them to the metrics endpoint.

The resolution of metrics is determined by the scrape interval of the Prometheus instances, which can differ for each metric endpoint. The endpoints are plaintext web pages, optionally with Snappy[13] compression enabled. The metric series are distinguishable via the usage of labels. Labels can be either present directly on the metric endpoint or added later in the pipeline, e.g. after scraping. Metrics are of different types, the most common of them being `counter` and `gauge`. The `counter` incrementally increases with

11. `https://microsoft.github.io/code-with-engineering-playbook/observability/log-vs-metric-vs-trace`
12. `https://prometheus.io`
13. `https://github.com/google/snappy`

each event it measures, resetting when the application or exporter restarts. The `gauge` represents the value of the given metric at one point in time.

The query language used by Prometheus is called PromQL. The queries can either just select metrics using labels or apply more advanced functions and transformations such as relabeling. A non-trivial example of PromQL is shown in Listing 3.

```
sum(
  rate(
    vector_component_received_events_total{component_kind="source"}[1m]
  )
) by (component_id)
```

Listing 3: PromQL query, with the ingest event rate for Vector.dev sources

The metrics being ingested into Prometheus are subject to recording and alerting rules. Recording rules just execute a given PromQL expression and store the result as a new metric. Alerting rules can contain the same syntax but should end in a comparison or another expression producing a boolean value, determining whether the alert is triggered. There are many pre-made alerting rules for the different Kubernetes components; the most prominent is the repository called kubernetes-mixin[14] which is used for the monitoring of the object storage cluster.

There are several deployment modes of Prometheus on Kubernetes. The first one would be to deploy Prometheus as any other application with *Statefulset*, service and configure it via *ConfigMap*. However, the Prometheus operator provides a much more streamlined service discovery experience, which, instead of a single YAML file, uses the following *CRDs* to describe endpoint, paths and additional labels of metrics.

- **Prometheus** – The resource configures an instance of Prometheus, this instance can either be limited to the *Namepace* where it is deployed or it can use cluster-wide RBAC to discover the configuration in the form of *ServiceMonitors*, *PodMonitoring*, *Probes* and *PrometheusRules* in the other *Namespaces*.
- **Alertmanager** – Defines a cluster of Alertmanager instances; the most important attributes specify the *ServiceAccount* the instances use to discover their configuration stored in *ConfigMaps*, the number of instances in the cluster and the usual things, such as resource requests and limits, security contexts and persistent storage.
- **ServiceMonitor** – This resource uses Kubernetes labels to select a *Service* from which the metrics should be scraped.
- **PodMonitor** – Similar to *ServiceMonitor*, but rather than selecting *Services* it points to ports on *Pods*. It is used in cases when each *Pod* of the application presents different metrics, e.g., an application with multiple replicas which are not aware of each other.

14. `https://github.com/kubernetes-monitoring/kubernetes-mixin`

- **Probe** – Principally similar to the two resources above, but it is used for remote targets, which are not represented by any Kubernetes resources. It typically reaches the service via the public endpoint, not the in-cluster endpoints.

- **PrometheusRule** – It provides a Kubernetes-native way of defining Prometheus recording and alerting rules. The resource schema is similar to plain Prometheus rules used without the operator. It does not reference or select any other resource in the cluster. Rather, the Prometheus instance uses its *ServiceAccount* and selectors defined on the *Prometheus* resource to discover configuration and reload itself.

1.5.2 Grafana

Grafana is an open-source observability platform which can visualize various time-series data including metrics, logs and traces. In the recent versions, the support alert configuration was added as well. It is a de-facto standard for monitoring visualisations in Kubernetes environments; because of this, there is a wide variety of prepared dashboards. Large technology companies such as Siemens and Paypal are among its most prominent users.

The service itself does not store any data such as metrics or logs, and it only stores the user information and configurations of visualizations, dashboards and alerts. These are stored in an adjacent database, which is either a fully featured database such as PostgreSQL or a lightweight one like SQLite. For purposes of this work, the SQLite variant proved to be simpler to deploy and maintain and more than sufficient in terms of performance.

The data being visualized is queried from so-called data sources, which Grafana supports a wide range of. The data sources relevant to this thesis are Prometheus, Alertmanager and Elasticsearch.

The aforementioned dashboards can be created manually in the web interface or loaded on startup as JSON files. This is especially useful in the case of pre-made dashboards for common services such as Kubernetes components or MinIO.

1.5.3 Alertmanager

Alertmanager handles alerts sent by other software, including Prometheus. It is being developed by the same team as Prometheus, resulting in a close integration of both services. The main purpose of Alertmanager is to deduplicate, route and group the received notifications to the configured integration, e.g. Slack or Mattermost [16]. Each notification can have a set of labels, which usually includes the severity of the notification, a link to the Grafana dashboard or a run-book, which contains the actions the responsible personnel must take. Based on the labels, the alerts can be silenced, grouped or inhibited, i.e. only the alert that the entire cluster is unavailable will be sent instead of alerts for all services inside said cluster.

Alertmanager instances deployed on Kubernetes cluster can be defined as a *CRD* and managed by the Prometheus Operator, which was introduced in the previous chapter. The instances of Alertmanager should consist of more than two replicas to prevent downtime. The sources of the alerts need to be configured to send the alerts separately

to each instance, and the Alertmanager cluster deduplicates the messages automatically in case multiple cluster members receive the same alert.

1.6 Logging

A typical cloud-native logging pipeline consists of three main components plus the applications which generate the logs. The first component are the agents deployed next to the services generating the logs; the only purpose of these agents is to forward the logs to the second component, the aggregator. The aggregator is only deployed once in a central location, preferably outside the cluster. It transforms, filters and normalizes the logs before sending them to the third component, the logs storage [17].

1.6.1 Vector.dev

Vector.dev is a log collection, transformation and aggregation tool developed by Datadog Inc.. It is written in Rust and distributed either as a standalone binary or as RPM[15] or DEB[16] package. It supports a wide range of input and output types, including those required for the Kubernetes logs collection and other use cases in the environment.

The configuration of Vector can be written in several markup languages, including YAML, which is used in this thesis. It can be split into more files, depending on the developer's preference. The inclusion of the configuration is implicit, as the developer only needs to place all relevant files into the given directory, and Vector automatically merges their contents. An example of the configuration can be seen in Listing 4.

There are three types of configuration items in Vector.dev configuration. The first type are sources; their wide range includes all the common ones such as file or TCP and UDP sockets, but also the Kubernetes source, which reads the *Pod* logs from the Kubernetes nodes file-system and adds queried metadata from the Kubernetes API before sending the log message to the next stage in the pipeline. This metadata includes the name of the *Namespace* of the *Pod*, its IP address, and labels. A special category of sources are internal metrics and logs, which generate data inside the Vector.dev process and do not receive anything from outside. The second configuration item type are transforms, whose relevant types are the aforementioned VRL remap, filter and deduplicate. The last item type in the configuration are sinks, which send the data out of the Vector.dev instance. The first relevant sink type is S3, which is primarily used for archival purposes; the logs written there can be relatively quickly replayed to any new storage type. The second relevant sink type is Elastisearch, as the Bulk API of OpenSearch and Elasticsearch is compatible [18], it can be used to write the logs to the Managed OpenSearch. Additionally, a third type of sink, the vector sink, can be used to forward the logs from the agents to the aggregator with minimal overhead and no reliability issues.

The transformation stages can have several different types, such as filter, lua or remap. The most important of these types is remap; it enables the developer to use VRL to parse and change the contents of each log message. The VRL is a *safe* language used for the transformation of the events, which means that it distinguishes between different

15. Package format used in RedHat family of Linux distributions.
16. Package format used in Debian-based Linux distributions.

```
data_dir: /var/lib/vector

sources:
  file:
    type: file
    ignore_older_secs: 600
    read_from: beginning
    include:
      - /var/log/auth.log
      - /var/log/dpkg.log

transforms:
  label_file:
    type: remap
    inputs: [ file ]
    source: |-
      .tag = "file"
      .host = get_env_var!("VECTOR_SELF_NODE_NAME")

sinks:
  forward:
    type: vector
    inputs: [ label_file, label_kube ]
    address: aggregator.example.local:6000
```

Listing 4: Example of Vector.dev configuration

types of attributes and requires a correctly typed default value for any attribute which can be null. Among other features, VRL provides a wide range of built-in functions for parsing various formats, checking and casting the variable types, and adding or removing attributes from the log message.

Vector includes several tools which simplify the development and testing of transformations and pipelines. These tools include the ability to generate a visual graph of the pipeline and built-in unit testing functionality, using provided input and expected output directly in the configuration. Furthermore, each instance can expose a GraphQL API, which can be used to intercept events in the pipeline and save them to local files on the developer's workstation. This feature simplifies debugging and, combined with the ability to execute VRL scripts on local files, considerably accelerates the development of new pipelines. Similar results can be achieved in the provided web application[17].

The above-listed tooling is especially important considering that other similar tools such as Fluent-bit[18] or Promtail[19] do not provide any comparable options.

17. `https://playground.vrl.dev`
18. `https://fluentbit.io`
19. `https://grafana.com/docs/loki/latest/send-data/promtail`

1.7 Supporting Tools

The section introduces the supporting Kubernetes tooling, which is not a component of the RKE2 distribution and is not an integral part of MinIO deployment, monitoring or logging.

1.7.1 System Upgrade Controller

System Upgrade Controller is a tool recommended by RKE2 documentation; it automates a large part of the upgrade process of RKE2-based clusters. It can automatically cordon a node, disabling the scheduling of any new *Pods*, then execute the given upgrade container image in a *Pod* with escalated privileges and mounted hosts file system. Once the upgrade *Pod* completes, the node is uncordoned and the process is repeated on other nodes. The process is halted if the upgrade *Pod* fails on the node.

Due to escalated privileges of the upgrade *Pod*, the process replaces the RKE2 binary directly in the host operating system and restarts the `rke2-server` process. The disadvantage of this approach is that an in-cluster workload has complete control over the node operating system, which can be easily used for privilege escalation by a potential attacker.

The controller can implement other upgrade processes besides the aforementioned RKE2, but this thesis does not try to implement these due to a lack of tested examples. Each upgrade process is configured as a separate *Plan CRD*. The *Plan* usually contains the desired version, the selector of nodes in the cluster, the concurrency of the process and whether the nodes should be cordoned off or drained during the process.

1.7.2 External DNS

This operator automatically creates and manages DNS records based on the resources in the cluster. In the environment, we are using it with the RFC2136[20] propagation, which utilizes TSIG keys to managed records in given DNS zone. However, a large variety of other propagation methods used in cloud environments is supported, such as Cloudflare, AWS and Google Kubernetes Engine.

In contrast to other above-mentioned operators, External DNS does not add any *CRDs* as it utilizes the *Ingress* and *Service* resources to discover configuration. In the case of *Ingresses*, the `.spec.host` attribute is used as the domain name, the *Services* must have the `.spec.type` set to `LoadBalancer` and have a custom annotation added to get an automatic DNS record. The IP addresses used in the records can either be inferred automatically by the operator from the cluster configuration but are usually explicitly listed in the operator's configuration. An annotation on the resource in question can also override this configuration.

To be able to determine which of the records in the given DNS zone are managed by which cluster, the operator creates a TXT ownership record with a given string, which should be unique to the given operator instance, typically the name of the cluster is used.

20. `https://www.rfc-editor.org/rfc/rfc2136`

1.7.3 Cert-manager

Cert-manager provides an automatic TLS certificate and certificate authority management. This enables the cluster users to issue certificates for endpoints which are only available from a private network and cannot be covered by a trusted certificate from an external authority. It adds several custom resource definitions, the most important of which are explained below.

- **Certificate** – Defines a single certificate-key pair. The key is generated automatically and stored in a *Secret* with a predictable name. The resource must contain a reference to existing *Issuer* or *ClusterIssuer*. Other attributes like duration, DNS name, and common name can also be specified.
- **Issuer** – Defines a way for the `x509` certificates to be created and signed. It can create self-signed certificates, certificates signed by the provided authority key or use an ACME service such as Let's Encrypt[21].
- **ClusterIssuer** – The same as the *Issuer* above, but can be referenced from other *Namespaces* inside the cluster.

1.7.4 Local Path Provisioner

Local Path Storage is an operator which does not provide any additional *CRDs*. Rather, it implements the existing API resource called `StorageClass`. There is no additional redundancy, performance penalty or availability of the volume of other nodes. The volume is only a special directory on the one node wrapped in resource definitions. The operator serves as a placeholder for more complex solutions and allows the deployment of Helm charts, which expect dynamically allocable block storage to be available on the cluster.

21. `https://letsencrypt.org`

2 Analysis and Design

This chapter deals with the analysis of the assets provided for the creation of the object storage cluster and with a high-level design overview of all four major architectural components as outlined in the Introduction.

2.1 Assets

The following section introduces both hardware and software assets which are necessary for the provisioning and operation of the object storage cluster. The rudimentary dependencies such as DNS and NTP servers or existing networking infrastructure are not listed.

2.1.1 Switch

Due to the parameters of the existing network, a special switch used exclusively for the purposes of the object storage cluster was acquired. It has four interfaces for each server: one high-speed 100G interface terminated in QSFP28 port, two metallic 10G interfaces, and one 1G interface dedicated to the out-of-the-band management. Notably, the switch is configured to support 9000 MTU jumbo packets to address performance constraints of additional encapsulation, which might be used in Kubernetes environment. The switch, as shown in Figure 2.1 and Figure 2.4, is an idealized depiction of the actual switch used. Only ports relevant to this thesis are displayed. The configuration of it was carried out by another department and is not the subject of this thesis. To be able to utilize the high-speed storage, the switch should also have several additional 100 Gbps interfaces for clients who are using the storage.

