POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

An abstract model of cloud-native networks for security enforcement and remediation

Supervisors:

Prof. Cataldo Basile

Dott. Franceso Settanni

Candidate:

Giuseppe Lisena

Academic Year 2024/2025
Torino

Abstract

In recent years, Kubernetes has established itself as a dominant platform for container orchestration, becoming an integral component of numerous cloud infrastructures. A significant portion of Kubernetes' popularity stems from its ecosystem of external components, including operators, network plugins, and various other tools. The proliferation of these components, each tailored to specific use cases, has resulted in a diverse and often overlapping landscape of solutions.

This thesis undertakes a comprehensive analysis of several prominent Kubernetes network plugins, operators, and tools, encompassing popular solutions such as Flannel, Calico, Cilium, Network Service Mesh, and Kube-router. The analysis delves into their respective features, performance characteristics, and security implications, enabling a comparative evaluation.

Furthermore, this thesis introduces an abstract model that encapsulates the diverse resource types within Kubernetes. Designed to encompass a broad spectrum of Kubernetes resources, the model has been validated against real-world deployments, including Google Boutique, IBM Java microservices, and practical applications powered by Kubernetes, Cilium, and KubeArmor. This structured representation offers a foundation for automated processing by software tools.

Ringraziamenti

Chi mi conosce sa bene che non sono una persona che dà peso alle formalità e che, anzi, cerca di evitarle quando può. Nonostante ciò, ritengo doveroso ringraziare le persone che hanno reso più piacevole questo percorso e altre, come i miei genitori, che lo hanno reso innanzitutto possibile. È quindi a loro, Mario e Maria, che va il mio sentito ringraziamento per aver sempre supportato le mie scelte e non avermi mai fatto mancare nulla, ricordandomi sempre il giusto criterio in ogni cosa. Ringrazio anche mio fratello, Davide, che, nonostante due anni non esattamente in discesa, è stato un ottimo confidente e mi ha sopportato meglio di chiunque altro, come solo un fratello minore sa fare.

Ringrazio Roberto, Nico e Marco, i compagni di ventura che hanno intrapreso questo percorso a Torino insieme a me: Roberto, compagno inestimabile di studi, la cui collaborazione è stata imprescindibile per il raggiungimento di questo traguardo; Nico, che è stato il mio coinquilino durante questo percorso, aiutandomi quando ne avevo bisogno, oltre che collega universitario, sempre pronto a mettere tutto in discussione; Marco, amico fedele e disponibile, che è sempre stato in grado di strapparmi una risata anche nei momenti meno sereni.

Ringrazio Lele, che nonostante la lontananza non ha mai mancato di far sentire la sua presenza, mettendosi sempre a disposizione per discutere delle questioni più disparate e consigliandomi sempre bene quando ne ho avuto bisogno.

Ringrazio Antonio, amico di una vita, sempre disponibile a sostenere interminabili telefonate con il sottoscritto, ricordando aneddoti e condividendo novità, dubbi e i più disparati pensieri.

Ringrazio anche Maty e Martina, amiche di (quasi) nuova data che hanno aiutato a rendere più piacevoli e leggeri questi ultimi mesi.

Un sentito ringraziamento va al Prof. Basile e al Dott. Settanni, per l'opportunità datami e per la loro infinita disponibilità. La loro guida e i loro consigli sono stati essenziali nella stesura di questo elaborato e nel completamento del lavoro su cui quest'ultimo si basa.

Table of Contents

List of Figures

Acronyms

CNI
: Container Network Interface

K8s
: Kubernetes

NAT
: Network Address Translation

DNS
: Domain Name System

CIDR
: Classless Inter-Domain Routing

VM
: Virtual Machine

RABC
: Role-Base Access Control

NFV
: Network functions virtualization

NSM
: Network Service Mesh

CRD
: Custom Resource Definition

eBPF

extended Berkeley Packet Filter

ACL

Access Control List

OVS

Open vSwitch

CNF

Cloud-Native Network Function

Chapter 1

Introduction

The 21st century witnessed the rise of cloud computing as the go-to solution for managing complex IT infrastructures, offering scalability, flexibility, and cost-effective deployment models for a broad range of applications. As businesses increasingly adopted cloud technologies to streamline operations and accelerate innovation, the demand for efficient and reliable management of applications and infrastructure grew exponentially. Cloud computing, initially seen as a tool to optimize storage and computing capabilities, soon became the cornerstone of digital transformation across industries. The ability to dynamically scale resources on demand, combined with pay-per-use pricing models, reshaped how businesses manage IT workloads, resulting in both operational efficiencies and new business opportunities. However, as cloud technologies evolved, the complexity of managing these infrastructures also grew. Organizations found themselves navigating intricate environments with various tools and platforms designed to manage diverse workloads, networking configurations, and security requirements. The increasing complexity of cloud infrastructures and the need to maintain a high level of availability, security, and performance pushed the boundaries of traditional IT management practices. Consequently, businesses began to look for solutions that could simplify this management while maintaining the agility and scalability offered by the cloud.

1.1 Simplifying Cloud Management

Efficiently managing large-scale cloud IT infrastructures became a significantly complex task to accomplish, mainly due to the inadequacy of traditional network and application management paradigms. The intricacy of cloud-native applications, the dynamic nature of virtualized resources, and the rapid pace of innovation made it evident that new approaches were necessary. To overcome these limitations, Software-Defined Networking (SDN) and Kubernetes have emerged as revolutionary technologies that address the core challenges faced by modern cloud infrastructures. SDN transformed the way networks are managed by decoupling the network control plane from the data plane, offering centralized control that enables programmatic control over network behavior. This paradigm allows for greater

automation and flexibility in configuring network topologies, traffic routing, and resource allocation. Kubernetes, on the other hand, emerged as the leading solution for orchestrating containerized applications, abstracting away the underlying infrastructure complexities. It automates tasks such as deployment, scaling, and management of containerized workloads, making it easier for developers and operations teams to focus on application logic rather than infrastructure management. Both SDN and Kubernetes have laid the foundation for simplifying cloud management, offering powerful tools that automate and abstract the complexities of managing networks and applications. As a result, organizations can now deploy highly resilient, scalable, and secure infrastructures with a degree of automation that was previously unthinkable.

1.2 The Growing Complexity of Kubernetes

As Kubernetes became the de facto standard for container orchestration in the cloud computing space, it gained widespread adoption across industries. Its modular architecture and extensibility enabled the Kubernetes ecosystem to grow rapidly, with a vast array of tools, plugins, and operators designed to enhance its capabilities. From networking solutions like Calico and Flannel to storage frameworks such as Rook and Ceph, Kubernetes' ecosystem has expanded to include a wide variety of components addressing diverse use cases. These components have brought immense value to Kubernetes users by providing enhanced networking, storage, security, and application management features, enabling businesses to run complex, distributed applications with ease. However, the rapid expansion of Kubernetes-related tools and plugins has introduced a new set of challenges. As the ecosystem grows, so does the risk of overlap between the functionalities of various tools, which can lead to confusion, inefficiencies, and even compatibility issues. Furthermore, the need for seamless integration of these tools, particularly in heterogeneous environments with different cloud providers and on-premise infrastructure, adds another layer of complexity. With such a broad selection of tools available, organizations often face difficulties in selecting the right components that align with their specific needs while maintaining a high degree of interoperability. This growing complexity in the Kubernetes landscape has highlighted the need for better strategies to manage and optimize the usage of these tools. Understanding how different Kubernetes frameworks interact, what features they offer, and where overlaps or conflicts may arise has become crucial for teams looking to harness Kubernetes' full potential without introducing unnecessary complexity into their workflows.

1.3 Goal of the Thesis

This work aims to conduct an in-depth analysis of the main network plugins, operators, and tools that populate the landscape of Kubernetes frameworks. By examining and comparing the features offered by each of these components, this thesis seeks to identify their strengths, weaknesses, and use cases, ultimately providing insights into how organizations can leverage these technologies to streamline the management of their Kubernetes environments. The

next step is to build an abstract model that can effectively represent Kubernetes resources in a way that is intuitive and easily manipulable by both applications and automated systems. This model will aim to simplify the interaction between developers, operations teams, and Kubernetes resources, ensuring greater flexibility and efficiency in the management of cloud-native applications. Once developed, the model will be validated against real-world use cases, providing practical evidence of its effectiveness in addressing the challenges posed by the growing complexity of Kubernetes and its ecosystem. Through this work, this thesis will contribute to a deeper understanding of Kubernetes management and offer solutions to mitigate the challenges faced by organizations as they navigate this increasingly intricate environment.

1.4 Results Overview

This work presents a comparative analysis of the main Kubernetes extensions, ranging from Container Network Interface to Service Meshes, and Security tools, highlighting the differences between the different solutions. An abstract model for various Kubernetes resources is also presented to simplify the management of clusters and help the process of migrating from one solution to another. For this scope, an automated Python framework has been developed, which can analyze the features of Kubernetes manifests and, using the abstract mode, translate them.

1.5 Outline

The thesis is organized as follows. All the notions necessary to be able to understand the work more technically will be presented in Chapter 2. Chapter 3 details the work done to achieve the previously presented objectives. Within Chapter 4, the results of the work are presented, along with the validation of such work. Finally, in Chapter 5, the conclusions and possible future work related to this project are presented.