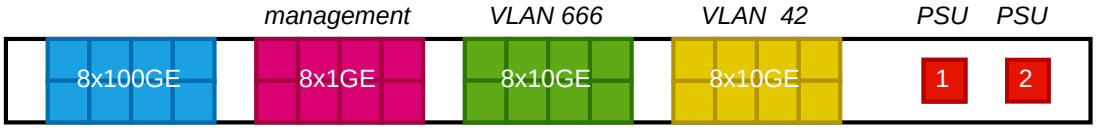

Figure 2.1: The port layout of the switch

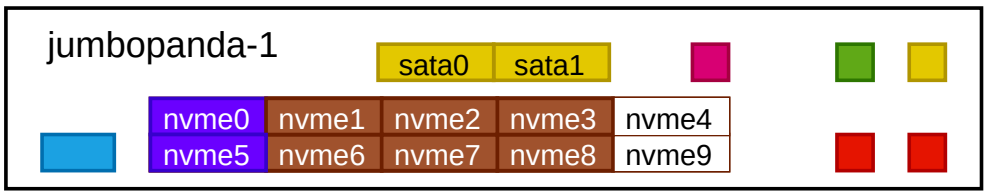

Figure 2.2: The important features of a server

2.1.2 Nodes

For the creation of the cluster, I was provided with four homogeneous servers manufactured by Asus. The most important property of the servers are the drives. Each node has two SATA based SSDs, which are meant to store the operating system. For the applications which are running on the nodes, there are ten additional drives, each with 14 terabytes of fast NVMe storage. Due to restrictions imposed by the architecture of MinIO, it is important that all drives have the same performance and size. Moreover, the disk back-plane and bus support pass-through, meaning that each drive needs to be accessible separately without being included in any type of RAID array.

Figure 2.3: Photo of the actual server assets

The second most important property of the servers are the networking interfaces. Each node has two QSFP28 ports capable of 100 Gbps transport speeds. Furthermore, there are two metallic ten-gigabit ports for access to public and private networks. The last network interface on each node is metallic, one gigabit dedicated to BMC remote management.

Regarding processing power, every server has two Intel Xeon Silver 4310 processors, each with 12 physical cores and 24 threads. The processors are accompanied by 512 GiB of DDR4 ECC memory. Remote management and disaster recovery are possible via the WebBMC interfaces connected via a dedicated 1G interface.

As it might be already apparent, the nodes were selected according to MinIO reference implementation guides [19].

2.1.3 Managed Kubernetes

The managed Kubernetes cluster is not a hard requirement for object storage itself, but it is necessary for reliable monitoring and independent log collection. Furthermore, it is

possible to host the subsection 2.1.5 and subsection 2.1.4 in this cluster, albeit it was not used for these in this particular case.

The assumptions about this cluster are that it runs the same version of Kubernetes, and it has a version of Prometheus Operator, which is compatible with the one used in the object storage cluster. Moreover, it must provide a way to allocate an IP address for *Service* and have a *StorageClass* for allocation of distributed block storage. Lastly, the expectation that this cluster is always available and does not suffer from downtime is maintained.

The monitoring and logging should be placed in this cluster for two major reasons. Firstly, in case of catastrophic failure of the object store cluster, the observability data will remain accessible. Secondly, it prevents cyclic dependencies and prevents the observability stack from interfering with the object storage. Moreover, in another environment, this cluster could be used to host managed Rancher and OpenSearch instances as well.

2.1.4 Managed OpenSearch

OpenSearch is an open-source fork of better-known Elasticsearch, usually used to store semi-structured data such as logs. For purposes of this thesis, I was provided with access to a managed instance.

The provided instance is interfaced with using the accounts, one with write-only permissions and the other with read-only for the web interface of OpenSearch, which is called dashboards. The write-only account is restricted to using Bulk API[18], which the log aggregator uses.

Each of the topics which is introduced in the subsection 2.2.4 is stored in a separate index with a separate index pattern.

The instance administrator configured the ILM policy to automatically rotate the indices after their size reaches 30 GiB in size. After this rotation, the indices are retained for a period of 14 days. The retention period is relatively short due to the large amount of data produced[1] and limited use-cases for said data. The typical use case for the data is debugging and auditing configuration and access control policies.

2.1.5 Managed Rancher

For the purposes of this thesis, the Rancher web interface serves as an access control provider, which allows me to easily generate separate `KUBECONFIG` files for each user of the cluster. The second purpose of Rancher is the easy provisioning of a part of the monitoring stack inside the cluster. This is mainly because both RKE2 and Rancher are developed by the same organization, which allows them to tailor the more general monitoring mix-ins according to the needs of the RKE2 Managed Kubernetes distribution and its components. The instance used is expected to be running on a version compatible with the used RKE2 version.

1. The size is large compared to the space available in the provided OpenSearch instance.

2.2 Architecture

This section describes the high-level architecture of each of the four major areas introduced in the Introduction.

2.2.1 Kubernetes

The overall goal is to create a Kubernetes cluster, which will not restrict the theoretically possible MinIO performance on given hardware and will allow at least some degree of fault tolerance. The design of the cluster can be divided into three areas. Firstly, the number of nodes and their roles. Secondly, the network layout and, lastly, the disk layout.

Node and Roles

The nodes and their roles in the cluster are influenced by two factors: RKE2 architecture requirements and MinIO architecture requirements. The smallest possible highly-available RKE2 cluster has three master nodes because the RKE2 uses etcd as the key-value store for the cluster. Etcd requires a quorum of more than half of its cluster members to be online in order to allow reads and writes, e.g. an etcd cluster with three members requires two of them to be functional, and a cluster with four members requires three members, resulting in the same fault tolerance of one node [20]. This results in an odd number of Kubernetes master nodes in a typical cluster because an even number of master nodes increases the probability that one will fail but does not increase the fault tolerance of etcd. However, MinIO requires an even number of nodes and drives because of parity calculations explained in the subsection 1.4.1.

These two criteria lead to the conclusion that the cluster must have an even number of nodes higher than 3 to achieve any fault tolerance in the cluster. Furthermore, all four nodes must be masters and run each replica of etcd.

Network Layout

The architecture of the cluster's network was dictated by the restrictions of the existing network infrastructure and available hardware. It relies on a single dedicated network switch, which presents both advantages, such as the prevention of any outside influence on the cluster performance, and disadvantages, like the single point of failure in the cluster's architecture. The problems could be resolved in the future by using two switches in an HA configuration, with each node connected by two 100 Gbps interfaces and subnets distinguished by VLAN tags. The nodes are prepared for this eventuality because they have one spare QSFP28 interface each.

As stated in subsection 2.1.2, each node has four active interfaces, and each of these interfaces is connected to a different subnet. The overall physical network configuration can be seen in the Figure 2.4.

The private interface is used for exposure of *Services* and *Ingresses* to the clients, represented by cables with yellow colour. The interface uses the standard 1500 MTU and is capable of 10 Gbps throughput. The IP address assigned to this interface comes from the `10.66.42.0/24` subnet. The public interface, shown in green colour, is only used for

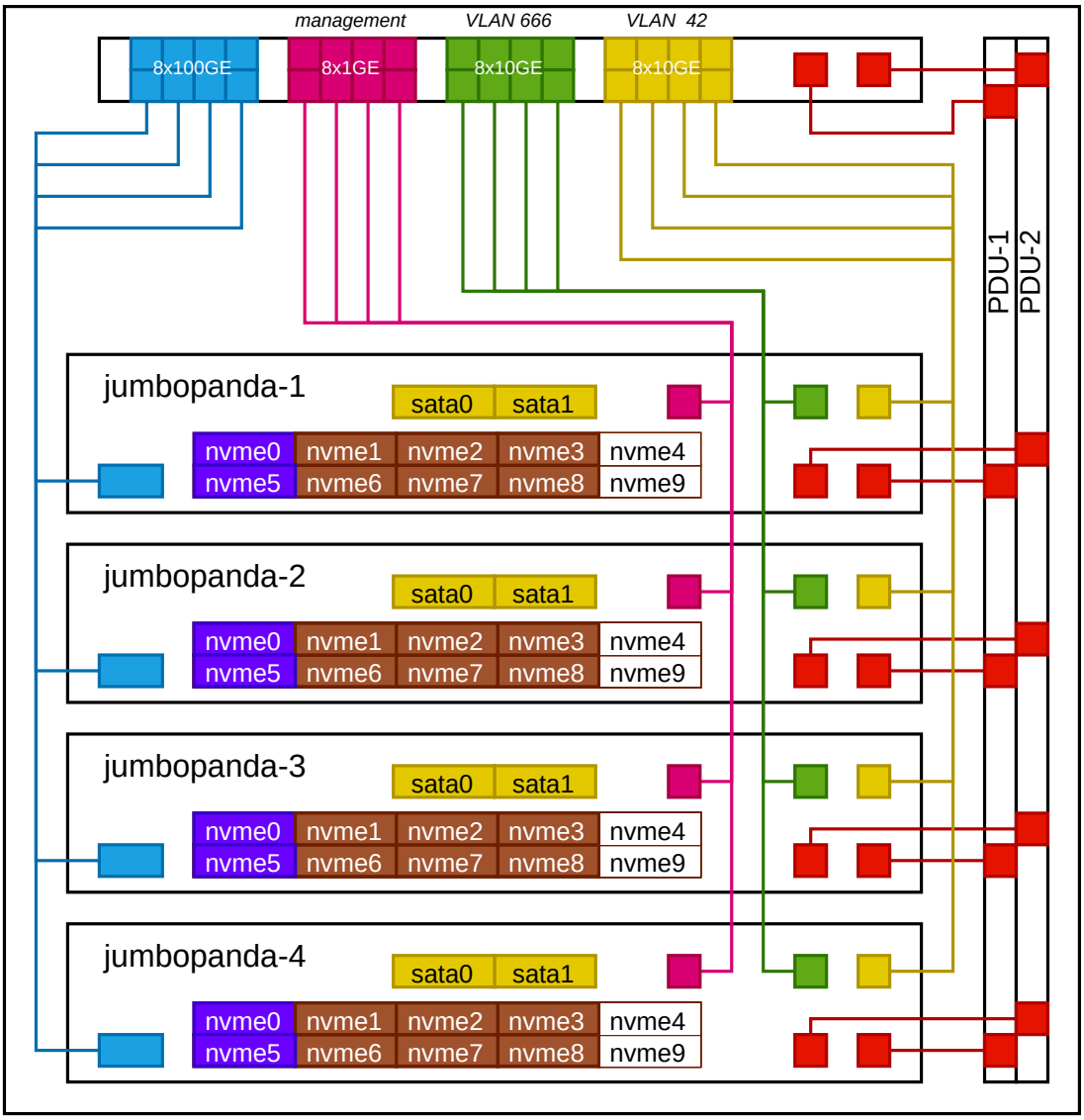

Figure 2.4: Rack layout of the machines in the cluster

access to the Internet, and a firewall blocks any incoming connections. It is physically similar to the private interface but has an IP address from the `147.251.66.32/27` subnet. The internal interface is shown in blue colour and connected to an air-gapped L2 network and serves for communication between the nodes of the cluster. A secondary purpose of this interface is access for high-performance clients; for this purpose, the switch must have an additional 100G QSFP28 ports. The interface is capable of 100 Gbps and uses 9000 MTU. The IP address is allocated from the `172.16.4.0/24` subnet. The last active interface is reserved for remote management, and its specific configuration is not important for this thesis. This interface is represented by magenta-coloured cables.

On the application level, the network is designed in the following manner. The Nginx Ingress Controller is deployed as a *DaemonSet* and uses `hostPort` to listen on

each node, External DNS automatically creates four round-robin DNS A records for each *Ingress* resource. For performance reasons, the *Service* of MinIO API is exposed using `.spec.externalIPs`. This leads to specific requirements for DNS records.

Two sets of DNS records are created, one for the private interface and one for the internal interface. The private records consist of an indexed record for each node private IP address, e.g. `jumbopanda-1.csirt.muni.cz` and one round-robin record which points all four private addresses, `jumbopanda.csirt.muni.cz`. A similar set of records is created for the internal IP addresses as well, but all records are suffixed with `-bootstrap` e.g. `jumbopanda-1-bootstrap.csirt.muni.cz`. It is important to note that the round-robin record should use short TTL, i.e. five seconds, to maximize the load distribution. Furthermore, the round-robin records should, at any given moment, point only to the nodes in a healthy state and part of the cluster. As stated, the public interface blocks any incoming connection and, consequently, does not require a DNS record. The management interfaces do not have a round-robin record. Only individual records are created, e.g. `jumbopanda-1-idrac.csirt.muni.cz`.

Disk Layout

As discussed in section 2.1, each node has ten disks, eight NVMes and two SATAs, which are utilized in the following manner. The two SATA disks are joined in RAID1 for increased system resiliency. These disks are yellow in the Figure 2.4. Two of the NVMes are joined in RAID0 and used for `/var`, which stores temporary file systems of the *Pods*, volumes and any other workloads. Furthermore, the etcd database of RKE2 is also located in the `/var`, increasing the random I/O performance requirements. The `/var` disks are magenta in Figure 2.4. The RAID0 might seem to be detrimental to the goal of cluster resiliency, but it can be changed during the cluster's lifespan, and it is necessary for future use cases of this cluster. This decision was consulted with the advisor of my thesis. Six NVMes are used by the MinIO cluster and shown in brown colour. The last two disks are kept empty in reserve and can be used either to extend the capacity of the object storage or to make the `/var` RAID0 more resilient, e.g. convert it to RAID10 using LVM.

2.2.2 Object Storage

The design of the object storage adheres to the minimum recommendations for a high-performance production deployment rather than high capacity and low-cost deployment. The architecture of the MinIO cluster is relatively simple. It consists of four instances, each with six drives, which exceed the minimum requirement of four nodes, each with four drives. In MinIO documentation, this architecture variant is called multi-node multi-drive.

The communication among the instances goes only through the Kubernetes network overlay, never outside the cluster. The measures taken during the design and implementation of the Kubernetes cluster ensure that the network overlay is capable of almost 100 Gbps of network throughput and that any additional storage layers do not degrade the performance of the drives. These assertions about the performance of the network overlay and drives are tested in the chapter 4. Furthermore, the requirement that each node should have at least 128 GiB of memory and 16 CPU cores per socket is also more

than fulfilled. The recommended network connectivity throughput is 25 Gbps as the minimum and 100 Gbps as the optimum with high-performance drives such as NVMes. The expected combined throughput of the object storage in this configuration is 12.5 Gbps [21]. The guidelines for precise memory size are not taken into account as the calculation depends on the number of requests sent to MinIO, and this number cannot be accurately estimated for our use case [21].

The parity is set to `EC:6` so that failure of two servers or 12 drives on different servers can be tolerated without any data-loss[2]. Furthermore, the write operations can still continue in the degraded state but with significantly decreased storage capacity.

2.2.3 Monitoring

The monitoring stack consists of two parts, one central and one distributed. The central part is deployed in the Managed Kubernetes and can be shared by several clusters.

The central component is responsible for storing the metrics, their visualisation and the execution of recording and alerting rules. It consists of three services: Prometheus, Alertmanager and Grafana.

The distributed part of the monitoring stack consists of Rancher Monitoring Helm chart[3] with a patched *Prometheus* and turned-off unnecessary features. All forwarded metrics are tagged with an additional label `cluster`, and recording and alerting rules here are disabled and replaced with similar rules in the central part of the monitoring stack. The purpose of the distributed component is to collect metrics and forward them to the central component.

The forwarding is done over the remote-write API of Prometheus because of this method retains the time information through transport. An alternative solution would be to re-scrape the metric from the agent Prometheus instance, but this would require the agents to have publicly available endpoints and would destroy the original scrape-time information. This design requires the distributed Prometheus to be running in so-called agent mode, which disables its capability of executing alerting and recording rules as well as the ability to query data from a given Prometheus agent instance. If the distributed Prometheus remained in its standard mode, the remote write API forwarding metrics to the central Prometheus instance would suffer from frequent out-of-order errors and become unreliable.

This division into two components is based on the subsequent reasoning. The configuration discovery of a large number of metric endpoints distributed across clusters is difficult in the case of only one central Prometheus instance. Moreover, it would negate the most significant benefit of the Prometheus Operator, the automatic configuration discovery. It is also desirable for the monitoring to be independent of the machines being monitored, and the centralized location for configuring dashboards and alerts greatly simplifies future changes and development.

Both the agent and the central instances of Prometheus along with the central Alertmanager are managed by Prometheus Operator. More complex Prometheus-compatible

2. `https://min.io/product/erasure-code-calculator`
3. `https://github.com/rancher/charts/tree/main/charts/rancher-monitoring`

federation solutions such as Mirmir[4] or Thanos[5] were considered but deemed unnecessarily complex for the amount of metrics in this use-case [22] [23].

2.2.4 Logging

The logging stack is divided into two components, similarly to the monitoring stack. The area covered by each component adheres to the aggregator-agent pattern [24]. To make the development of the pipeline easier, the parsing, transformation and other logic is concentrated in the aggregator component. The agents are kept as simple as possible and only read log files or listen on endpoints. This asymmetric architecture also allows for the agents to use only one communication channel to the aggregator, minimizing the potential attack surface.

The aggregator component is not highly available because the Managed Kubernetes does not provide a load balancer, which would allow active-passive replica failover. The available round-robin load balancer is unsuitable for the aggregator because the transformations with message deduplication or reduction do not work predictably on multiple replicas simultaneously. Furthermore, sinks which do not combine the data after writing, e.g. S3, would contain approximately twice the amount of chunks because each replica would write the batches separately.

The logs are to be collected from three main areas: firstly the host operating system of nodes, secondly the components of the Kubernetes cluster and lastly from the services running inside the cluster, which are currently represented only by MinIO. These varied sources require basic normalization of the log messages and their division into several topics, which are discussed in subsection 3.4.1. Vector.dev uses a separate buffer for each sink. The buffers can either be stored in memory or on disk. Due to resource and reliability constraints, both sinks of the aggregator are configured to use the disk as buffer storage. The agents use only an in-memory buffer as all their sources are local and will retain events even when the agent forwarding location is down.

The log messages are sent to the sinks in batches. For the compression to be more efficient on the S3 sink, batch timeout is increased from the default ten seconds to five minutes. This leads to increased memory demands because the batches are built in the memory in their uncompressed state. In the case of this thesis, there are two log storage locations, similarly to the architecture pattern shown in Figure 2.5. The first location is an S3 object storage, and the second location is an Managed OpenSearch cluster. Both these locations should preferably deployed in another cluster. However, if the resources are restricted, they can be deployed in the same cluster, with the caveat that the complete failure of the cluster will mean the loss of observability data and prevent the administrators from finding the cause of this failure.

Even with the aforementioned measures in place, it is necessary that the applications in the cluster are configured with a reasonable log level, e.g. `info`, because even a single high-traffic service can easily flood the logging pipeline with superfluous messages.

4. `https://grafana.com/oss/mimir`
5. `https://thanos.io`

3 Deployment

The chapter details the initial provisioning of the cluster and the configuration of all necessary components. The first section of this chapter deals with the deployment of the cluster itself and the supporting operators, as well as other components. The second section describes the deployment of the object storage in the cluster. The third details the configuration of the monitoring stack of this cluster. The fourth details the centralized collection of logs from the object storage and supporting components. The fifth describes the operational procedures required for the updates and maintenance of the object storage cluster.

3.1 Cluster

This section describes the configuration of the nodes in the cluster, installation of the operating system, provisioning of RKE2 of these nodes and finally, the registration of this cluster in Managed Rancher with the subsequent deployment of supporting operators and controllers.

3.1.1 Operating System

As the first step before the installation of the operating system, several adjustments need to be made in the BIOS; these changes are not described in detail, as they are specific to particular hardware and not crucial for the goals of the thesis. The changes include setting the IP address of BMC, changing the default admin password and disabling any

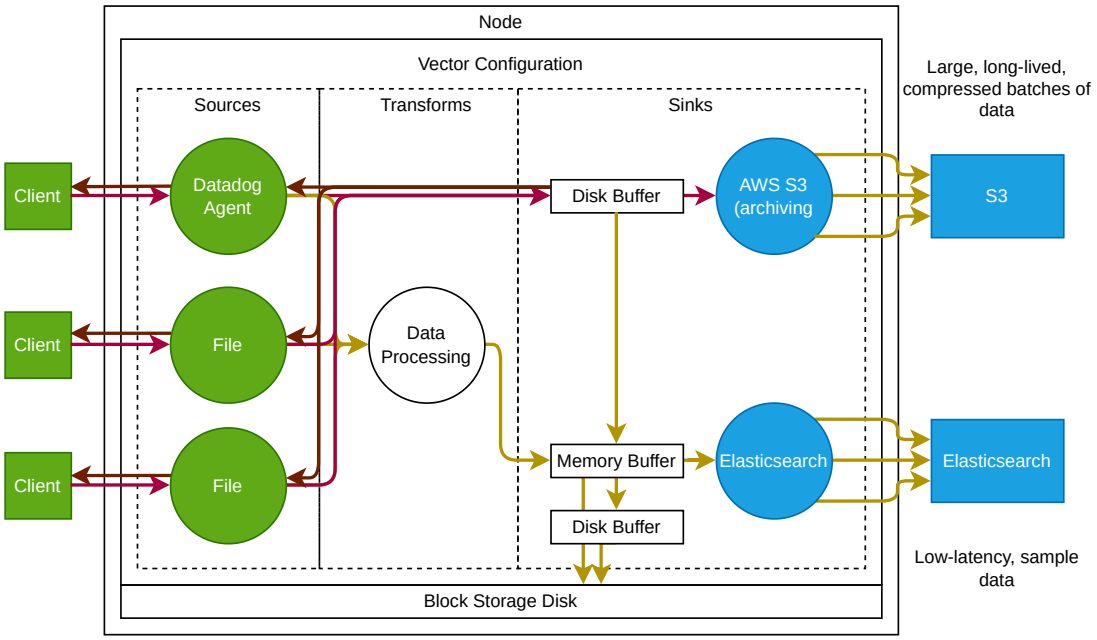

Figure 2.5: Vector.dev architecture with separated archival and analytics [24]

other user account present in the factory settings. Any remote management features which are not going to be utilized should be disabled as well. This includes SSH service on the BMC, RedFish API.

The installation of the chosen operating system, Ubuntu server 22.04.3 LTS, is done manually because the environment makes fully automatic provisioning impractical. Furthermore, the effort necessary to create supporting infrastructure for net-booting and fully automatic configuration provisioning would outweigh the repeated effort needed to (re-)install the nodes manually at this small scale. A further argument against the usage of cloud-init[1] is that any possible future clusters will likely have slightly different hardware configurations.

The guide through the installer is relatively simple. The language is set to US English, and so is the keyboard layout. All names of the default user are set to the value `csirt`, and the user's password is now chosen to be memorable and easily entered on the keyboard in the virtual console. The only IP address to be configured through the installer is the private address of each node (`10.66.42.xxx`), in order to make the system accessible over the network.

The most significant part of this process is the configuration of the disk partition layout. The two SATA-based SSDs are joined in RAID1, this virtual device contains 1 GiB `/boot/efi` formatted as FAT32, 2 GiB `/boot` and the rest of the available space for `/`. Two of the NVMe drives are joined in LVM stripped mode to double the capacity of the volume, which is mounted as `/var`. The three aforementioned partitions are formatted as EXT4 file systems. This is visualized in the Figure 2.4. Out of the remaining eight NVMe drives, six should be formatted as XFS and labelled disk1-6.

Package `openssh-server` must be added at the last step of the installer before rebooting the server. Additional network configuration is done after the reboot via Netplan[2] according to the architecture proposal of the cluster from chapter 2.

3.1.2 Ansible

The deployment process is automatized via an Ansible playbook with two roles. The product of the playbook executions is an empty Kubernetes cluster consisting of nodes with a slightly hardened operating system. Additional components, such as operators, are deployed manually in the subsection 3.1.4.

The playbook in the attachment of the thesis is configured for the repetitive runs on an existing cluster. The initial provisioning of a new cluster requires a few changes and has additional requirements. As the first step, the playbook is executed only on the first node using the `ansible-playbook -l jumbopanda-1 jumbopanda.yml` and the future round-robin DNS record exists only with the IP address of the first node. This is done so that the new nodes do not resolve the registration address to an IP where the Kubernetes API is not listening yet. Secondly, the registration token outputted by the last task in the playbook needs to be set as `rke2.token` variable for the other nodes, replacing the current placeholder.

1. Cloud-init is an automatic provisioning tool used by Ubuntu.
2. `https://netplan.io`

Role Core

š This Ansible role performs the basic configuration of the Ubuntu server, installs common tools used by the administrators and applies rudimentary hardening to the system. The password prompt for `sudo` command is enabled, the default user's password is changed to something stronger and unique for each node, and a list of provided SSH public keys is added to the default user's authorized keys.

The common SSH configuration hardening is applied. This includes disabling login using passwords or root accounts, enabling verbose logging and disabling unnecessary features such as tunnelling or port forwarding. Furthermore, it prohibits the usage of obsolete and unsafe ciphers[3]. The configuration is based on internal security policy and the Mozilla InfoSec guidelines[4].

The administrator toolkit added by the role consists of sensible bash defaults[5], allowing for easier manipulation with shell history and a list of chosen tools.

Among the Kubernetes related changes is the disabled SWAP. SWAP is undesirable due to possible performance impact on MinIO. IPv6 networking stack is disabled in the system via a kernel parameter. This is done because it greatly simplifies networking configuration and any potential debugging. Moreover, the IPv6 is not required for the functionality of the cluster, which is accessible only out of a few specified networks. NTP time synchronization is configured, and several undesirable packages are removed. The undesirable packages include Ubuntu telemetry and Snap[6]

The role is configurable by several variables shown in Listing 5. After execution of this role, the nodes should be rebooted. The playbook does not do this because it should be written to run against an existing cluster to enforce or update the applied configuration; in other words, it should be idem-potent. Automatically restarting nodes is undesirable in all playbook runs except for the first.

Role RKE2

This role is idempotent and automates the installation and hardening process of RKE2; for this purpose, it utilizes the installation scripts provided by its developers. An override for the hardening profile was created for practicality purposes; it downgrades the pod security admission policy from enforcement to only warnings. The last tested version of RKE2 is `1.26.6-rke2r1`. The role supports the following configuration options:

3. `https://github.com/jtesta/ssh-audit`
4. `https://infosec.mozilla.org/guidelines/openssh`
5. `https://github.com/mrzool/bash-sensible`
6. Snap is application containerization and packaging tool, developed by Canonical Ltd..

```yaml
core_hostname: "{{ inventory_hostname }}"
core_user: "{{ ansible_user }}"
core_user_pass: "changeme"

core_user_ssh_keys: ""

core_ssh_allowed_users:
  - "{{ ansible_user }}"

core_time_providers:
  - ntp.muni.cz

core_time_fallback_providers:
  - tik.cesnet.cz
  - tak.cesnet.cz

core_tls:
  cert: ""
  key: ""
```

Listing 5: Default variables of Ansible role core

- `rke2_host` – A string with the host to which the `rke2-server` binds should be set to the address on the internal high-speed network interface.
- `rke2_init` – A boolean, if set to `true`, the registration variables are ignored, and the new empty cluster is initialized. It should be set to `true` only on the first playbook run on the first node of the cluster.
- `rke2.version` – A string with desired version of RKE2, the version must be in the following format, `1.26.6-rke2r1`.
- `rke2.url` – A string with node registration URL, which should be located at the round-robin DNS record with `-bootstrap` suffix, which points to all existing nodes.
- `rke2.token` – A string with node registration token.
- `rke2_s3` – A dictionary with the optional configuration of etcd backups to another S3 object storage. The backups are kept only locally on each node if the variable is not specified.
- `rke2_tls_san` – A list of strings, with the SANs to the certificate of the Kubernetes API. This variable must be set to the same value on all nodes. Furthermore, it must include `jumbopanda-bootstrap.csirt.muni.cz` record.
- `rke2_labels` – A list of dictionaries with labels and their values to be applied to the node.

- `rke2_taints` – A list of dictionaries with taints and their values to be applied to the node.