Chapter 2

Background

This chapter presents the principal concepts and technologies essential for understanding the background of the developments undertaken in this thesis. The core ideas outline various approaches within the field of networking and their evolution over time. It explores key advancements that have shaped modern network architectures and management, providing a foundation for the subsequent discussions.

2.1 Traditional networking

In the past, applications were predominantly deployed on physical servers, marking the early days of enterprise infrastructure management. This approach, while simple in concept, came with several limitations that organizations had to contend with. At its core, physical server deployment was a one-to-one relationship: each application or service typically had its dedicated hardware. While this allowed for some isolation between applications, it was not sufficient to enforce strict resource boundaries, particularly when multiple applications shared the same physical server. The lack of proper resource allocation and isolation mechanisms meant that poorly behaved applications could consume an excessive amount of resources, leading to performance degradation, system crashes, or downtime. This became a significant problem as applications grew in complexity, and their demands on infrastructure increased.

In response to these challenges, organizations began to dedicate separate physical servers to individual applications. By isolating applications onto their physical machines, administrators could ensure that the resource consumption of one application wouldn't impact the others. This approach, while somewhat effective in providing isolation and stability, introduced a new set of problems. The biggest among these was the underutilization of resources because physical servers often had excess capacity. This inefficiency became increasingly costly as businesses scaled their operations.

Furthermore, the practice of managing several physical servers proved to be both complex and costly. Each server required its own set of maintenance routines, including regular updates, hardware checks, and troubleshooting. The need for physical space to

house servers, as well as the associated energy consumption and cooling requirements, added to the overhead. This not only inflated operational costs but also created a logistical challenge for IT departments that had to keep track of multiple machines, handle the physical installation and replacement of hardware, and ensure that the right resources were allocated to the right application at all times.

As organizations grew and the number of applications multiplied, IT teams faced significant challenges in scaling their infrastructure to meet the demands of modern applications. In addition, maintaining the security and integrity of each server was a daunting task. Each server was a potential point of failure, and if one server went down or was compromised, it could jeopardize the availability of crucial applications and services.

The inefficiencies and challenges associated with physical server deployment ultimately paved the way for more advanced technologies, such as virtualization and cloud computing. These innovations allowed for better resource allocation, improved scalability, and greater flexibility in deploying and managing applications.

2.2 Virtualization

While virtualization technology can be traced back to the 1960s, it wasn't widely adopted until the early 2000s. The technologies that enabled virtualization were developed decades ago to give multiple users simultaneous access to computers that performed batch processing [1].

Virtualization uses software to create an abstraction layer over computer hardware, enabling the division of a single computer's hardware components into multiple virtual machines (VMs). Each VM runs its operating system (OS) and behaves like an independent computer, even though it is running on just a portion of the actual underlying computer hardware [2].

Server virtualization enables the consolidation of multiple VMs onto a single physical server, maximizing resource efficiency (Figure 2.1). This approach allows for the optimal use of computing capacity, storage, and networking resources.

Virtualization also simplifies IT management by facilitating the automation of tasks such as provisioning, deployment, and configuration. By defining and deploying VMs as software templates, administrators can streamline processes and reduce manual intervention, minimizing errors and accelerating service delivery. Additionally, virtualization enables the implementation of granular security policies, tailored to the specific needs of each VM.

With virtualization, it is possible to reduce downtime through the deployment of redundant VMs. In the event of a failure, these redundant VMs can seamlessly take over, ensuring business continuity. Moreover, virtualization accelerates the provisioning of new applications and services, as it eliminates the need for physical hardware procurement and configuration.

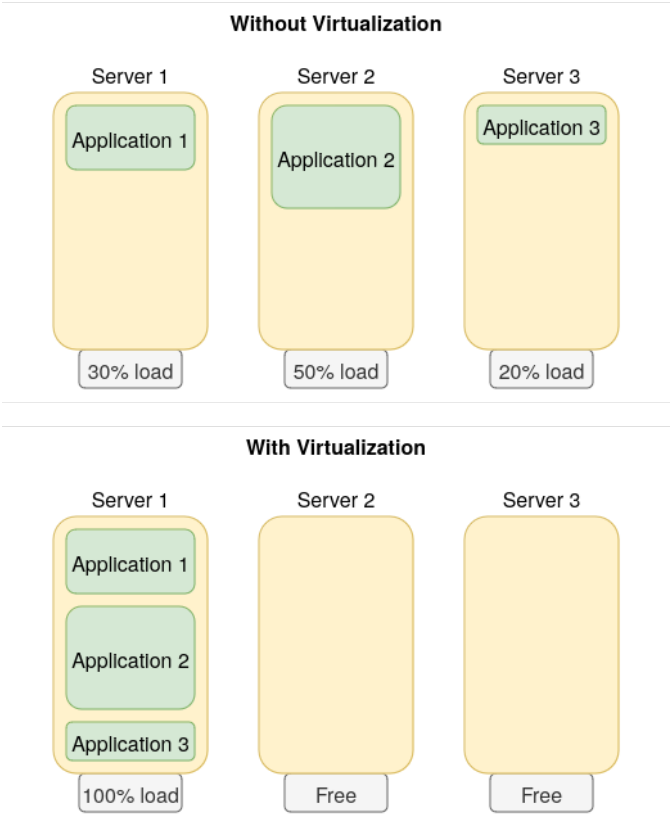

Figure 2.1: No virtualization vs Virtualization

2.3 Network Functions Virtualization

Network Function Virtualization (NFV) is a way to virtualize network services, such as routers, firewalls, and load balancers, that have traditionally been run on proprietary hardware. These services are packaged as virtual machines on commodity hardware, which allows service providers to run their networks on standard servers instead of proprietary ones. It is one of the primary components of a telco cloud, which is reshaping the telecommunications industry [3].

A network functions virtualization standard was first proposed at the OpenFlow World Congress in 2012 by the European Telecommunications Standards Institute [4].

NFV enables service providers to execute network functions on standardized hardware, eliminating the need for dedicated equipment. By virtualizing network functions, multiple instances can be executed on a single server, leading to reduced hardware requirements and, consequently, lower power consumption, space utilization, and overall operational costs.

Additionally, NFV offers service providers the flexibility to deploy Virtual Network Functions (VNFs) across diverse servers or dynamically relocate them in response to

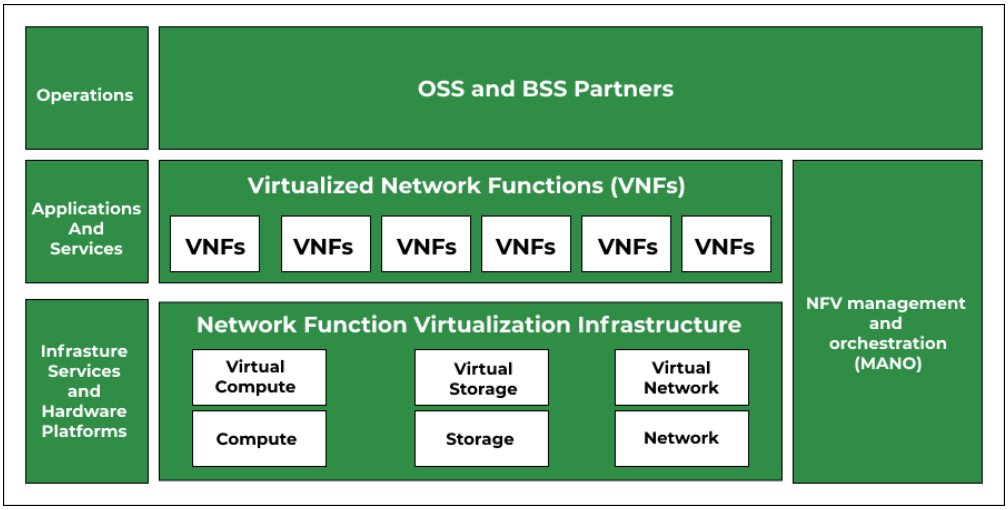

Figure 2.2: Network Functions Virtualization overview [4]

fluctuating demand. This adaptability accelerates the delivery of services and applications. For instance, when a customer requests a new network function, a VM can be rapidly provisioned to fulfill the requirement. Subsequently, if the function becomes redundant, the VM can be decommissioned. This approach also facilitates low-risk testing of potential new services.

2.4 Software-Defined Networking

Software-defined networking (SDN) is a category of technologies enable managing a network via software. SDN technology enables IT administrators to configure their networks using a software application. SDN software is interoperable, meaning it should be able to work with any router or switch, no matter which vendor made it [5].

The key advantage of SDN is that it leverages the separation of the control plane from the data plane. In networking terminology, a "plane" refers to an abstract conceptual layer where networking processes occur. The **control plane** encompasses the network control functions, such as routing and forwarding decisions, while the **data plane** handles the actual transmission of data packets. Traditional network architectures tightly couple the control and data planes within individual network devices like routers and switches. This configuration necessitates device-by-device configuration, limiting scalability and flexibility. SDN, on the other hand, decouples the control plane from the data plane and the underlying hardware. This separation enables centralized control and management of the network.