Calico network overlay packet encapsulation is replaced with BGP mode, and consequently, the MTU of the overlay network can be set to 9000. This would not be possible with enabled encapsulation. Moreover, the node address detection is fixed to the internal address of the node, which is in turn set to the `rke2_host`. Because of the firewall configuration, which is discussed later, the fail-safe port ranges of Calico are emptied.

The role implements several sysctl parameter optimizations, which are divided into three groups. The first is the parameters recommended by the MinIO reference implementation guide [19]. The second adds several settings recommended for high-speed networking[7], most of which were already included in suggestions from MinIO. The third is a custom addition, which resolves the issue encountered during the initial testing involving the *Too many files open* error message, which seems to have been caused by the combination of MinIO and log collection, both of which utilize atypically large amounts of comparatively small files.

Then, the maximum count of *Pods* on a single node is doubled from 110 to 220. A Kubernetes audit policy logging any resource modification of resources in the cluster is also added.

3.1.3 Rancher

The newly created cluster must be registered in the Managed Rancher instance. The process is manual and consists of two steps. The first is the retrieval of the registration address from the Rancher web interface at the *Cluster Management* > *Import Existing* > *Generic* path. The second is the execution of the registration command on one of the master nodes in the cluster6.

After the registration command is executed, components for user management are provisioned in the cluster. This process typically takes approximately three minutes, after which two new *Namespaces* should appear both names are prefixed with `cattle-` keyword, and all *Pods* inside should be either in `Running` or `Completed` state.

```
export KUBECONFIG=/etc/rancher/rke2/rke2.yaml
export PATH=$PATH:/var/lib/rancher/rke2/bin

curl -i -sfL https://rancher.../... | kubectl apply -f -
```

Listing 6: Cluster registration into the Rancher

3.1.4 Operators and Controllers

This section describes the operators and related resources which are added to the cluster after its provisioning. The steps are not automated and require some manual intervention. The additional resources for the installation of the supporting operators from

7. `https://fasterdata.es.net/host-tuning/linux/test-measurement-host-tuning`

section 1.7 are compiled into one Git Kustomization repository, called `baseline`, which is further divided into four Kustomizations each for one of the operators. The configuration of the Calico-based firewall is stored in a separate Kustomization repository called `calico-firewall`.

The System Upgrade Controller is installed using the official resource bundle, and the last tested version is `0.13.1`. The operator resources are located in `system-upgrade` *Namespace*. The additional *Plan* resource is added to the same *Namespace*; it contains the following upgrade configuration. The nodes are only cordoned off before the beginning of the upgrade process. All nodes in the cluster are selected for the upgrade. The concurrency of the upgrade process is set to one due to availability concerns. The upgrade image and *ServiceAccount* are set to the values suggested by the RKE2 documentation.

The Cert-manager installed using the official resource bundle, the last tested version is `1.13.2`. The operator components are placed into the `cert-manager` *Namespace*. The only additional resources are two *ClusterIssuers*, one issuing self-signed certificates and the other issuing Let's Encrypt signed certificates. The Let's Encrypt issuer is using the ACME solver for DNS challenges with TSIG keys.

The Local Path Provisioner references the official install bundle; the last tested version is `0.0.24`. All operator components are placed into the `local-path` *Namespace*. The only change to the default configuration is that the path which is used for data is set to `/var/local-path-provisioner`. This is done because of the disk layout introduced in section 2.2.1.

The External DNS is deployed in a slightly different manner because no official up-to-date resource bundle is provided. An example from the official repository was adopted for my environment. It consists of RBAC resources, a *Secret* with TSIG key and a *Deployment*. The address of the DNS server is hard-coded into the arguments of the *Deployment*. The last tested version `0.13.6`.

The next item to be configured is the firewall implemented using Calico CNI. In the course of the initial testing, it was discovered that a simple IPtables rules-based firewall is neither suitable nor feasible due to the fundamental incompatibility with Calico. Consequently, the firewall rules had to be implemented via the Calico provided *GlobalNetworkPolicy* resource in combination with *HostEndpoint* resource, which represents the interfaces to be protected by the firewall.

The `HostEndpoint` resources are specified only for private and public interfaces; the internal high-speed interface is not covered by the firewall, nor are any other interfaces which do not have a corresponding `HostEndpoint` resource defined. The *GlobalNetworkPolicies* are divided into two groups based on which interface they are applied to. The public group allows only outgoing connections on the public interface. The private allows outgoing connections, incoming from Masaryk University network on the following protocols, HTTP(s), incoming SSH and MinIO API.

The last adjustment needs to be made to the CoreDNS component of RKE2. During the cluster benchmarks, which is described in a later chapter, it was discovered that the amount of DNS queries generated by MinIO is too high for default-sized CoreDNS *Pods*. Therefore, the default resource allocation was increased manually to request two CPU cores and a limit of four CPU cores. Similarly, the memory request and limit were both set to one gigabyte. The modification is done using the `kubectl get helmchart -n kube-system rke2-coredns` command because it requires change to a resource

managed by the RKE2 installer, and doing it in the Ansible role would complicate future updates.

3.2 MinIO

The implementation of the MinIO cluster consists of a single Git repository with Kustomization. The overall relations between the resources in the aforementioned repository can be seen in Figure B.1. The fundamental architecture idea is based on the currently unmaintained official Helm chart but is tailored to the specific environment and available hardware. The last tested release of MinIO is `RELEASE.2023-09-30T07-02-29Z`. The deployment requires an extra step with the creation of the monitoring token, which cannot be statically defined; this is shown in Listing 7.

```
export MC_HOST_jumbopanda=https://xxx:xxx@jumbopanda.csirt.muni.cz:9000
mc admin prometheus generate jumbopanda
```

Listing 7: Monitoring authentication token generation

The main resource is a *StatefulSet* with four replicas. Each of the replicas has mounted six volumes; these volumes are mapped to the XFS formatted drives. Each physical drive has a single *PersistentVolume* resource with a given path to the drive mount-path with a label-based affinity to the particular node. An abstract *StorageClass* named *xfs* is created to map the *PersistentVolumes* to *PersistentVolumeClaims*, which are created according to the *volumeClaimTemplate* in the *StatefulSet*. This *StorgeClass* also guarantees that each *Pod* of the *StatefulSet* has its volumes allocated on one node, not on multiple nodes.

Each *Pod* has resource requests with 24 CPU cores and 128 GiB of memory. The resource limits are set to 30 CPU cores and 128 GiB of memory. This resource allocation is slightly higher than the recommendations discussed in chapter 2 and was influenced by future workloads, which are planned to be scheduled on the same cluster. During the benchmarks described in the later chapters, these limits did not bottleneck the expected performance of the object storage.

The configuration of the MinIO process is done via environment variables injected from a *Secret*. The *Secret* is created by a `.secretGenerator` in the Kustomization file. This configuration includes the MinIO audit log webhook location on the Vector.dev agents.

The MinIO *Pods* have two TLS certificate-key pairs mounted as volumes, common for all four *Pods*. One of them is managed by Cert-manager, and the other is injected from `/files` into a *Secret*. The injected one is signed by a trusted authority and is used for communication with the API and console by clients. The generated one is signed by an internal authority and is used for communication between the *Pods* of MinIO cluster. The domains required to be present on both external and internal certificates are listed in Listing 8.

The logic certificates and their usage is convoluted because of a two-layered issue caused by the inconsistent certificate reload introduced by MinIO. According to the documentation, MinIO should be able to reload certificates while running. However,

```
# external certificate
jumbopanda.csirt.muni.cz
jumbopanda-1.csirt.muni.cz
...
jumbopanda-4.csirt.muni.cz
jumbopanda-bootstrap.csirt.muni.cz
jumbopanda-1-bootstrap.csirt.muni.cz
...
jumbopanda-4-bootstrap.csirt.muni.cz

# internal certificates
minio.minio.svc
minio-hl.minio.svc
minio-0.minio-hl.minio.svc.cluster.local
...
minio-3.minio-hl.minio.svc.cluster.local
```

Listing 8: The required domains of MinIO TLS certificates

the practical testing revealed that this happens only partially or not at all. Resulting in a variety of TLS errors when the *Pods* forming the MinIO try to communicate with each other after the internal certificate was re-issued by the Cert-manager.

The first layer of this issue is that MinIO must trust the certificate of the internal CA. However, the *Certificate* is reissued every 60 days by the Cert-manager if the internal CA is of the self-signed type. This leads to the need to manually restart the *Pods* every 60 days. To be able to set the expiration of the CA certificate to a longer period, one level of indirection needed to be added. Firstly, an *Issuer* of the self-signed type is created, then a *Certificate* with the attribute `.spec.isCA: true` and `.spec.duration: 20y` is issued by this *Issuer*. Secondly, the *Secret* of the previously issued *Certificate* is used to create a new *Issuer*, which then issues the *Certificate* with `.spec.duration: 5y` used by the MinIO *Pods*. This logic is visualized in the Figure 3.1.

However, after deploying the above-described logic, it was discovered that the Cert-manager can re-issue *Certificates* without any warning for many reasons other than the expiration of the certificate, the problem is mitigated by the usage of `.spec.initContainer`, which copies the TLS certificates and keys into an `EmptyDir` volume, shielding the MinIO process from any changes during runtime.

MinIO process inside each *Pod* listens on two ports: on 9001 TCP is the web console, and on 9000 TCP is the S3 API. There are three *Service* resources defined, one normal with a *ClusterIp* address, and one headless for purposes of referencing each replica of the *StatefulSet* separately. The third *Service* uses `.spec.externalIPs` attribute the expose only the S3 API on port 9000 TCP on each node in the cluster. This is done due to performance bottleneck, which would be caused by adding an *Ingress*. The web console has no performance impact and is exposed as a *Ingress*. It is important to note that load

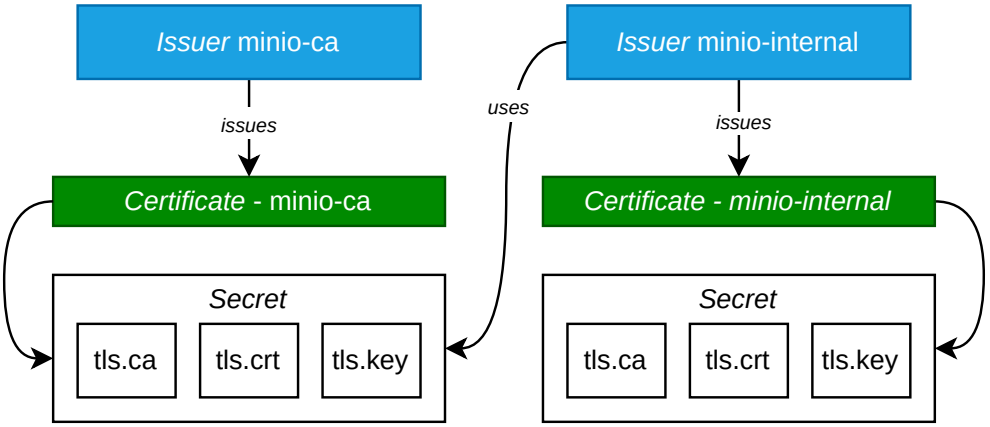

Figure 3.1: MinIO TLS certificate logic

balancers such as MetalLB[8] would cause the same throughput bottleneck as an *Ingress*. MetalLB has two modes of operation: L2, which propagates the addresses using ARP announcements and BGP, which uses ECMP routes. The L2 mode restricts the overall throughput of the *Service* to the maximum throughput of one node. The BGP mode does not have this restriction, but unfortunately, the environment in which our cluster is deployed does not support such setup [25].

All four pods are covered by a *NetworkPolicy*, which further restricts the communication with the API to only necessary subnets and chosen *Namespaces* inside the cluster. From outside of the cluster, the S3 API is visible on two distinct endpoints, which should be used depending on the location of the client. All whitelisted clients can access the S3 API on the `jumbopanda.csirt.muni.cz:9000`, but the network throughput is limited to 10 Gbps. For high-speed clients located in other clusters with the same internal switch, the alternative address is `jumbopanda-bootstrap.csirt.muni.cz:9000`, this address provides the full 100 Gbps network throughput.

This chapter ends with a brief summary of reasons why the available MinIO operator[9] was not used. The operator deployed a previous testing iteration of S3 object storage, but the version of the operator frequently introduced regressions, which required unnecessary debugging and had a negative impact on the reliability and stability of the storage. Version 5 of the operator removed almost all additional features, such as audit log querying and metric collection, which now need to be added manually and are not managed by the operator. The aforementioned bugs and frequent changes in the core functionality of the operator led to the conclusion that the manually managed MinIO cluster will be easier to maintain and repair if needed.

For the same reason, the drives used by the MinIO cluster are provisioned manually instead of DirectPV operator[10], which is developed by the same team and has similar issues.

8. `https://metallb.org`
9. `https://github.com/minio/operator`
10. `https://github.com/minio/directpv`

3.3 Monitoring

As discussed in the chapter 2, the monitoring stack is divided into two components, as shown in Figure 3.2. One is called *distributed*, it contains mostly the exporters and the configuration discovery for metrics endpoints. This component is deployed on the object store cluster. The other component is called *central* and deployed is in the Managed Kubernetes in one *Namespace*; it includes instances of tools needed for metrics storage, querying, visualisation and alerting. This is done to keep the monitoring available even in case of partial or complete failure of the object store cluster. The *distributed* component uses the Rancher Monitoring Helm chart[11]. The central monitoring stack is defined as one Git repository with Kustomization. The

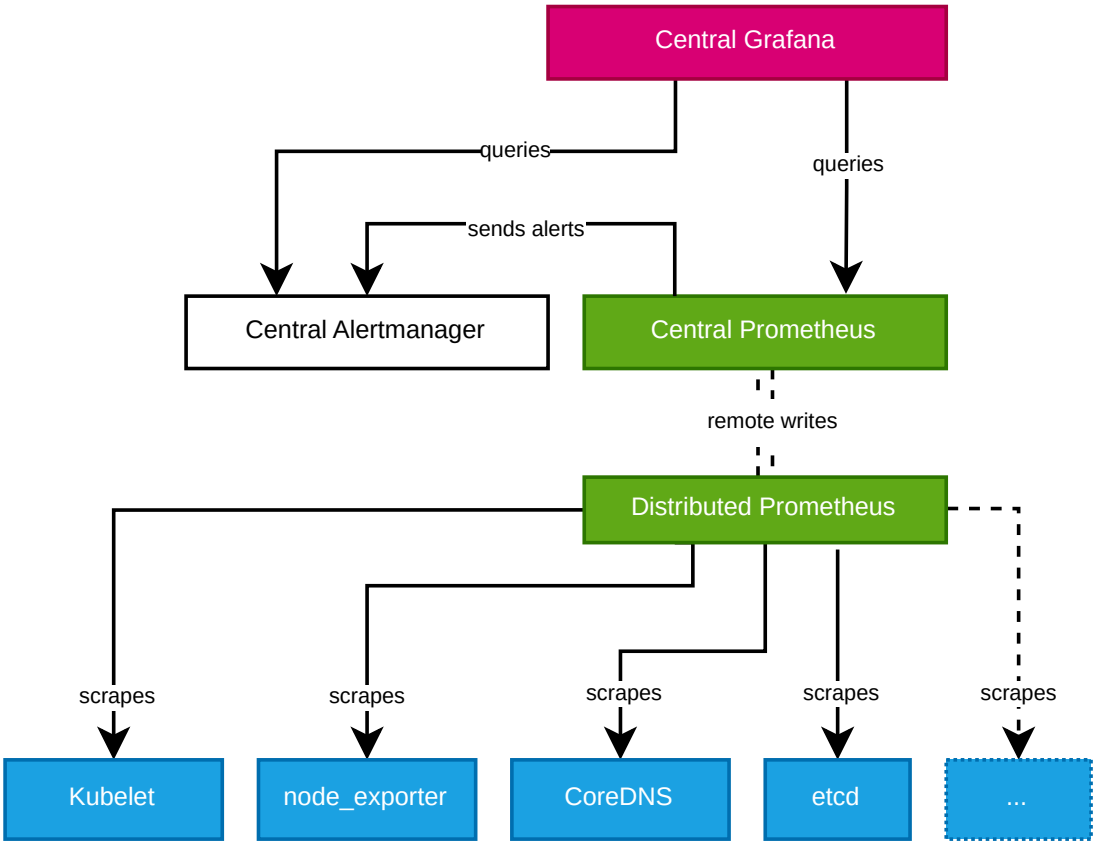

Figure 3.2: Graph of metrics collection pipeline

The *central* component is enriched with the Prometheus alerting, recording rules and Grafana dashboards from the Rancher Monitoring Helm chart. These resources are not copied to the repository but are linked remotely as discussed in the possibilities of Kustomization. Furthermore, this component is self-monitored, which is not ideal, but the probability of the object storage cluster failing at the same time as the Managed Kubernetes is deemed low and the risk acceptable. The repository containing the resource definitions for the central monitoring component is slightly different

11. `https://github.com/rancher/charts/tree/main/charts/rancher-monitoring`

from a typical Kustomization. All resources are divided into `resources/stack/` and `resources/content` directories. This separates the rules, sources and other content from the definition of the stack itself and allows the future expansion of metric sources and alerting rules without reducing the readability of the code. TLS certificates used by the services are not managed by Cert-manager, but are generated externally and injected via Kustomization `secretGenerators` from the `files/` directory. The most important Kubernetes resources used in the central component are shown in Figure B.2, the remotely linked *ConfigMaps* and *PrometheusRules* are omitted for the sake of clarity.

The expected version of the Prometheus Operator is determined by the version which is available in the Managed Kubernetes, `0.59.1`.

3.3.1 Distributed Exporters

This component removes the necessity to manually define and maintain the *ServiceMonitors*, *PodMonitors* and *Probes* for all metric endpoints of the cluster components. It provides a complete monitoring stack, similar to the centralized one. However, the majority of its features is not utilised due to the reasons listed in the chapter 2.

Several important changes need to be made in the default `values.yaml` of the chart; these are shown in the Listing 9. Furthermore, a *Secret* with the keys `username` and `password` must be created in the `cattle-monitoring-system` *Namespace*, the credentials in it should match the ones created in subsection 3.3.3. The effects of the modifications include the disabling of the included instances of both Grafana and Alertmanager as the functions of these services are implemented in the *central* component of the monitoring stack. The values, which are used for the templates of the included *Prometheus CRD*, need to be patched so that the instance is running in *agent* mode. This mode disables the discovery and execution of any recording and alerting rules and optimizes the instances for the role of the metric collection agent.

The agent mode prevents out-of-order errors, which were encountered during testing without having the instance switched to agent mode. The errors essentially paralysed the central Prometheus instance as it was unable to write the metric samples, which it scraped with the correct time, making the WAL overflow to the extent in which the Prometheus, even with extensive resource could not read the WAL before the readiness probe timed out and the cluster sent `SIGTERM` to the unready Prometheus *Pod*.[12] [13]

The last tested version of the chart is `102.0.0+up40.1.2`, deployed from Managed Rancher version `2.7.9`.

12. `https://github.com/prometheus-community/helm-charts/issues/1519`
13. `https://github.com/prometheus-community/helm-charts/issues/2506`

```yaml
prometheus:
  prometheusSpec:
    enableFeatures:
      - agent
    retention: ""
    retentionSize: ""
    walCompression: false
    remoteWrite:
      - url: https://prometheus.csirt.muni.cz/api/v1/write
        basicAuth:
          username:
            name: prometheus-remote-basic-auth
            key: username
          password:
            name: prometheus-remote-basic-auth
            key: password
        writeRelabelConfigs:
          - action: replace
            targetLabel: cluster
            replacement: mintaka-storage
defaultRules:
  create: false
grafana:
  enabled: false
alertmanager:
  enabled: false
```

Listing 9: Rancher Monitoring chart `values.yaml` patch

3.3.2 Central Grafana

The Grafana instance is relatively small, both in terms of resources and its topology. It is defined in the same Kustomization repository as the rest of the central component of the monitoring stack. The resource requests of a single replica are set to 128 MiB of memory and one eight of CPUs, with the limits set to one CPU core and 1 GiB of memory. The last tested version is `10.2.0`.

The definition of the *Deployment* resource is appended with a large number of volumes since the pre-made dashboards from kube-prometheus[14] repository are defined each in one *ConfigMap*. The user data, user-made dashboards and alerts are stored in 1 GiB *PVC* volume. The main configuration file of Grafana is called `grafana.ini` and contains mostly the default configuration with the following notable exceptions. The default administrator credentials and the URL used by the *Ingress* are set to the values appropriate for the environment. The OpenID Connect authentication is configured to make the management of users easier. Further adjustments only affect the expected

14. `https://github.com/prometheus-operator/kube-prometheus`

locations of the pre-made dashboards, nothing else. The built-in SQLite database is sufficient for the deployment of this scale as demonstrated by the Rancher Monitoring Helm chart and stated previously.

Four more dashboards are added to the Grafana instance manually. Three are pre-made and published by the MinIO developers and visualize the metrics generated by the object storage cluster. The first of the dashboards is called *MinIO Dashboard* and visualizes general metrics focusing on cluster health and resource utilization. *MinIO Bucket Dashboard* focuses on metrics, which can be partitioned by the bucket name. The *MinIO Replication Dashboard*, currently, this dashboard does not have any purpose because replication is not being used due to the lack of a suitable sufficient. The fourth is a custom dashboard for Vector.dev aggregator, shown in Figure B.4.

Two data sources are added manually for Prometheus and Alertmanager. The addresses used are of the *Services*, not *Ingresses*; this removes unnecessary complexity of authentication and TLS certificates. The Alertmanager data source does not allow querying of any data. Rather, it is used to extend and audit existing alerting rules of given Alertmanager instance. OpenSearch is not added to Grafana as a data source, even though it is possible, because the included web interface offers a wider range of features for the work with semi-structured log data.

3.3.3 Central Prometheus

As previously stated, the Prometheus is deployed using the Prometheus Operator into the Managed Kubernetes cluster, with the *CRD* set to a single replica. The cluster disallows the usage of cluster-wide RBAC resources, allowing the users only the usage of namespaced RBAC resources.

Therefore, the included *ServiceAccount* has RBAC, which allows only the discovery of configuration inside the same *Namespace* as the Prometheus *Pod* is deployed in. This namespaced nature of the central Prometheus is also reflected in the label selectors used for the discovery of its *CRDs* such as *PrometheusRules* and *Probes*. The retention is set to 30 days, which was deemed optimal for operational usage of the metrics. This corresponds with the 256 GiB *PVC* bound to the instance. The instance is exposed by a *Ingress* resource with two sets of BasicAuth credentials, one for the administrators in case they need to access the web interface for debugging and a second for the remote write clients.

The seemingly high resource limits do not correspond with the typical resource usage of the instance, but they are necessary during the startup of the Prometheus as the WAL is being processed, and all remote write clients will try to send all metrics collected since the last contact at once. This is further exacerbated by the issue described in subsection 3.3.1.

Additional services, including non-Kubernetes based services, can be added to the same central monitoring stack via *Probes*. These additions should adhere to the following rules. Firstly, all metrics endpoints should be authenticated and use HTTPs. Secondly, no service should have public endpoints added just for the purpose of metric collection. If the service is private and needs to be monitored, it should use a metric collection agent and push the metrics to the central Prometheus. These *Probes* should be all contained in the central monitoring Kustomization repository so that the entire

stack is clearly versioned and easily recoverable. The last tested version of Prometheus is `2.47.0`.

3.3.4 Central Alertmanager

The Alertmanager instance consists of a simple Alertmanager *CRD* set to the version `0.26.0` and three replicas. It is accompanied by a *Service*, a *ServiceAccount* with no bound RBAC and a *NetworkPolicy*, which blocks any incoming connections except for Prometheus and other Alertmanagers. The operator creates a small 64 MiB *PVC* in which the state of the Alertmanager instance is stored. The web interface of the instances is not published as an *Ingress* because Grafana provides sufficient alert configuration options.

The most important resource is a custom notification template stored in a *ConfigMap*. It is compatible both with the pre-made alerts and the newly created alerting ones, whose structure is described in the next section. The aforementioned template was tested with a Mattermost[15] instance and should be compatible with Slack[16] as well.

The other aspect worth noting is that the alert group internal is set to five minutes, meaning that the alerts generated by the same rule within a five-minute window is sent as one notification. This behaviour prevents unnecessary spam but adds a delay between the alerting rule triggering and the notification being received by the administrator. This delay did not prove to be a problem during evaluation.

3.3.5 Alerting Framework

Prometheus alerting rules are flexible in their definition, necessitating the existence of a framework which defines the common set of labels and annotations for all alerts. These rules simplify the design of a common notification template, which was deployed in Central Alertmanager. The following suggestions are loosely based on the pre-made alerting rules from the kube-prometheus repository[17].

- Annotation **summary** – A one-sentence general description of the issue containing no metric label variables. This is shared by several instances of the alert firing together.
- Annotation **description** – A more specific, parameterized description of the concrete instance of the firing alert.
- Annotation **runbook_url** – A link to the chapter in our run-book, which contains the previous solutions to the issues which triggered this alerting rule.
- Annotation **dashboard_url** – A link to Grafana dashboard, which contains visualisations of the metrics which are used in the alerting rule. Typically, many alerts share one dashboard. This URL replaces the links to particular alert rule definitions, which is impossible with only Grafana exposed to the users. This expects that the addresses of the Grafana instance and dashboard UIDs will remain static.

15. `https://mattermost.com`
16. `https://slack.com`
17. `https://github.com/prometheus-operator/kube-prometheus`

- Label **severity** – The accepted values are `critical`, `warning`, `error` and `info`. As of this thesis, custom alerts utilize only `critical` severity in case something is broken and needs to be fixed as soon as possible and `warning` severity in case something is likely to fail in the near future.

All alert names should be in upper camel case. Alerts should be generalized; e.g. alerts will be automatically evaluated on a new instance of the same service regardless of the instance-specific label values. While `_url` annotations are required, they will be added step by step in future to already existing rules.

3.4 Logging

The implementation of logging consists of two Git repositories with Kustomization, one for agents and the other for the aggregator. Both components are implemented from scratch and do not reuse any preexisting automation or resource definitions. Aggregator should be deployed in the Managed Kubernetes cluster. The only communication channel needed between these components is the 6000 TCP port used by the Vector.dev forwarding source and sink, respectively. Both aggregator and agents use the same version of Vector.dev, `0.33.1`.

3.4.1 Aggregator

The aggregator is deployed as a single replica *StatefulSet* with configuration stored in one *ConfigMap* and the credentials being injected from a *Secret* as environment variables using the same mechanic as MinIO Kustomization. Vector.dev in the aggregator is configured to automatically reload any *ConfigMap* changes, preventing the need for downtime during most configuration changes. The configuration in the aggregator repository is divided into four YAML files in `files/configuration` directory, allowing easy testing in a local environment outside of the cluster. The aggregator is provided with two TLS certificates. One uses a custom certification authority and is managed by Cert-manager with the design pattern which is described in section 3.2. The other certificate is created externally and manually added to `files/certificates` before deployment of the aggregator. The current configuration only uses the externally created one. The other one is planned for future use cases.

All ingest ports are covered by a *NetworkPolicy*, which allows only necessary source addresses. The aggregator currently has only one external source called *forward*. There are two further sources, both of which only capture internal information about the aggregator, one for metrics and one for logs. The pipeline for the internal metrics is simple as they are only served on an HTTP endpoint, which scraped the Central Prometheus instance.