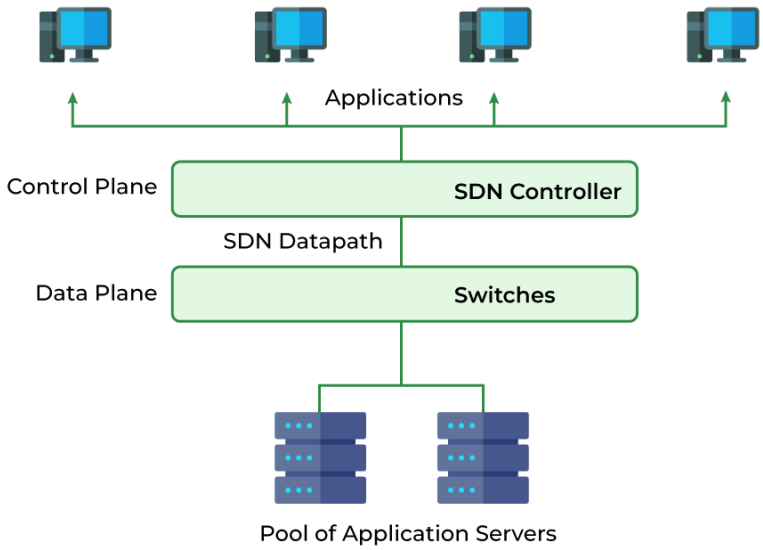

Figure 2.3: SDN architecture [6]

2.5 Microservices

Complex application deployments can be subdivided into multiple simpler components, also called *micro-services*, each of them implementing a specific function of the application. This approach also simplifies the development, testing, and scalability of the application. Microservices can be implemented and deployed independently of the others. Each element could be made in a different language, eliminating implementation-specific limitations (Figure 2.4).

However, the microservices-based approach has some drawbacks. For this type of architecture to be effective, it is necessary to have efficient management of requests, coordinating the different modules to avoid problems. Furthermore, having a copious amount of different modules could become problematic for monitoring and configuring services, as well as debugging and logging. Finally, automating the entire architecture could become a challenging task.

The implementation of Virtualized Network Functions based on Virtual Machines (VMs) faces several inefficiencies, as discussed before. These challenges can degrade service quality, especially when large-scale resources are needed. To address these limitations, the shift to Cloud-Native Network Functions (CNFs) is essential. CNFs, designed for cloud environments, utilize microservices architecture and containerization, making orchestration easier with tools like Kubernetes.

CNFs offer better scalability and efficiency than VNFs [7], benefiting from cloud computing's infrastructure to manage service scalability and provide cost-effective resource usage. Microservices, isolated in containers, ensure greater flexibility and resource management, allowing independent management and updates without impacting other services.

Additionally, containers require fewer resources than VMs, allowing for the consolidation of multiple microservices into a single container and optimizing resources and performance.

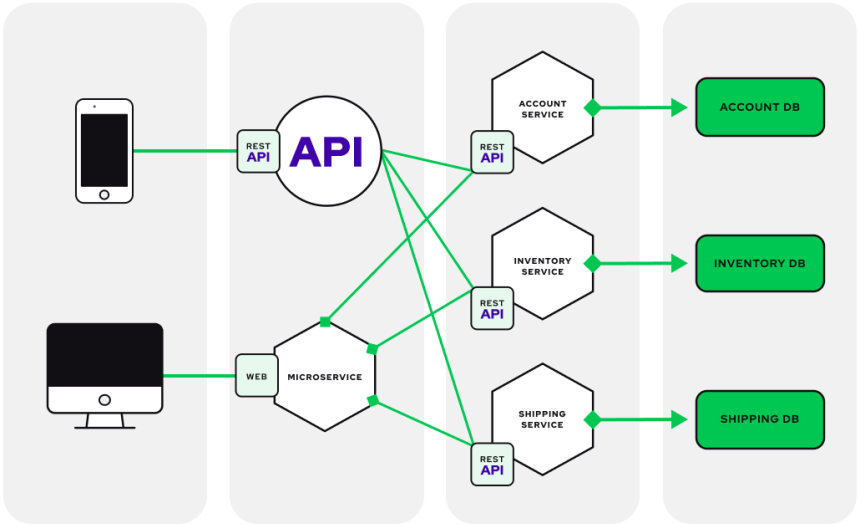

Figure 2.4: Microservices Application [8]

2.6 Containerization

Containerization is the packaging of software code with just the operating system libraries and dependencies required to run the code to create a single lightweight executable, a *container*, that runs consistently on any infrastructure [9].

Historically, developers faced challenges when transferring applications between different systems, often encountering bugs and errors due to discrepancies in configurations and dependencies. Containerization addresses this issue by bundling applications with their required components, creating portable units that can be executed seamlessly across various platforms and clouds.

Containerization offers a lightweight and efficient approach to packaging and deploying applications. By sharing the host operating system and isolating applications within containers, it reduces overhead, improves performance, and enhances security. Containers are highly portable, allowing them to run consistently across different platforms and cloud environments. This portability, combined with the ability to package monolithic and microservice-based applications, accelerates development and deployment cycles [10].

One of the key advantages of containers is isolation. This feature can be achieved by employing two technologies that are part of the Linux kernel:

- **Cgroups**, that limits and allocates resources, such as CPU, memory, disk I/O, and network bandwidth, among process groups, preventing any single container from

consuming all available system resources.

- **Namespaces**, that partitions kernel resources so that processes within a namespace have their isolated instance of global resources, creating the illusion for containers that they are running on their own independent system. Typical namespaces are PID, to isolate process IDs, NET, to isolate network interfaces, and MNT, to isolate filesystem mount points.

2.6.1 Docker

Docker is a software platform facilitating the rapid construction, testing, and deployment of applications. It packages software into standardized units termed containers, each encompassing all requisite components for execution, including libraries, system tools, code, and runtime environments. By leveraging Docker, developers can swiftly deploy and scale applications across diverse environments, assured of their consistent operation.

Docker is based on a client/server architecture, with many components that interact with each other:

- **Docker host**, a physical or virtual machine running a Docker-compatible operating system such as Linux.
- **Docker Engine**, Docker client-server architecture, comprising a Docker daemon, Docker API, and Docker CLI.
- **Dockerd**, Docker daemon service that creates and manages Docker images, using the commands from the client.
- **Docker client**, provides the CLI that accesses the Docker API.
- **Docker objects**, components that facilitate the packaging and distribution of applications. These components include, but are not limited to, images, containers, networks, volumes, and plugins.
- **Docker containers**, the instances of Docker images. Docker images serve as immutable, read-only templates, whereas containers are dynamic, executable instances derived from these images. Users can interact with containers, and administrators can modify their configurations and runtime environments through Docker commands.
- **Docker images**, images contain executable application source code and all the tools, libraries, and dependencies that the application code needs to run as a container.
- **Docker registry**, registries where it is possible to upload or download images in such a way that centralized repositories are available. Generally, they are public, but can also be configured for private use.

Docker images are composed of a series of layers, each representing a specific version of the image. When a developer modifies an image, a new top layer is added, becoming the latest version. Previous layers are retained for potential rollbacks or reuse in other projects.

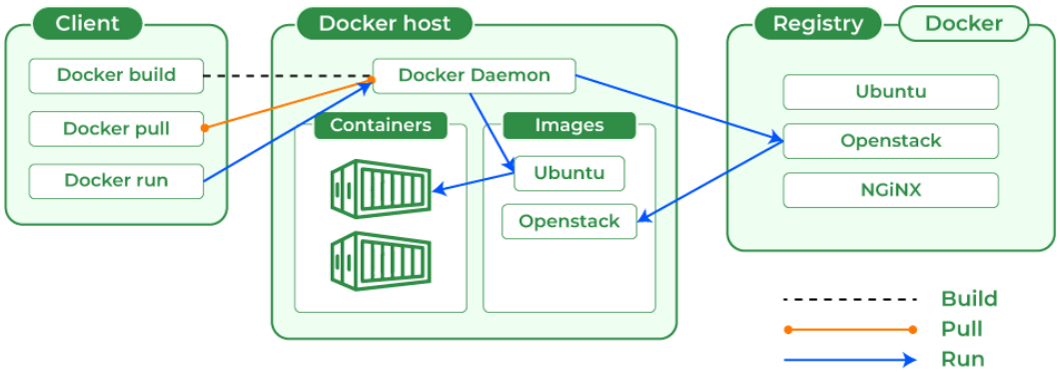

Figure 2.5: Docker architecture [11]

Upon container creation from a Docker image, an additional layer, known as the container layer, is generated. Modifications made to the container, such as file additions or deletions, are stored within this layer, which persists only for the duration of the container's runtime. This layered approach to image creation enhances efficiency by enabling multiple live container instances to operate from a single base image. This shared base image allows for a common software stack across all instances.

Docker Compose

Docker Compose serves as an effective instrument for the definition and management of multi-container applications. Enhance development and deployment processes, thus facilitating a more seamless and efficient experience. Through the use of Compose, it is possible to exercise comprehensive control over an entire application stack by managing services, networks, and volumes using a straightforward YAML configuration file. The execution of a singular command suffices to instantiate and initiate all services delineated within the configuration. Compose works in all environments: production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application [12].