The forward source is followed by a router stage, which splits the log stream into topic-based branches. Each branch contains one of the topics and has exactly one remap stage, which does additional topic-specific parsing and adds the `.@topic` attribute. The branch with `kube.minio` topic, has an additional deduplication stage which removes any duplicate audit messages sent by different MinIO *Pods*. The `kube.pods` has an additional filtering stage which filters out any logs messages with empty `.message`

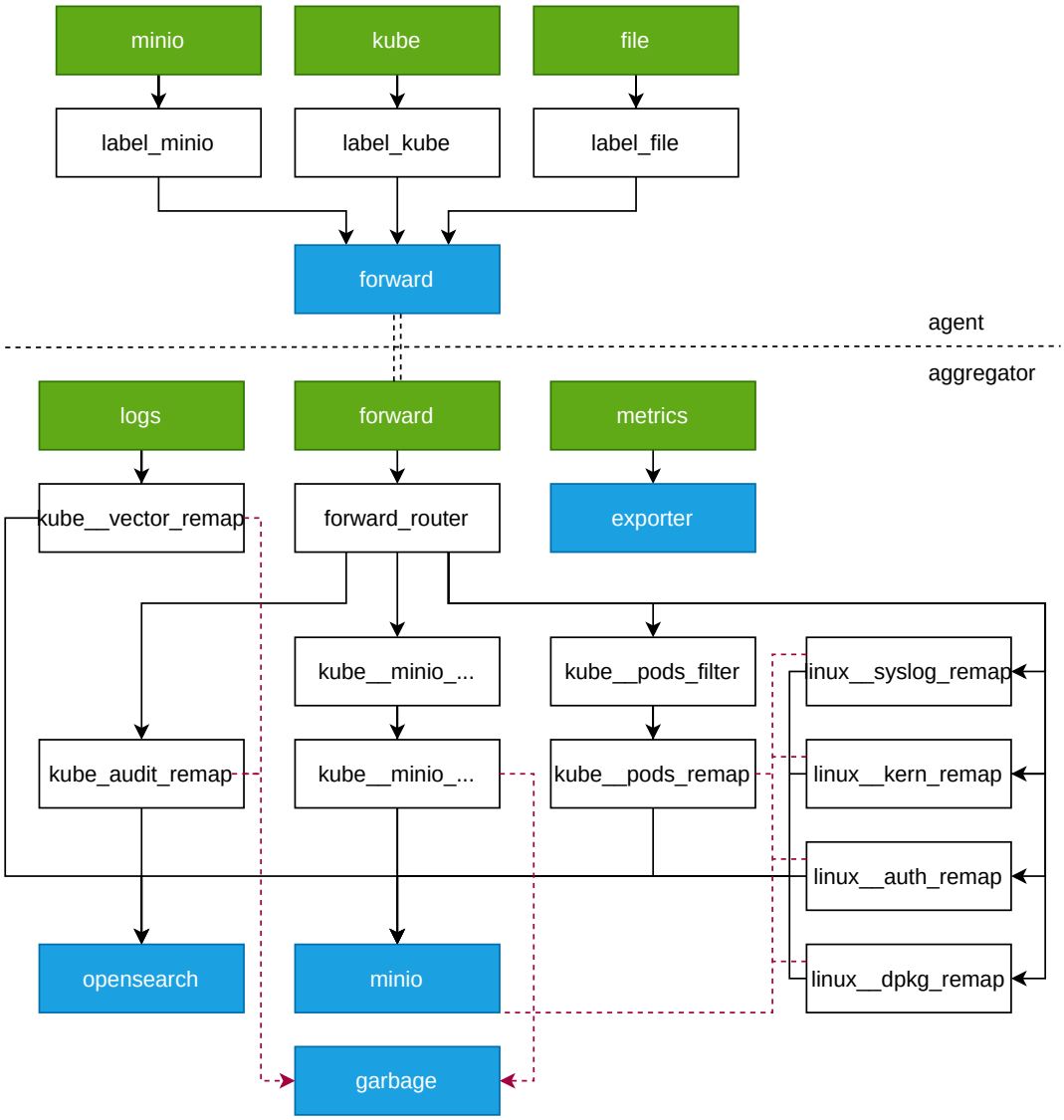

Figure 3.3: Graph of pipeline inside the aggregator and the agents

attribute. The aforementioned internal logs of the aggregator are transformed into the `kube.vector` topic.

Since the aggregator is going to route other unrelated events in the future, the aforementioned topics are prefixed either by `kube` or `linux` keyword. These prefixes signify if the source of the logs is running inside or outside of Kubernetes *Pods*. All received log messages are normalized in the following manner: the existence of `.@topic` attribute is asserted and the `.timestamp` attribute is set to the UTC time of arrival of the message to the relevant remap stage on the aggregator. The schema of each topic contains the following other main attributes.

- `kube.pods` – This topic consists of a `.message` attribute which contains the actual text outputted by the *Pod*, a `.stream` attribute which signifies whether the

message was written to `stdout` or `stderr` and `.kubernetes` attribute, which is a nested object with the metadata retrieved by Vector.dev[18].

- `kube.audit` – As stated in the section 3.1.2 the Kubernetes audit log is being collected as well, the message structure is described in detail in the documentation of Kubernetes API[19]. The auditing policy configured by the Role RKE2 ensures that only modification of resources is logged to prevent an excessive amount of messages.
- `kube.vector` – This topic contains the aggregator's log messages. It is routed separately because the aggregator is placed in Managed Kubernetes, which does not allow simple collection of logging output of all *Pods*.
- `kube.minio` – This topic contains the audit log messages from the MinIO cluster. The schema of the message is determined by MinIO. The official documentation does not provide extensive documentation on these messages but provides several examples[20], which can be used as a reference for further parsing and visualisations.
- `linux.syslog` – The topic contains the typical text attributes of a syslog message such as `appname`, `hostname`, `procid` and `message`.
- `linux.auth` – The topic attributes are similar to the syslog topic. It is routed and stored separately for improved visibility of access to the servers. This is desirable, especially considering that access to the host operating system of Kubernetes nodes is typically rare.
- `linux.kern` – The messages contain the information usually obtained via the `dmesg` command. This information is placed in the `message`. The only other attribute is `host` containing the hostname of the node which produced the message.
- `linux.dpkg` – The messages structure is the same in other `linux.*` topics, but the `.message` contains information about installed and removed packages.

The aggregator has three sinks *minio*, *opensearch* and *garbage*. The *minio* sink is aimed at another instance of object storage with a similar feature set but lower performance. The objects are compressed by ZSTD[21], partitioned by time and topic e.g. `year-2023/month-07/day-07/kube.pods...json.zstd`. The bucket used for logs has an object life-cycle management policy, which automatically deletes the objects 30 days after their creation. The *opensearch* sink ingests the same topics as MinIO; the exact configuration of the OpenSearch instance has already been introduced in subsection 2.1.4. The third sink is called garbage and uses the same low-performance object storage instance; the difference is that it manually overrides `.@topic` value to *garbage*. This is done because this sink ingests only messages from `*_remap.dropped`, which could not be successfully parsed by transformation stages. All sinks have disk buffering enabled,

18. `https://vector.dev/docs/reference/configuration/sources/kubernetes_logs`
19. `https://kubernetes.io/docs/reference/config-api/apiserver-audit.v1`
20. `https://min.io/docs/minio/windows/operations/monitoring/minio-logging.html`
21. `https://github.com/facebook/zstd`

with a relatively large maximum buffer size of 2 TiB per sink. This allows the aggregator to survive several days of low-performance object storage being unavailable without data loss. When the buffers overflow, the newest messages arriving in the stage with overflowing buffers are dropped to prevent cascading failure of the entire logging pipeline.

In order to keep the configuration consistent in the future, several rules were established. Firstly, as shown on Figure 3.3, all stages in the Vector pipeline should be named according to the topic they generate and the type of the stage, e.g. topic called `kube.pods` is parsed in the stage called `kube__pods_remap`, the dots in the topic name are replaced with a double underscore. The port number should always be the same on the *Service* and the *Pod*; further mapping using the *.spec.ports*[]*.targetPort* on *Services* is not desirable.

3.4.2 Agents

The configuration of the agents is significantly simpler than the configuration of the aggregator, and it is placed in a single YAML file containing three sources. The first is used for the collection of selected logs stored in files in `/var/log` on the host. The second ingests logs generated by the Kubernetes *Pods* and adds the metadata retrieved from Kubernetes API. The third exposes a webhook, which is used by MinIO to send audit log messages. Each source has a corresponding transform, which only adds a temporary `.tag` attribute containing the type of the source. This attribute allows me to send the logs via the same sink forward to the aggregator and split them into topics there. This configuration is visualized in the Figure 3.3 above the dotted line.

Since the agents use a volume with log files from the host operating system, a *Pod* with an agent must be running on each node. Therefore, the *StatefulSet* of the aggregator is here replaced with a *DaemonSet*. Other resources the agents require are RBAC, which is used to retrieve the metadata for the log messages generated by the *Pods*. A *Service* is used for the webhook, and a *NetworkPolicy* restricts access to the webhook endpoint. All aforementioned resources are contained in a Kustomization repository.

4 Evaluation and Operation

The section describes the procedures which were used to evaluate the performance of the object storage and the processes needed for the continuous maintenance and updates of the software components, as well as the troubleshooting of the most common operational problems encountered during testing.

4.1 Benchmarks

Three categories of performance metrics need to be measured to be able to express the overall performance of the object storage. The first is the network throughput.

The network performance measurement was done via Iperf[1]. All tests were conducted with the four parallel threads because a single thread proved to be unable to saturate the 100 Gbps link. The *Pod* to *Pod* measurement taken from inside the *Pods* of a *DaemonSet* and used the `10.42.xx.xx` addresses of the *Pods* as targets. The results are shown in the Table 4.1 and Table 4.2. The first table shows the network throughput from and to each node host operating system. The second table shows the network throughput between *Pods* from a *DaemonSet* running on each node. This dual test is necessary because, as previously stated, the networking overlay can degrade the expected network characteristics.

Table 4.1: Node to node network throughput (in Gbps)

From / To	panda-1	panda-2	panda-3	panda-4
panda-1	95.6	97.2	96.6	97.4
panda-2	95.4	96.3	95.7	95.8
panda-3	98.4	95.2	95.0	95.1
panda-4	96.8	94.9	96.1	96.2

Table 4.2: Pod to pod network throughput (in Gbps)

From / To	panda-1	panda-2	panda-3	panda-4
panda-1	92.3	91.1	91.7	93.1
panda-2	93.1	91.9	93.4	91.1
panda-3	92.6	95.5	93.2	90.2
panda-4	92.2	92.9	92.1	94.0

Table 4.3: Drive performance of one node (in GBps)

Total Read	Total Write
14.0	26.0

1. Iperf, not Iperf3 was chosen because of the lack of multi-thread support.

The second metric is the disk performance of each node. For this purpose, MinIO developer team provides a tool called Dperf[2]. This tool resolves the obvious question of which characteristics of the drives should be measured because it is developed specifically for MinIO and the MinIO blog[3] contains several reference results. The results presented in Table 4.3 come only from six drives on one node, as all nodes are identical.

The third and last metric is the performance of the object storage itself. As stated in the subsection 1.4.2, Warp is the standard benchmark for S3 object storage performance. It has the same benefit as Dperf since the aforementioned blog also contains Warp results from various MinIO environments.

The Warp benchmark was executed from three distributed Warp clients on three different client servers connected to the switch via one 100 Gbps optical fibre each. The internal endpoint of the S3 API was used in order to prevent network bottle-neck between the Warp clients and the MinIO cluster nodes. It is important to note that all benchmarks were done on an empty S3 storage cluster without any additional workloads running in said cluster. The exact command is shown in Listing 10. The version Warp used is `0.6.9`.

```
warp get \
  --tls \
  --analyze.v \
  --warp-client warp-{0...3}:7761 \
  --host jumbopanda-{1...4}-bootstrap.csirt.muni.cz
```

Listing 10: The Warp benchmark command

The object size was left on the default 10 MiB; other runs of the benchmarks were conducted, e.g. with 100 MiB, but the performance remained similar. From the verbose output of the benchmark several statistics were deemed the most important and shown in the Table 4.4, Table 4.5 and Table 4.6, please note that partial results in the tables for each node come from the same round of benchmark, executed in parallel.

Table 4.4: Average GET throughput results from Warp

	Average MiB/s	**Average obj/s**
jumbopanda-1	2827.78	282.78
jumbopanda-2	2758.96	275.90
jumbopanda-3	2609.93	260.99
jumbopanda-4	2641.49	264.15
sum	10841.87	1084.19

To summarize, the achieved benchmark results were in the expected range and more importantly, the goal of being able to saturate 100 Gbps link during the reads was achieved.

2. `https://github.com/minio/dperf`
3. `https://blog.min.io`

Table 4.5: Average GET TTFB results from Warp

	Average ms
jumbopanda-1	27
jumbopanda-2	27
jumbopanda-3	28
jumbopanda-4	28
avg	28

Table 4.6: Average PUT throughput results from Warp

	Average MiB/s	**Average obj/s**
jumbopanda-1	92.3	112.45
jumbopanda-2	93.1	110.28
jumbopanda-3	92.6	109.15
jumbopanda-4	92.2	108.11
sum	**4577.91**	**457.79**

4.2 Updates

There are several levels of components which need to be regularly patched and updated. Firstly, the packages on the host operating system. Secondly, the Kubernetes itself and thirdly, the applications running inside the cluster itself, e.g. MinIO.

The common aspect of all three levels is that the person responsible for the maintenance of the cluster should never install anything without reading the release notes of the software in question. Moreover, special attention should be paid to the supported Kubernetes versions of each operator since the major versions of Kubernetes, e.g. `1.25` often bring API changes, making the older operator version incompatible. The last common aspect is that, after the provisioning process described in chapter 3, all versions of the important components are stored in Git repositories, which should be updated synchronously with the live cluster environment.

Please note that the following sections are guidelines, not a definitive algorithm, which should be upheld at all costs.

4.2.1 Operating System

The updates of the host operating system are partially solved by Ubuntu's unattended upgrade functionality, which is present and sufficiently configured in a default installation of Ubuntu server 22.04.3 LTS. However, the kernel updates still require the restart of the machine in question. Furthermore, certain packages, especially those whose configuration was adjusted during the provisioning process, require manual intervention and are not subjected to unattended-upgrades. This manual intervention can be summarized in the commands shown in the Listing 11.

The node is only cordoned off because the primary and currently only serious workload present in the cluster is MinIO, whose *Pods*, as stated previously, cannot be rescheduled on other nodes because they are hard-bound to the drives accessible only

```
kubectl cordon jumbopanda-1

ssh csirt@jumbopanda-1.csirt.muni.cz \
    apt-get update && \
    apt-get upgrade -y && \
    reboot

kubectl uncordon jumbopanda-1
```

Listing 11: Command required for OS upgrade of one node

from one node. The draining of the node would be suitable if the cluster hosted other workloads, which do not require exactly one *Pod* on each node.

The upgrade of the cluster to the next Ubuntu LTS release is neither documented nor planned because version 22.04 of Ubuntu is supported until May of 2027.

4.2.2 Kubernetes

The RKE2 distribution allows the updates of itself to be executed via the System Upgrade Controller, which was along with its *CRDs* introduced in the subsection 1.7.1. While our MinIO cluster is able to tolerate the unavailability of up to two nodes, the upgrade *Plan* has configured upgrade concurrency of one. This is done because updates can occasionally fail, leaving the cluster damaged. With this configuration, a failed upgrade can only destroy one node, leaving the cluster in a non-critical state, as it can still lose one more node without data loss occurring. The upgrade process itself is simple: execute `kubectl edit -n system-upgrade plan rke2-server-plan` and change the `.spec.version` attribute to the desired value, the operator will cordon and uncordon the nodes automatically.

The most important rule is that the RKE2 should only be updated after all deployed operators and other components support the given version. Due to the faster development pace of the infrastructure in the environment of this thesis, the rule for RKE2 version selection is the following: find the latest stable version and use the next one, e.g. if the official stable is `1.24.x`, than `1.25.y` should be used, of course only after all previous assertions are verified.

4.2.3 Applications

The general recommendation for updates of operators and other applications in the cluster is to subscribe to the relevant Github[4] projects release notification, check the Kubernetes version compatibility and update as soon as possible.

For MinIO, a different approach was found to be the most sustainable. MinIO project differentiates between so-called `feature` and `bugfix` releases; the release frequency is approximately one release per two weeks. After a new `feature` release is published, the latest available `bugfix` should be used for the update to achieve the highest degree

4. `https://github.com`

of stability. This approach leads to an approximately monthly cycle of updates, which can be easily coordinated with other aforementioned upgrade levels.

4.3 Troubleshooting

This chapter describes the diagnostic options available and solutions to the problems, which were either deemed likely to occur in the future or already happened during the evaluation and subsequent operation of the cluster.

4.3.1 Observability

In order to increase the ability of the administrators to resolve any future issues, the following measures were taken. All potentially destructive operations executed via the Kubernetes API are logged. The same holds for all actions done by users of S3 object storage, including any read operations. Any package additions or removals and user logins on the host operating system are also logged. All aforementioned logs are available in the Managed OpenSearch independently of the availability of the cluster itself. Furthermore, a standard set of Grafana dashboards, along with corresponding alerting and recording rules, is available. However, the initial collection of alerts is not definitive; each new incident should result in at least one additional rule which will detect it. Other than the severity label, the user tagging capability of the chat software, which receives the notifications from the Alertmanager, can be used to signal additional urgency to the administrators. This is used in the serious problem indicators:

- **NodeOffline** – This alert is triggered when the five-minute average of MinIO node in the offline state is over zero for more than ten minutes.
- **DriversOffline** – The same the `NodesOffline` alert, but with MinIO drives instead of nodes.
- **500ErrorRate** – This alert is based on observations made while evaluating the MinIO cluster. For a period of one month, the only occurrences of the non-zero rate of HTTP `5XX` response codes were during unplanned downtime and other incidents. The alert detects a non-zero `5XX` error rate that lasts longer than one minute.
- **DriveErrosTimeout** – This alert detects a non-zero error rate which lasts more than a minute. According to the MinIO documentation, the errors included are timeouts, permission denied and I/O errors.
- **DriveErrorsAvailability** – It was also suggested by MinIO documentation and is similar to the rule above, but should detect timeout errors since the server start.

4.3.2 Availability

While both the Kubernetes and the MinIO clusters can tolerate the downtime of two nodes without service disruption, this is not entirely reflective of the end-user experience. The users must consider that their requests can fail and retry them in case of failure.

These retries should not use cached resolved domains because it is likely that a particular node causes the failure of requests. The main purpose of the object storage cluster will be to serve as a data warehouse [15], where both the read and write components of the current version of this platform implement the above-mentioned retries.

4.3.3 Recovery

The following paragraph describes the recovery scenarios, including the theoretical worst-case scenario in which everything fails, but twelve drives survive. According to the MinIO documentation [13] it does not matter which particular drives survived, as long as there are at least twelve of them (in our particular cluster configuration).

The new *Pods* must always be scheduled on the same nodes where old *Pods* were, e.g. `minio-0` was on `jumbopanda-1`, so it must be scheduled there again. MinIO instances are aware of their identities, meaning that the drives contain the domain under which they belong. The identity is stored in the `.minio.sys/pool.bin` in the root of every second drive on the node.

Therefore, the proper restart procedure is to restart one *Pod* and wait until it starts and becomes ready, only then continue with the next *Pod*, in case the *Pod* does not start correctly, immediately stop the process. During the recovery procedure, the *Pods* can be scheduled to the correct node by using `kubectl scale` to increase replicas one by one and `kubectl uncordon` to allow scheduling to the nodes in the correct order.

5 Conclusion

In conclusion, I believe that all outlined goals were achieved, with some areas allowing for additional improvement beyond the original assignment.

The RKE2 cluster is deployed according to the best practices included in the documentation confirming all resiliency and performance requirements given by MinIO, with one notable exception of the pod security audit policy is not enforced and only emits warnings to the administrators. This could be improved but would likely require custom-made container images for some of the operators and MinIO itself. In terms of the firewall configuration, adding another policy to the internal interfaces facing the top of the rack switch is possible but would not result in any increase in security, as the allowed communication required by the RKE2 gives any imaginable access to the attacker.

The object storage as it is is stable and easily maintainable. All observability data is collected in an independent location and is accessible to administrators and users. The cluster allows for a complete failure of one node without any loss of functionality. The upgrade process of MinIO is seamless and tested. Additional functionality, such as site-to-site replication and different storage tiers, could be implemented with additional hardware. The same goes for performance improvement, which would require either more nodes and drives or faster than 100 Gbps networking.

The main area which could be developed further is observability. While all available data is collected, the presentation of it offers many further possibilities which were not properly analyzed. These custom dashboards and alerts could be tailored for the specific use cases of each user and could be templated and provisioned automatically in the way as the pre-made Grafana dashboard for Kubernetes components.

Additionally, the current logs could be extended by additional visibility tools such as OSQuery[1], which combine with a suitable set of queries, e.g. osquery-defense-kit[2] could track and account for any changes in the hosts operating system. The output of any such additions could be easily added to the current logging pipeline.

1. `https://osquery.io`
2. `https://github.com/chainguard-dev/osquery-defense-kit`

Bibliography

1. HOCHSTEIN, Lorin; MOSER, Rene. *Ansible: Up and Running: Automating configuration management and deployment the easy way*. " O'Reilly Media, Inc.", 2017.
2. MOLÍK, Ondřej. *Provisioning of Monitoring Infrastructure for Traffic Flow Collection*. Brno, 2022. Available also from: https://is.muni.cz/th/vbs0t/.
3. BERNSTEIN, David. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*. 2014, vol. 1, no. 3, pp. 81–84. Available from DOI: 10.1109/MCC.2014.51.
4. PAHL, Claus; BROGI, Antonio; SOLDANI, Jacopo; JAMSHIDI, Pooyan. Cloud Container Technologies: A State-of-the-Art Review. *IEEE Transactions on Cloud Computing*. 2019, vol. 7, no. 3, pp. 677–692. Available from DOI: 10.1109/TCC.2017.2702586.
5. BURNS, Brendan; BEDA, Joe; HIGHTOWER, Kelsey; EVENSON, Lachlan. *Kubernetes: up and running*. " O'Reilly Media, Inc.", 2022.
6. LUKSA, Marko. *Kubernetes in action*. Simon and Schuster, 2017.
7. *Recommended Labels* [online]. Brno: The Kubernetes Authors, 2023 [visited on 2023-10-01]. Available from: https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels.
8. JOHNSTON, Craig; JOHNSTON, Craig. DevOps Infrastructure. *Advanced Platform Development with Kubernetes: Enabling Data Management, the Internet of Things, Blockchain, and Machine Learning*. 2020, pp. 33–69.
9. KAPOČIUS, Narūnas. Overview of kubernetes cni plugins performance. *Mokslas–Lietuvos ateitis/Science–Future of Lithuania*. 2020, vol. 12.
10. TOVARŇÁK, Daniel; RAČEK, Matúš; VELAN, Petr. Cloud Native Data Platform for Network Telemetry and Analytics. In: MÜGE SAYIT, Stuart Clayman (ed.). *17th International Conference on Network and Service Management* [elektronická verze "online"]. Izmir, Turkey (Virtual): IFIP Open Digital Library, IEEE Xplore, 2021, pp. 394–396. ISBN 978-3-903176-36-2. Available from DOI: http://dx.doi.org/10.23919/CNSM52442.2021.9615568.
11. *Erasure Coding* [online]. Brno: MinIO, Inc., 2023 [visited on 2023-10-23]. Available from: https://min.io/docs/minio/linux/operations/concepts/erasure-coding.html.
12. *Object Storage Erasure Coding vs. Block Storage RAID* [online]. Brno: Matt Sarrel, 2022 [visited on 2023-11-01]. Available from: https://blog.min.io/erasure-coding-vs-raid/.
13. *Erasure Coding 101* [online]. Brno: Matt Sarrel, 2022 [visited on 2023-11-05]. Available from: https://blog.min.io/erasure-coding/.
14. *How to Benchmark MinIO with WARP and Speedtest* [online]. Brno: Matt Sarrel, Cesar Celis Hernandez, 2023 [visited on 2023-10-05]. Available from: https://blog.min.io/how-to-benchmark-minio-warp-speedtest.

15. TOVARŇÁK, Daniel. *Normalization of Unstructured Log Data into Streams of Structured Event Objects*. Brno, 2018. Available also from: https://is.muni.cz/th/rjfzq/. Disertační práce. Masarykova univerzita, Fakulta informatiky. Supervised by Tomáš PITNER.

16. *Configuration* [online]. Brno: Prometheus Authors, 2023 [visited on 2023-10-13]. Available from: https://prometheus.io/docs/alerting/0.26/alertmanager.

17. *Deployment topologies | Vector documentation* [online]. Brno: Datadog, Inc., 2023 [visited on 2023-10-16]. Available from: https://vector.dev/docs/setup/deployment/topologies/#centralized.

18. *Bulk - OpenSearch documentation* [online]. Brno: Opensearch Contributors, 2023 [visited on 2023-10-13]. Available from: https://opensearch.org/docs/latest/api-reference/document-apis/bulk/.

19. *Reference Architecture | Build a High-Performance Object Storage-as-a-Service Platform with Minio** [online]. Brno: Intel Corporation, 2019 [visited on 2023-10-23]. Available from: https://min.io/resources/docs/CPG-MinIO-reference-architecture.pdf.

20. SAYFAN, Gigi. *Mastering kubernetes*. Packt Publishing Ltd, 2017.

21. *Hardware Checklist* [online]. Brno: MinIO, Inc., 2023 [visited on 2023-10-22]. Available from: https://min.io/docs/minio/linux/operations/checklists/hardware.html.

22. *High Availability - Prometheus Operator* [online]. Brno, 2023 [visited on 2023-10-30]. Available from: https://prometheus-operator.dev/docs/operator/api/.

23. ZVONÍK, Tomáš. *Logging and Monitoring for Kubernetes Infrastructure* [*online*]. 2022 [cit. 2023-10-05]. Available also from: https://is.muni.cz/th/db88p/. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno. SUPERVISOR : Lukáš Hejtmánek.

24. *Architecting your Deployment* [online]. Brno: Datadog, Inc., 2023 [visited on 2023-11-01]. Available from: https://vector.dev/docs/setup/going-to-prod/architecting.

25. *Configuration* [online]. Brno: The MetalLB Contributors, 2023 [visited on 2023-11-11]. Available from: https://metallb.org/configuration/.

A Structure of Attachment

The attachment is compiled from several Git repositories, but their histories are removed for security reasons. The credentials and sensitive information present in the files are redacted and replaced with the `CENSORED` keyword. The attachment is structured according to the types of artifacts.