Docker image

Containers are instantiated from images, which function as immutable templates encompassing all required instructions for the assembly of a Docker container. The construction of an image is facilitated by a Dockerfile (Listing 2.1), a text document meticulously configured to encompass all necessary commands for image creation. These instructions are executed sequentially as written, therefore, careful consideration of their order is imperative. The most important instructions available are:

- `FROM`, specifies the base image to build upon; must be the first instruction in a

Dockerfile.

- **RUN**, executes commands in a new layer and commits the results; used to install packages, run builds, etc. Can be formatted in two ways:

```
RUN <command to execute>
RUN ["executable-file", "params"]
```

- **ADD**, copies files from the source to the container's filesystem with special features like auto-extraction of archives and URL support.
- **COPY**, copies files from build context to container; simpler than ADD without extraction or URL features.
- **ENV**, sets environment variables available during both build and runtime in the container.
- **ARG**, defines global variables available only during build time; can also be passed at build time with *–build-arg*.
- **CMD**, provides default commands and arguments for executing the container and can be overridden at runtime.
- **ENTRYPOINT**, configures the container to run as an executable; harder to override than CMD and is often used together with CMD.
- **WORKDIR**, sets a working directory for subsequent instructions. It creates the directory if it does not exist.

Listing 2.1: Dockerfile example

```
# Base image
FROM ubuntu:22.04

# Build argument
ARG VERSION=1.0

# Environment variables
ENV APP_HOME=/app

# Run commands
RUN apt-get update && \
    apt-get install -y python3 && \
    mkdir -p ${APP_HOME}

# Copy local files
COPY ./app.py ${APP_HOME}/

# Add files
```

```
ADD ./data.tar.gz ${APP_HOME}/

# Set working directory
WORKDIR ${APP_HOME}

# Define the executable
ENTRYPOINT ["python3"]

# Default arguments
CMD ["app.py", "--version", "${VERSION}"]
```

2.7 Kubernetes

This section provides an overview of Kubernetes, the foundational technology underpinning the work presented in this thesis. We will dive into its core concepts and components.

2.7.1 History

To address the challenges of managing a rapidly growing cloud infrastructure and delivering a robust platform, Google developed Borg, a pioneering container management system. Inspired by the concept of collective consciousness, Borg was designed to orchestrate containerized workloads across large-scale clusters. In 2013, Google introduced Omega, its second-generation container management system. Omega further developed the Borg ecosystem [13].

Docker's popularity, with its focus on individual container creation and deployment, inspired Google engineers to explore the concept further. However, limitations arose in managing large-scale deployments with individual nodes. The need for automation across multiple containers and machines became apparent, leading Google to develop its next-generation container management system: Kubernetes.

Coinciding with the release of Kubernetes 1.0 in 2015, Google donated Kubernetes to the Cloud Native Computing Foundation [13]. Since then, Kubernetes has emerged as the most widely adopted container orchestration platform.

2.7.2 Control Loop

Kubernetes leverages a declarative model, allowing developers to define their desired application state rather than specifying the precise steps to achieve it. This means developers focus on "what" they want (e.g., a running application with specific features) and leave the "how" and "where" to Kubernetes. This abstraction simplifies application management and ensures continuous service availability.

From a developer's perspective, the primary goal is a consistently running service that's resilient and scalable. Kubernetes achieves this through internal mechanisms like ReplicaSets and Control Loops.

ReplicaSets act as intermediaries, ensuring the desired number of pod replicas are always running. They use selectors to identify pods, define the desired replica count, and provide a pod template for creation. While ReplicaSets manage pod replication, Deployments are preferred for their higher-level, declarative update capabilities.

Underpinning these mechanisms is the *Control Loop* (Figure 2.6), a continuous process that maintains the desired state. It operates in three steps:

- **Observe**: Monitor the current state of the desired object.
- **Analyze**: Compare the current state to the desired state and identify discrepancies.
- **Act**: Take corrective actions to align the current state with the desired state.

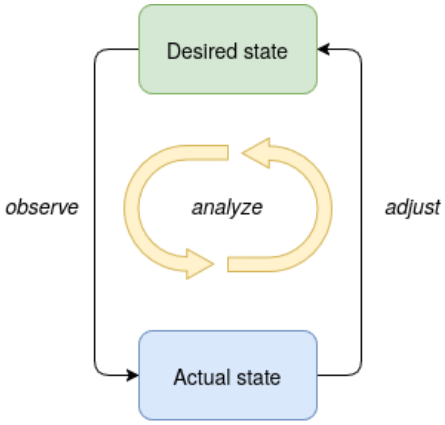

Figure 2.6: Kubernetes Control Loop

2.7.3 Architecture

A Kubernetes cluster is organized in a single master node, referred to as the control plane. The control plane handles one or more worker nodes.

Worker nodes contain all the necessary components to run the assigned workloads assigned by the master node.

Control Plane Components

The control plane maintains the desired state of the cluster as defined by the user. It ensures that the cluster operates as intended, assigning workloads to appropriate worker nodes based on different parameters, reacting to different events in the cluster, including failures, and storing the configuration data, state, and metadata of the cluster.

API Server. The API Server acts as the front-end for the Kubernetes control plane. It exposes the Kubernetes API, which is used to interact with the cluster. The API Server handles RESTful requests, processes them, and updates the cluster state.

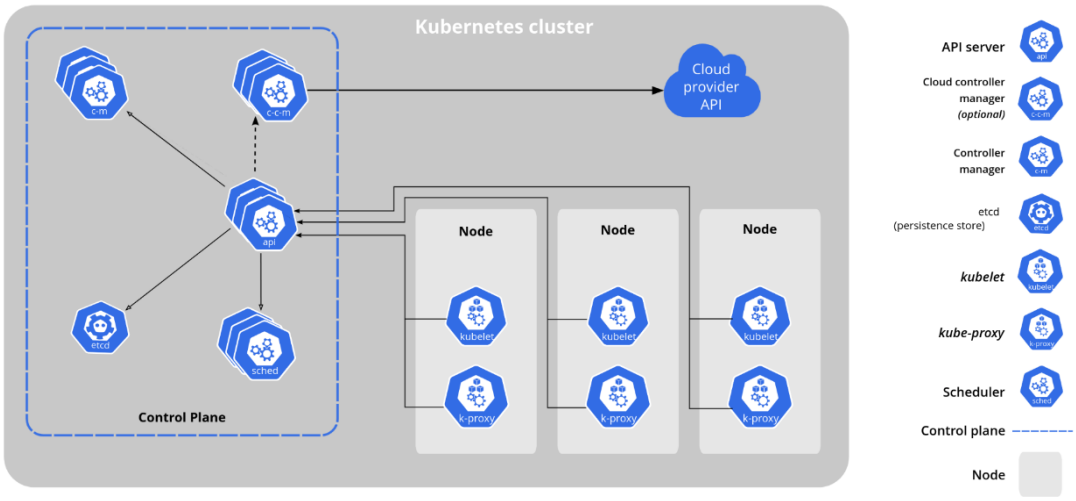

Figure 2.7: The components of a Kubernetes cluster [14]

Etcd. Etcd is a distributed key-value store that stores the cluster configuration, state, and metadata. It ensures data consistency and reliability across the cluster by providing a highly available and fault-tolerant storage solution. The API Server interacts with Etcd to store and access information about the cluster.

Scheduler. The Scheduler is responsible for assigning pods to worker nodes based on different factors. It takes into account factors such as CPU and memory requirements, affinity rules, and policy constraints.

Kube Controller Manager. The Kube Controller Manager runs various controller processes to shift the current state of the cluster towards the defined desired state. Each controller is responsible for a specific aspect of the cluster's operation, but they are all compiled into a single binary and run in a single process for the sake of reducing complexity. Examples of controllers are *Node controller*, responsible for noticing and responding when nodes go down, and *Job controller*, which watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.

Cloud Controller Manager. The Cloud Controller Manager integrates Kubernetes with cloud provider-specific services and resources, allowing the Kubernetes control plane to interact with the underlying cloud infrastructure. The Cloud Controller Manager runs cloud-specific controller processes that handle the lifecycle of cloud resources, ensuring that they are properly configured and maintained. As with the Kube Controller Manager, these processes are compiled into a single binary.

Node Components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment [14].

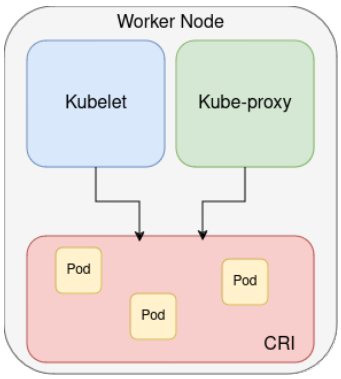

Figure 2.8: The components of a worker node

Kubelet. The Kubelet is an agent that ensures that containers described in PodSpecs are running and in a healthy condition by continuously monitoring their status. The Kubelet communicates with the API Server to receive instructions and report back the status of the node and its pods.