- `playbook` – Ansible playbook deploys bare-metal RKE2 cluster
- `roles`
  - `core` – Ansible role for basic host OS hardening and configuration
  - `rke2` – Ansible role, which provisions an empty RKE2 cluster
- `kustomizations`
  - `agents` – deploys Vector.dev log collection agents
  - `aggregator` – deploys Vector.dev central log aggregator
  - `minio` – deploys MinIO object storage cluster
  - `calico-firewall` – configures the Calico firewall
  - `baseline` – deploys basic operators and controllers
  - `monitoring` – deploys the central monitoring stack
- `other`
  - Grafana Vector.dev dashboard

B Attachments

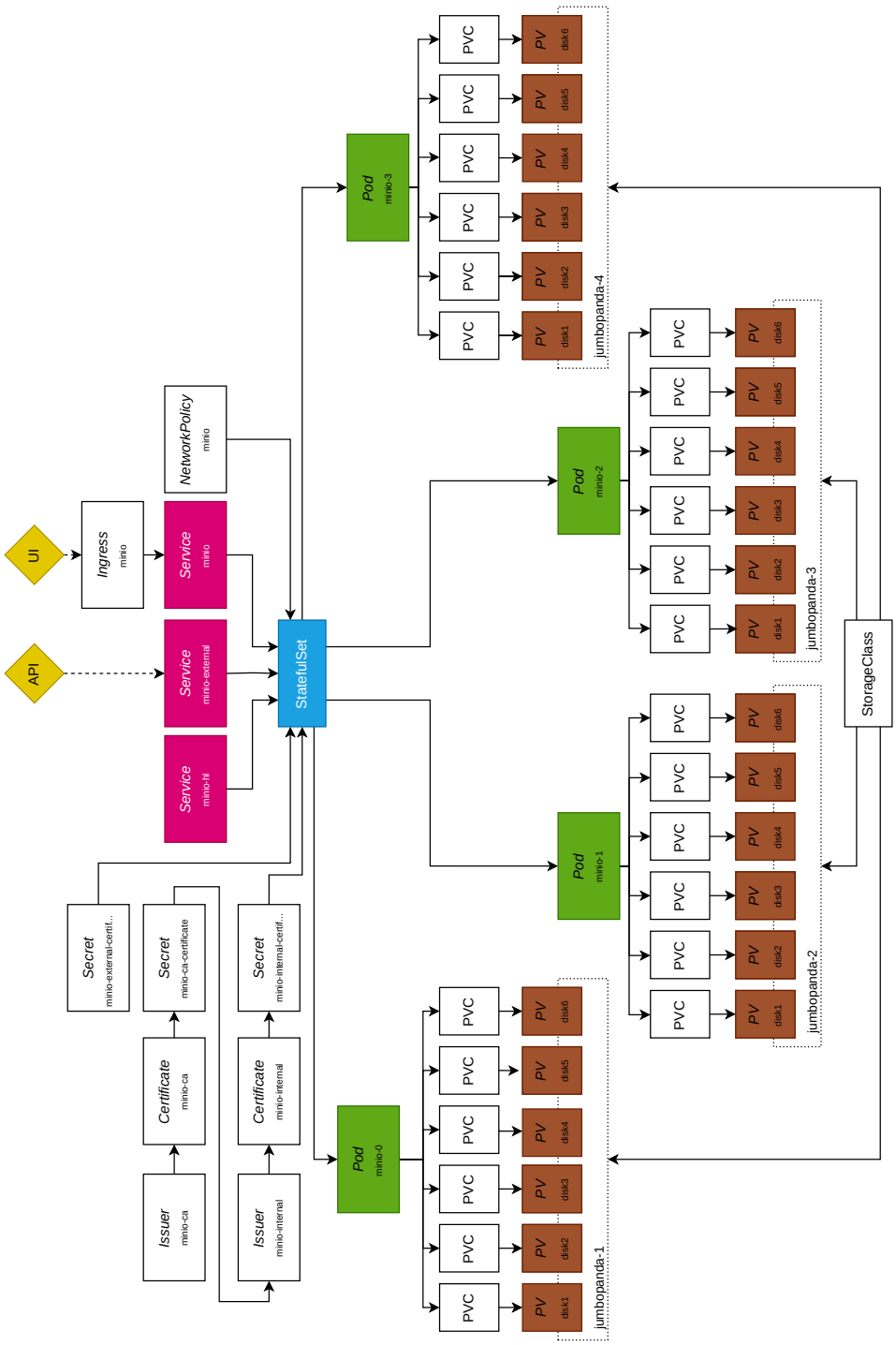

Figure B.1: Kubernetes resources related to MinIO cluster

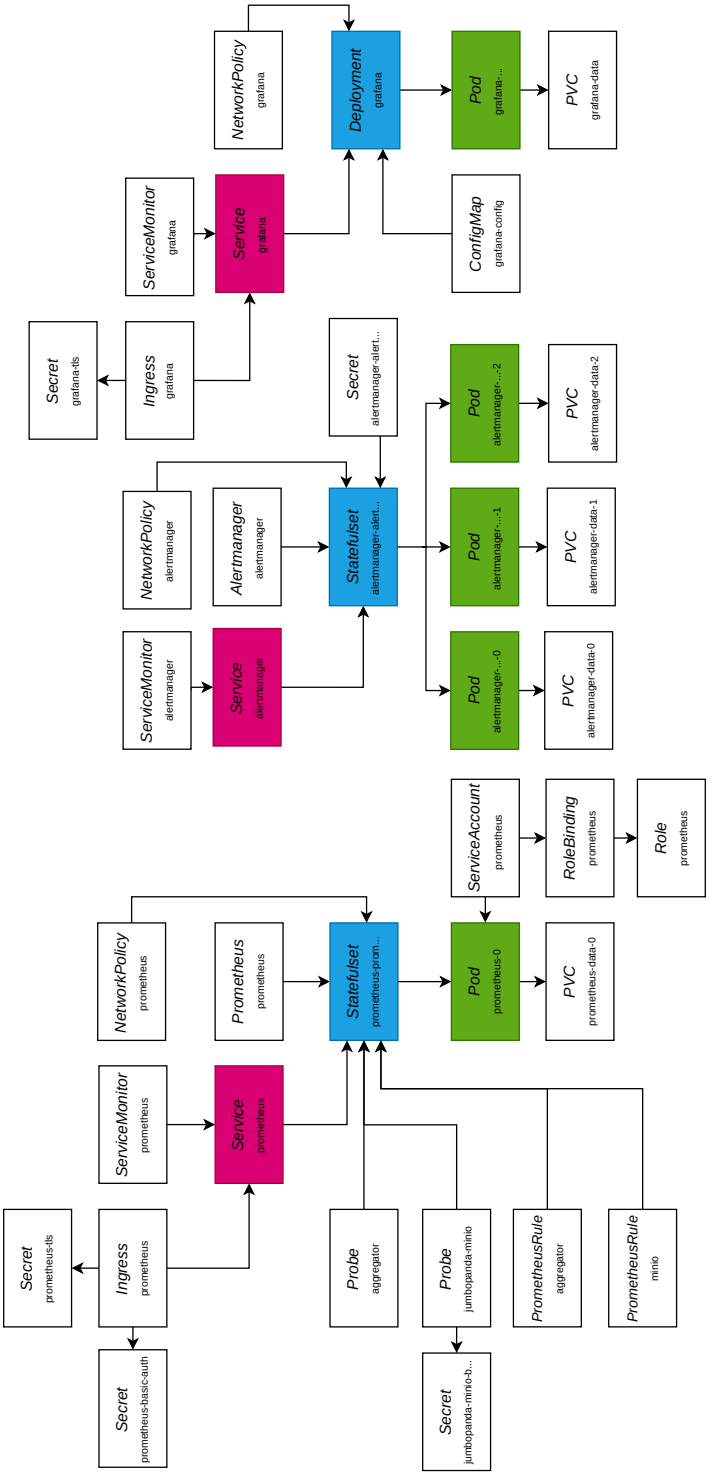

Figure B.2: Kubernetes resources related to the central monitoring component

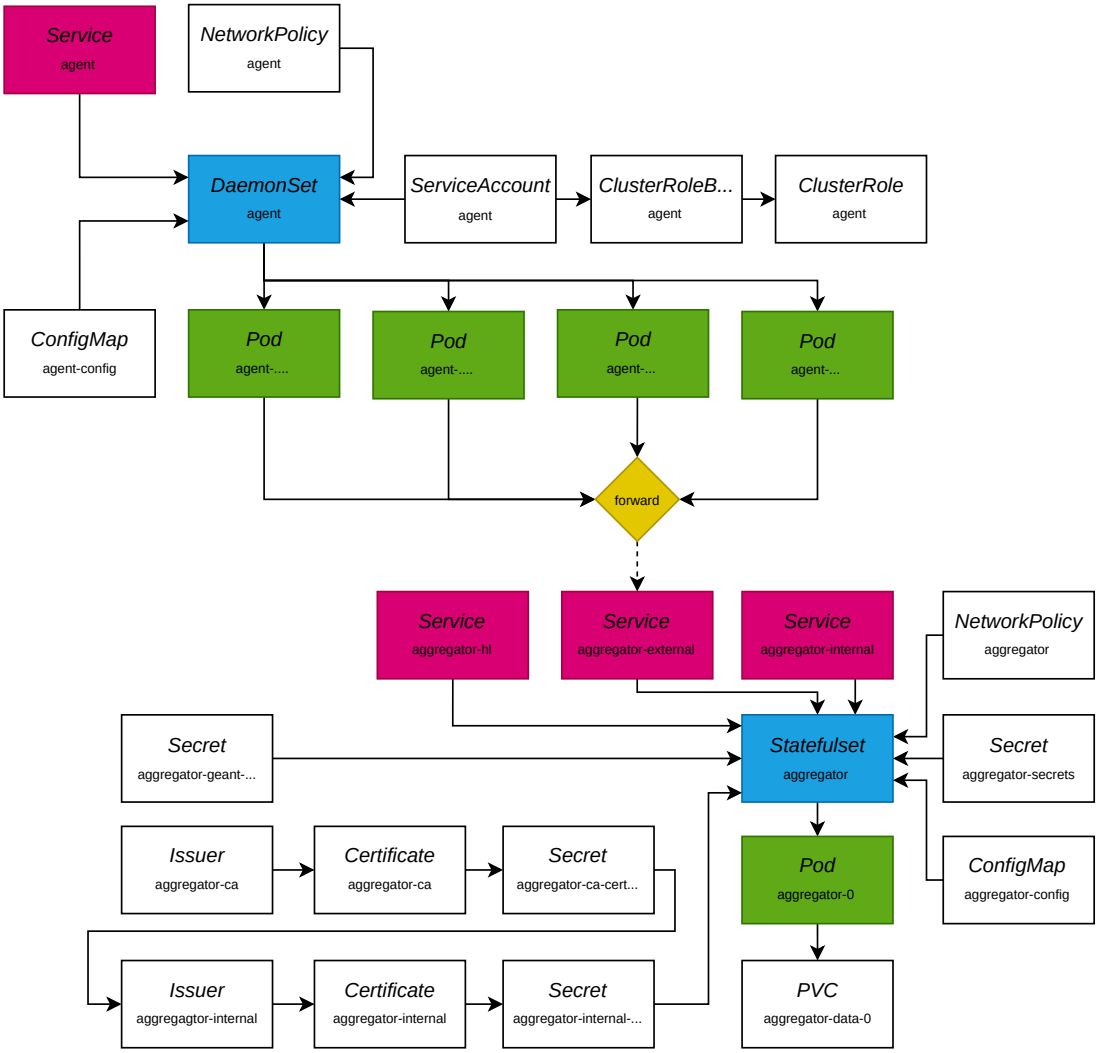

Figure B.3: Kubernetes resources related to both logging components

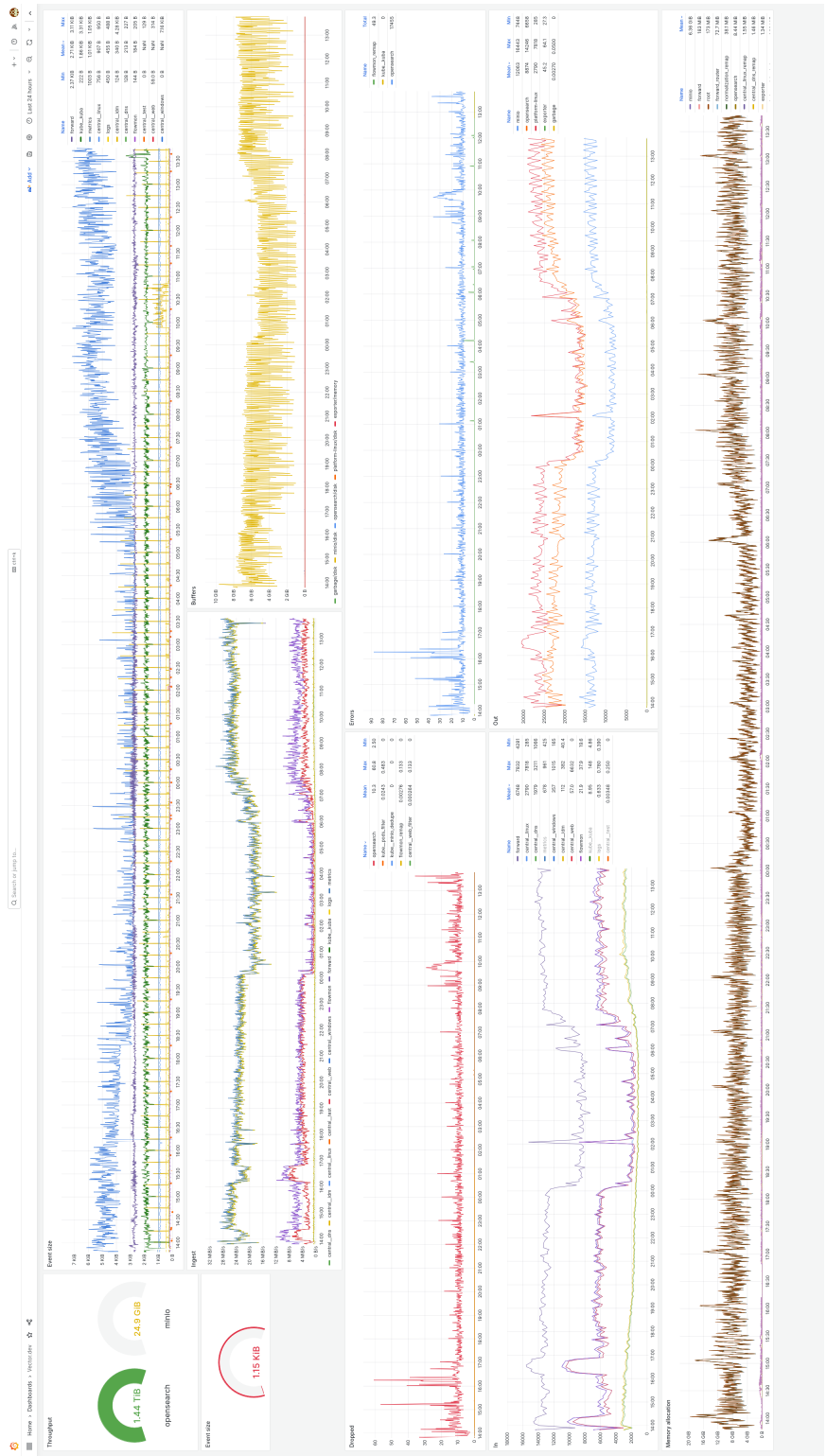

Figure B.4: Grafana dashboard with aggregator metrics