Kube-proxy. Kube-proxy is a network proxy that manages network rules and handles traffic routing for services within the cluster. It maintains the network rules to allow seamless communication between pods and services, regardless of their location within the cluster. Kube-proxy is an optional component since a dedicated network plugin can execute its function.

Container Runtime. The Container Runtime is the software responsible for running containers and managing their lifecycle. Examples of container runtimes are Docker, containerd, and CRI-O. The Container Runtime interfaces with the Kubelet to execute operations as specified in the PodSpecs, ensuring that the containers are running as intended and providing the necessary environment for applications to operate correctly.

Objects

In a Kubernetes cluster persistent entities are used to represent the state of the cluster. These entities can describe different aspects of the cluster, including running applications, defined policies, and available resources. The way these objects are used is through the use of the Kubernetes API calls. A typical Kubernetes object includes:

- *apiVersion*, specifies the Kubernetes API version used by the object.

- *kind*, is the type of the object.
- *metadata*, includes fields to uniquely identify the object, like a UID or a string, and, optionally, a namespace.
- *spec*, describes the characteristics of the object.
- *status*, an optional field that describes the current state of the object.

Below are some of the fundamental objects of Kubernetes.

Pod. A Pod is the smallest deployable unit in Kubernetes, representing a single instance of a running process in the cluster. A pod can contain a single container or a group of them sharing resources.

Service. Services are abstractions employed to expose an application running on a set of pods, handling load balancing among the pods, and service discovery.

The need for the service abstraction is rooted in the way Kubernetes works: pods are automatically generated, restarted, destroyed, and so on, thus making it difficult to keep track of their IP addresses. So Kubernetes assigns each pod an IP address and groups all the pods running the same application under a single DNS name.

Kubernetes offers four types of services, specified via `spec.type`, that allow exposing the application in different ways:

- `ClusterIP`, default option if no type is specified; the service is exposed only within the cluster via a defined IP address.
- `NodePort`, allows the assignment of an IP address and a port to a service, allowing it to be reached from outside the cluster.
- `LoadBalancer`, exposes the service externally employing a load balancer provided by a cloud provider; by defining a LoadBalancer service, ClusterIP and NodePort services are also created and the external traffic is routed to them.
- `ExternalName`, maps the service to a particular DNS name, the `externalName`; when reaching the service, a `CNAME` record containing the defined `externalName` is returned.

Deployment. A Deployment provides declarative updates for *Pod* and `ReplicaSet` resources [14]. With a `Deployment`, a desired state for the cluster is defined. The component responsible for applying changes to the current state to converge toward the desired one is the Deployment Controller. When defining a Deployment, a ReplicaSet is created.

ReplicaSet. A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. The ReplicaSet is defined in the Deployment and automatically managed. A ReplicaSet is linked to its Pods via the Pods' `metadata.ownerReferences` field, which specifies what resource the current object is owned by [14].

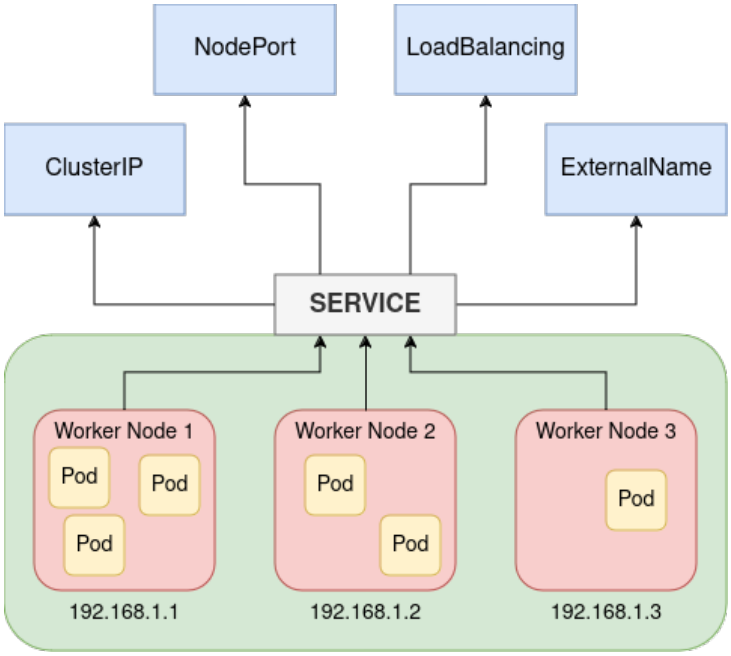

Figure 2.9: Kubernetes Service

Namespace. Kubernetes namespaces allow the isolation of groups of resources within a single cluster. Inside a namespace, the different resources have unique names; these names can be reused in other namespaces. Namespace-based partitioning applies to those resources that are not cluster-wide objects, like Pods or Services. Thus, objects like `Node` and `PersisteVolume` are not namespace-partitionable.

NetworkPolicy. Network policies are employed to specify rules for traffic flow within your cluster, and also between Pods and the outside world, controlling traffic flow at the IP address or port level (OSI layer 3 or 4) [14]. To employ network policies, it is necessary to use a network plugin that supports them. Network policies apply to a connection with a pod on one or both ends and are not relevant to other connections. If allowed, a pod can communicate with other pods, namespaces, or IP blocks. For pod-based and namespace-based policies, a `selector` is used to determine the allowed entities, while for IP blocks, CIDR ranges are used. As further detailed, network policies serve as essential guardrails, explicitly defining allowed and denied connections to restrict traffic between pods (both within and across namespaces) and between pods and external networks. A network policy specification includes a `podSelector` to identify the pods to which the policy will apply, and `policyTypes` to specify whether the policy governs ingress (inbound) or egress (outbound) traffic, or both. Ingress rules dictate the allowed inbound traffic to the target pods, while egress rules specify the permitted outbound traffic from these pods. Each rule incorporates a `NetworkPolicyPeer` to select the entities (pods, namespaces, or IP blocks using CIDR notation) on the other side of the connection, and a

`NetworkPolicyPort` to explicitly define the allowed ports or protocols for communication with the pod. Importantly, network policies are additive in nature; if multiple policies select a particular pod, the effective set of rules is the union of all the ingress and egress rules defined across those policies [15].

2.7.4 Storage

Pods are ephemeral, meaning that the state of the information is not saved once the pod is terminated. Also, sharing files between the same pod among multiple containers can be quite challenging. For the mentioned reasons, Kubernetes provides two storage solutions.

Ephemeral Volumes

Ephemeral volumes have a lifetime equal to that of a pod and are primarily used when applications need additional space, such as when files need to be available for reading configurations useful to the pod. Common types of ephemeral volumes are:

- **`EmptyDir`**, an empty volume is created when a pod is assigned to a node and can then be read and written.
- **`ConfigMap`**, read-only volumes used to make data for configurations available to the pod; data is saved within a `ConfigMap` volume and used by the container inside the pod

Persistent Volumes

Persistent volumes (PVs) represent a portion of the cluster's storage that is made accessible, allowing the volume's lifecycle to remain independent from the lifecycle of the pod. Users can create Persistent Volume Claims (PVCs), which are requests for resources from the persistent volumes, where they specify the amount of storage required and the preferred access mode.

There are two primary methods for provisioning persistent volumes: dynamic and static. In the dynamic approach, when PVs do not meet the specifications in a PVC request, new volumes are automatically created to fulfill the request. This is managed through storage classes. In contrast, the static method involves pre-creating a set of PVs, which can then be allocated to users as needed.

2.7.5 RBAC

Kubernetes employs Role-Based Access Control to govern resource access based on user roles. The Kubernetes API group defines four core objects for managing permissions:

- `Role`, defines permissions within a specific namespace.
- `ClusterRole`, defines permissions across the entire cluster.

- `RoleBinding`, assigns a Role to a user or group within a namespace.
- `ClusterRoleBinding`, assigns a ClusterRole to a user or group across all namespaces.

RBAC employs a simple permission model:

- **Subjects**, the allowed entities (users, groups, service accounts).
- **Resources**, the target resources (pods, deployments, secrets).
- **Verbs**, the access methods (get, list, create, update, delete).

2.7.6 Operators

Another fundamental Kubernetes concept is the *operator*. The reason for introducing this notion is associated with the desire to personalize the management of certain automated processes within the cluster by specifying what should be done when certain events occur. Operators leverage the concept of *Custom Resource Definitions* (CRD), enabling the definition of new types of resources that can then be monitored by the operator itself (Figure 2.10. Custom resources are extensions of the standard Kubernetes API that provide a fine-grained level of control over applications and their components, making it possible to automate tasks beyond what Kubernetes itself offers.

The critical point lies in the coding aspect of the actions that need to be executed within the cluster whenever an event related to the CRD occurs. The operator utilizes high-level configurations, implementing a logic close to the one adopted by control loops.

After creating and defining a new CRD, an object can be generated containing customizable fields. By default, all unrecognized fields are deleted; hence, the field 'x-kubernetes-preserve-unknown-fields: true' must be introduced to preserve unknown fields.

Operators can be differentiated based on their capabilities in managing the life cycle of applications, which can be more or less comprehensive. In this sense, a complete operator should be capable of managing all aspects related to pods, such as scheduling, installing, upgrading, scaling, recovering, and so forth [16]. The five macro levels by which operators can be classified, indicating their maturity level, are:

- **Basic Install**: The operator must be able to install and configure all the necessary workloads to run the custom resources.
- **Seamless Upgrades**: Upgrades to various workloads should be possible without data loss.
- **Full Lifecycle**: This level refers to the operator's ability to back up and restore the state of workloads.
- **Deep Insights**: Monitoring and alert systems are implemented for pod management.
- **Auto Pilot**: Based on well-defined metrics, the operator can automatically scale resources.

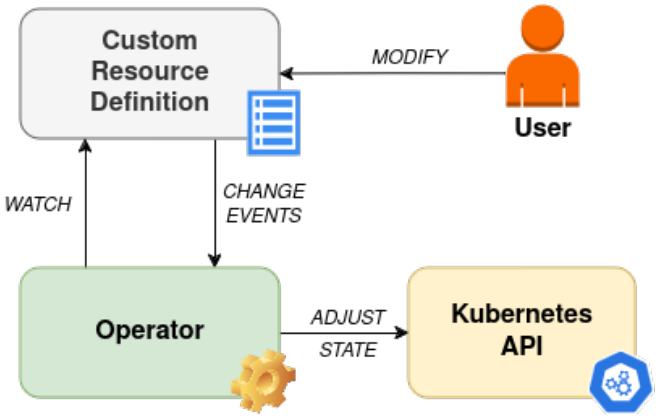

Figure 2.10: Kuberntes Operator workflow

2.7.7 Container Network Interface Plugins

Kubernetes, starting from version 1.3, supports the usage of Container Network Interface plugins to manage the networking needs of a Kubernetes cluster. A CNI plugin is required to implement the Kubernetes network model [14]. Popular Kubernetes network plugins include Kube-Router, Flannel, Calico, and Cilium. A CNI plugin is required to implement the Kubernetes network model.

Container Network Interface. The Container Network Interface (CNI) constitutes a foundational project under the auspices of the Cloud Native Computing Foundation, providing a comprehensive specification and associated libraries for seamless container networking. These resources enable developers to develop plugins capable of dynamically configuring network interfaces within Linux containers. Moreover, the project includes an array of officially supported plugins that offer immediate solutions to common networking requirements. Within the Kubernetes ecosystem, the CNI assumes a pivotal role as the standard interface for network configuration within the cluster. Kubernetes depends on CNI plugins to provide essential networking functionalities for its pods, including the assignment of IP addresses, the establishment of inter-pod communication, and the facilitation of external access to services. Prominent CNI plugins within the Kubernetes ecosystem, such as Calico, Cilium, and Flannel, each present distinctive features and capabilities, extending from basic network connectivity to sophisticated network policy enforcement and security functionalities [17].

Kubernetes network model. The Kubernetes network model defines how containers and pods communicate within a Kubernetes cluster.

- **Pod-to-Pod communication**: each Pod is assigned a unique cluster-wide IP address. All the pods in the cluster can communicate without the use of a NAT or proxies through the *pod network*.

- **Pod-to-Service communication**: pods running an application are grouped under a long-lived IP address or hostname, exposing a service.
- **Service-to-External communication**: services are exposed to clients external to the cluster.

Network policies can be used to manage inter-cluster traffic or external traffic.

2.7.8 Microservices

Considering the requirements for a micro-services-based architecture, Kubernetes becomes a fundamental tool for its realization, and in particular, two Kubernetes elements have to be considered:

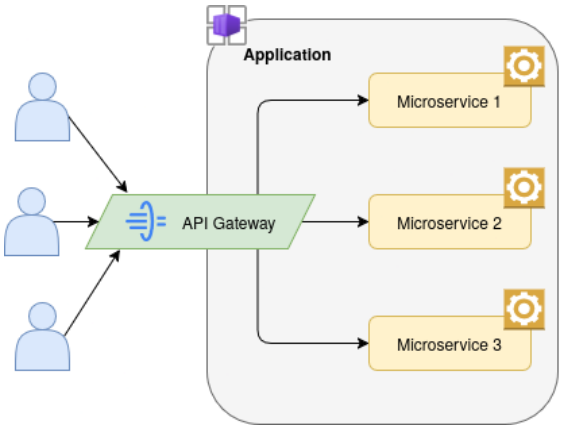

Figure 2.11: API Gateway

- **API Gateway**: a specific component, acting as a single entry point, designed to handle all requests coming from the client and forward the request to the required microservice (Figure 2.11).
- **Services Mesh**: additional network layers with the primary task of managing communication between different services, separating the network aspects from the application ones.

2.7.9 Service Meshes

In the context of a microservices-based architecture in Kubernetes, managing a large number of services is a crucial aspect. One of the primary challenges in this setup is handling the communication between services.

Service meshes provide a solution to this problem by introducing a dedicated communication layer. This solution is typically implemented by deploying a sidecar container

alongside the primary container where the service logic resides. The sidecar functions as a proxy, streamlining communication between services.

Without this approach, developers would have to build and manage the communication logic themselves. Consequently, the network is established through sidecar containers, which interact with the sidecars of other services to enable seamless communication.

- **Sidecar Proxy**, a lightweight component deployed alongside each service instance designed to handle critical functionalities essential for inter-service communication, including but not limited to load balancing, circuit breaking, and service discovery.
- **Control Plane**, it enables the specification of authentication policies, the generation of metrics, and the configuration of service proxies across the entire mesh.
- **Business Logic**, core application logic and underlying code of a microservice. This component is responsible for encapsulating the service's computational tasks and inter-service communication.
- **Primitive Network Functions**, utilized by a microservice to initiate network calls and establish connections with the service mesh sidecar proxy.
- **Application Network Functions**, maintain and manage critical network functions of the sidecar proxy.

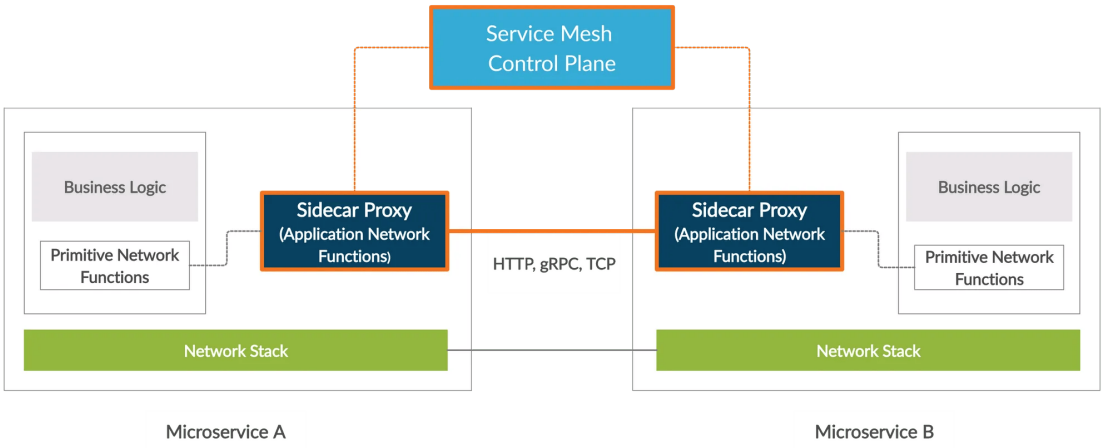

Figure 2.12: Service Mesh architecture [18].

The main Kubernetes open-source solutions for implementing service meshes are:

- **Istio**: a service mesh that provides advanced traffic management, security, and observability for microservices applications. It enables fine-grained control over service communication, including routing, retries, fault injection, and circuit breaking. Istio also ensures secure service-to-service communication through mutual TLS (mTLS) and integrates deeply with monitoring and logging tools to offer observability at scale.

- **Linkerd**: a service mesh designed to simplify microservices management while delivering essential features like secure service-to-service communication (mTLS), observability, and traffic routing. It focuses on ease of use and performance, making it ideal for less complex scenarios compared to other service meshes.

2.7.10 Workflow

All the resources in Kubernetes can be declared through YAML manifests, which contain all the details regarding the specified resource, such as the number of replicas for a specific pod, how it should be externally exposed, what ports should or can be used, and additional configuration files. Other key fields are those related to the containers that are to be deployed within the pod, to specify which image to use, details related to the privileges the container must have, and the resources it can consume or needs. An example manifest is provided by Listing 2.2.

Listing 2.2: Kubernetes YAML manifest example

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.21
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: "0.5"
            memory: "512Mi"
          requests:
            cpu: "0.2"
            memory: "256Mi"
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
```

```
35   selector:
36     app: nginx
37   ports:
38   - port: 80
39     targetPort: 80
40   type: ClusterIP
```

The YAML manifest can then be applied to the cluster using the command provided by the *Kubectl* command line tool for interacting with the cluster, which works by communicating with the Kubernetes API server, translating command-line instructions into API calls that the cluster understands. The pattern used for the commands is the following:

```
kubectl [command] [resource] [name] [flags]
```

2.8 Network Service Mesh

Before describing what Network Service Mesh (NSM) it is necessary to introduce the concept of workloads: a workload is a running application, and it is often executed in a runtime domain associated with a connectivity domain. Each workload has a different connectivity domain, however, it is common to have scenarios in which multiple workloads are achieving the same objective but running in different runtime domains. These workloads need to communicate with each other, and NSM allows us to overcome this challenge: different workloads have a way to connect to the required services, regardless of the runtime domain in which they are being run [19].

NSM is also runtime domain-independent and is complementary to the common service mesh solutions mentioned previously like Istio or Likerd. This is because NSM operates at a lower level, by focusing on traffic management, compared to the other mentioned service meshes, which focus on application-level traffic. The integration of these different solutions that operate at different levels of granularity allows for a fine-grained level of control over a cluster to better manage pods and services.

2.8.1 Architecture

Below, the main elements that compose the architecture of NSM are described.

- **Network Services**, the set of features to which workloads can connect, including connectivity, observability, and security. The network services allow for specifying the type of traffic that the workload has to handle. The default type is `IP`.
- **Network Service Endpoint**, the pod where the service(s) is provided. When the endpoint is deployed, its name, along with the list of the provided services and the ones it needs to connect to, is logged into a registry.
- **Network Service Registry**, a registry where both network services and endpoints are registered.

- **Network Service Client**, workload that needs to connect to a given network service. The client is authenticated and authorized before making the connection to the service. The connection to the service is made through an annotation by which the service is specified, the type of mechanism desired, and the name of the corresponding interface that is then created.

- **Network Service Manager**, responsible for discovering Network Services and Network Service Endpoints and processing requests from clients. It must be located on the same machine as the NSM client to provide it with a connection to the NSM network. It can also serve as a registry if there is no real registry.

- **Forwarder Vector Packet Processor**, uses the Vector Packet Processor Framework (VPP) as a backend to create network interfaces. VPP is a fast, scalable layer 2-4 multi-platform network stack that runs in Linux userspace.

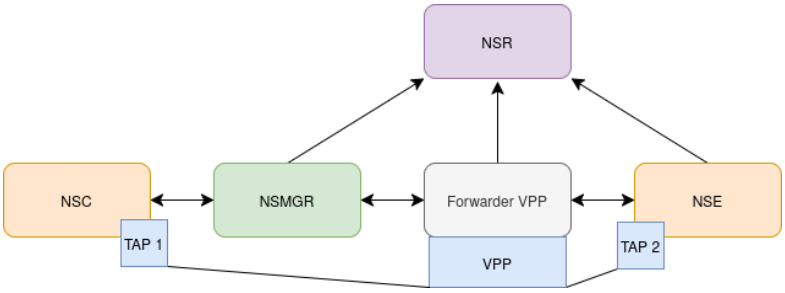

Figure 2.13: NSM architecture

As in figure 2.13, when an NSC requests a specific service, it must first pass through an NSMGR. This component is responsible for authenticating both the client and the request, thereby enhancing security. Typically, a separate NSM instance is deployed for each node in the cluster. The *Spiffe (Secure Production Identity Framework for Everyone)* and *Spire (SPIFFE Runtime Environment)* projects are leveraged to provide a robust identity management infrastructure within distributed environments, such as Kubernetes. These frameworks work in tandem to ensure secure communication between services.

Each NSC is typically assigned a unique identifier, known as a *SPIFFE ID*, which is essential for identifying and distinguishing components within the system. During the cluster configuration process, it is recommended to deploy dedicated components to manage this authentication procedure. After the authentication process is completed, the NSM determines the appropriate data plane to forward the request to, which is then processed by the forwarder VPP.

Upon arrival at the forwarder, the request parameters must be registered within a registry, including the service requested and the desired mechanism. Additional parameters, such as the interface name, are also specified, and the corresponding service endpoints are identified. Similarly, once the endpoint registers its available services within the registry, it will proceed to select the mechanisms it supports. In the final step, both the NSC and NSE

interfaces are created, and traffic is handled directly by the forwarder, which establishes a tunnel between the two endpoints.

Chapter 3

Implementation

The subsequent section presents a formal abstraction model for Kubernetes resources aimed at simplifying cluster orchestration and supporting migration across diverse technological ecosystems (e.g., CNIs, Service Meshes, Security tools). To support this model, an automated Python-based framework was developed, capable of analyzing Kubernetes manifest data and performing configuration translation based on the abstraction. This development was informed by a comparative analysis identifying key differentiators among prevalent Kubernetes extension categories.

3.1 Comparative analysis

A comparison of some key characteristics is needed to provide a comprehensive overview of the diverse landscape of Kubernetes network and security solutions. Tables 3.1, 3.2, 3.3, and 3.4 present a simplified view of the comparative analysis of key functional categories. The solutions reported in the comparison, listed below, have been chosen based on the wide adoption they have reached in the Kubernetes ecosystem.

- **Kube-router**, a turnkey solution for Kubernetes networking that provides service abstraction, support for network policies, and load-balancing functionality [20].
- **Flannel**, is a plugin focused on networking for providing a layer 3 IPv4 network between multiple nodes in a cluster; it does not control how containers are networked to the host, only how the traffic is transported between hosts [21].
- **Cilium**, a software for transparently securing the network connectivity between application services deployed using Linux container management platforms; it is based on eBPF, which enables the dynamic insertion of powerful security visibility and control logic within Linux itself. [22].
- **Calico**, a networking and security solution that enables Kubernetes workloads and non-Kubernetes/legacy workloads to communicate seamlessly and securely. Calico

consists of networking to secure network communication, and advanced network policy to secure cloud-native microservices/applications at scale [23].

- **Weave Net**, a CNI that creates a virtual network that containers across multiple hosts and enables their automatic discovery; with Weave Net, portable microservices-based applications consisting of multiple containers can run anywhere: on one host, multiple hosts, or even across cloud providers and data centers [24].

- **Romana**, a network and security automation solution for cloud-native applications that automates the creation of isolated cloud-native networks and secures applications with a distributed firewall that applies access control policies consistently across all endpoints and services [25].

- **Network Service Mesh**, a hybrid/multi-cloud IP Service Mesh enabling L3 Zero Trust, per workload granularity, per Network Service Connectivity/Security/Observability, and interoperability with existing CNI [19].

- **Istio**, a service mesh that layers transparently onto existing distributed applications, providing a uniform and more efficient way to secure, connect, and monitor services with load balancing, service-to-service authentication, and monitoring [26].

- **Cert-manager**, a tool for managing TLS certificates for workloads in a cluster that renews the certificates before they expire [27].

- **OVN-Kubernetes**, a project that provides a robust networking solution for Kubernetes clusters with Open Virtual Networking and Open Virtual Switch at its core that enables fine-grained cluster egress traffic controls and advanced networking features along with Kubernetes core networking conformance [28].

- **KubeArmor**, a cloud-native runtime security enforcement system that restricts the behavior (such as process execution, file access, and networking operations) of pods, containers, and nodes at the system level [29].

The comparative analysis utilizes functional categories chosen to provide a thorough characterization of the internal attributes of each Kubernetes extension. The functional categories are presented as follows.

- **Data plane(s)**, describes the underlying technologies and mechanisms used by each solution to handle packet forwarding and network traffic at the Data plane(s) level. Understanding this is crucial to assess performance, efficiency, and compatibility with different infrastructure environments.

- **Control Plane(s)**, details the components and processes involved in the control plane, which manages the network configuration, policy enforcement, and overall orchestration. This highlights the architecture and complexity of each solution's management.

- **K8s Native Network Policy Support**, indicates whether the solution natively supports Kubernetes Network Policies, a fundamental feature for implementing network segmentation and security within Kubernetes.
- **Network Authorization Policy Support**, explores the types of authorization policies supported, distinguishes between endpoint-based and identity-based approaches, and highlights the level of access control.
- **Load Balancing**, describes the load balancing mechanisms provided by each solution, crucial for distributing traffic and ensuring application availability within the Kubernetes cluster.
- **Security features**, summarizes the security features offered by each solution, encompassing aspects such as encryption, access control, and specific security policies. This is particularly important for assessing the security posture of applications deployed in Kubernetes.
- **Network Isolation**, examines the level of network isolation provided, from basic segmentation to more advanced, identity-based isolation, which is essential for multi-tenancy and security in complex deployments.
- **Packet Filtering**, focuses on the packet filtering capabilities, indicating the layers at which filtering can be performed (e.g., Layer 3, Layer 4, Layer 7) and the granularity of control.

Kube-router

Data plane(s). Kube-router makes use of the Linux kernel's IP Virtual Server (IPVS), Linux Virtual Server (LVS), iptables, ipset, and BGP. IPVS/LVS is used for service proxying, while iptables and ipset are used for implementing network policies. BGP is used for inter-node pod-to-pod communication. For pods residing on the same node, communication typically occurs through a local bridge interface. When traffic needs to traverse between pods on different nodes, the BGP-learned routes in the node's routing table ensure that the packets are forwarded to the correct destination node. Communication with Kubernetes Services is handled by Kube-router's service proxy, which utilizes IPVS/LVS for layer 4 load balancing. Traffic destined for a Service's ClusterIP or NodePort is intercepted and load-balanced across the backend pods that implement the service.

Control plane(s). The *Network Services Controller* monitors Kubernetes Services and Endpoints. When a new Service is created or an existing Service is updated, this controller reads the relevant information from the Kubernetes API server and configures IPVS on each node in the cluster. The *Network Policy Controller* is responsible for implementing Kubernetes Network Policies. This controller watches for changes in namespaces, network policies, and pods in the Kubernetes API. Based on the defined network policies, the controller configures iptables rules on each node to provide ingress filtering to the pods. The *Network Routes Controller* manages the routing of network traffic between pods across

different nodes. This controller monitors the pod CIDR allocated to each node by the Kubernetes controller manager. For each node, the NRC advertises the allocated pod CIDR to the rest of the nodes in the cluster using BGP.

K8s Native Network Policy Support. Kubernetes Network Policies are fully supported. The Network Policy Controller within Kube-router is responsible for watching for `NetworkPolicy` objects and associated pods, and then translating these policies into the appropriate iptables rules and ipset configurations on each node.

Network Authorization Policy Support. Kube-router does not directly support authorization policies, though these are provided through Kubernetes Network Policies.

Load Balancing. Kube-router offers Layer 4 load-balancing through the use of IPVS/LVS. Kube-router also supports advanced load balancing features like Direct Server Return (DSR). Kube-router primarily operates in IPVS masquerading mode. This means that when traffic from a client (either internal or external) reaches the IPVS virtual service, the destination IP address is translated to the IP address of one of the backend pods. In DSR mode, the load balancer forwards the initial request to a backend server, but the server responds directly to the client, bypassing the load balancer for the return traffic.

Security features. Kube-router includes a built-in Network Policy Controller which actively monitors the Kubernetes API server for `NetworkPolicy` objects and updates related to `Pods`. Upon detecting these changes, the controller translates the high-level network policies defined in Kubernetes into low-level rules within the Linux kernel's firewalling system, configuring iptables rules and ipset configurations. Kube-router also supports MD5 password-based authentication as an advanced BGP capability, adding a layer of security to the routing control plane.

Network Isolation. Basic isolation support through the use of Kubernetes network policies.

Packet Filtering. Basic packet filtering based on Kubernetes network policies.

Flannel

Data plane(s). *VxLAN* operates by encapsulating Layer 2 Ethernet frames within Layer 3 UDP packets, allowing the creation of an overlay network that can traverse any IP network. This "L2 over L3" capability makes *VxLAN* highly adaptable to various network environments, including cloud platforms where direct Layer 2 connectivity between nodes is not always available. *Host-GW* backend offers an alternative approach by directly updating the routing table of each host to facilitate inter-pod communication. When a pod on one node needs to communicate with a pod on another, Flannel adds a route on the source

node that directs traffic destined for the remote pod's subnet to the IP address of the destination node. The *WireGuard* backend leverages the Linux kernel's native *WireGuard* VPN capabilities to establish secure tunnels between nodes. This backend encapsulates the traffic and encrypts it, providing enhanced security for inter-pod communication. The UDP backend encapsulates IP packets within UDP datagrams but performs this encapsulation and decapsulation in user space, which can introduce significant performance overhead.

Control plane(s). The `flanneld` agent runs as a daemon on each host in the Kubernetes cluster. Its key responsibilities include allocating a unique subnet lease to the local host from the preconfigured address space. This involves communicating with the chosen backing store (*etcd* or Kubernetes API) to request and obtain a non-overlapping subnet. The `flanneld` agent also monitors the backing store for subnet information related to other hosts in the cluster, synchronizing this data locally to maintain awareness of the entire network topology. Furthermore, `flanneld` configures the local network stack on each node.

K8s Native Network Policy Support. Not supported.

Network Authorization Policy Support. Not supported.

Load Balancing. Not supported.

Security features. Both WireGuard and IPsec encryption methods are supported.

Network Isolation. Flannel, when configured with an overlay network backend like *VxLAN*, inherently provides a degree of network isolation. By encapsulating pod traffic within UDP packets and using a virtual network identifier (VNI), the overlay network creates a logical boundary that separates the pod network from the underlying physical network infrastructure.

Packet Filtering. Not supported.

Cilium

Data plane(s). based on *Extended Berkeley Packet Filter* (eBPF), a Linux kernel technology that permits the dynamic insertion of bytecode at various integration points within the kernel. Cilium forwards network traffic through a simple, flat Layer 3 network that can seamlessly span multiple Kubernetes clusters. This is achieved through the support of both native routing and overlay networking modes. In native routing mode, Cilium leverages the regular routing table of the Linux host, requiring the underlying network infrastructure to be capable of routing the IP addresses assigned to application containers, while overlay mode employs encapsulation-based virtual networks, such as

VxLAN and *Generic Network Virtualization Encapsulation (GENEVE)*, to span across all hosts. The latter approach has minimal infrastructure requirements, needing only basic IP connectivity between hosts.

Control plane(s). The Cilium Agent is a crucial component of the control plane, running as a daemon on each node within the Kubernetes cluster. This agent is responsible for writing `CiliumEndpoint` and `CiliumIdentity` custom resources to the Kubernetes API server, representing the network information and security identity of pods running on its node. It also updates the eBPF maps on the node, ensuring that the Data plane(s) enforce the correct policies based on these identities and endpoints. The Cilium Operator serves as a cluster-wide management component, automating various tasks related to Cilium's operation. It also handles the lifecycle of Cilium custom resources, such as `CiliumNetworkPolicy` and `CiliumClusterwideNetworkPolicy`. The Cilium CLI provides a command-line interface that allows administrators and developers to interact with the Cilium API server.

K8s Native Network Policy Support. Cilium fully implements the standard Kubernetes NetworkPolicy specification, meaning that any network policies already defined within a Kubernetes cluster will function as expected without requiring any modifications when Cilium is deployed. Cilium translates the high-level declarative network policies defined in Kubernetes into low-level eBPF programs that are then loaded directly into the Linux kernel on each node.

Network Authorization Policy Support. Cilium's network authorization policies heavily leverage the concept of identity-based security. Instead of relying on IP addresses, which can be dynamic and difficult to manage in Kubernetes, Cilium assigns a security identity to each group of application containers (pods) based on their associated labels. Policies are then defined based on these identities, specifying which identities are allowed to communicate with each other.

Load Balancing. for traffic entering or leaving the cluster, Cilium's eBPF implementation can be attached to eXpress Data Path (XDP) for extremely fast packet processing. It also supports Direct Server Return (DSR) and Maglev consistent hashing when the load balancing operation is not performed on the source host, further enhancing efficiency and scalability. In multi-cluster environments managed by Cilium's Cluster Mesh, global service load balancing is also supported.

Security features. Cilium's approach to Layer 3 and Layer 4 network policy enforcement is based on an identity-based security model that leverages Kubernetes labels. Labels offer more stability compared to transient IP addresses, ensuring that security rules remain effective even as the underlying infrastructure changes. These policies are implemented through Kubernetes CRDs, namely `CiliumNetworkPolicy` and `CiliumClusterwideNetworkPolicy`, which extend the capabilities of standard Kubernetes

Network Policies. Cilium also supports traditional CIDR-based security policies for managing access to and from external services. To enforce these Layer 7 policies, Cilium can examine the content of network packets at the application level and filter traffic based on HTTP methods, paths, headers, and DNS. For advanced Layer 7 policy requirements, Cilium integrates with Envoy proxy, which is then responsible for enforcing the more intricate Layer 7 rules. Cilium offers transparent encryption support through the use of either WireGuard or IPsec. When WireGuard is enabled, the agent running on each cluster node establishes secure WireGuard tunnels with all other known nodes in the cluster. This process is automated, with each node generating its encryption key pair and securely distributing its public key via Kubernetes annotations on the `CiliumNode` custom resource object. These public keys are then used by other nodes to encrypt and decrypt traffic originating from and destined to Cilium-managed endpoints running on those nodes. Configuring IPsec encryption within Cilium involves the use of Kubernetes secrets to manage and distribute pre-shared keys (PSKs). The process begins with generating a strong, random PSK that is then stored as a Kubernetes secret within the same namespace as the Cilium deployment. The Cilium agents, running as pods on each node, are configured to mount this secret, allowing them to access the necessary IPsec configuration. Cilium also offers real-time visibility network observability through Hubble, offering the possibility to interact with the collected data using a command-line interface or a user interface. Cilium also offers a valuable feature called Policy Audit Mode. When enabled, this mode allows all network traffic to pass through without being blocked by network policies, but it logs all connections that would have been dropped if the policies were in full enforcement mode.

Network Isolation. By implementing policies that initially block all ingress and egress traffic for specific pods or namespaces, it is possible to allow only the necessary communication paths selectively. Policies can also be defined based on pod labels, allowing for the isolation of individual applications or microservices even within the same cluster network. Cilium also supports network segmentation using multi-network configurations. This feature allows connecting pods to multiple network interfaces, enabling the segregation of traffic based on different security or functional requirements.

Packet Filtering. Cilium packet filtering is based on the network policies to filter at layers 3 and 4 for traditional network parameters such as source and destination IP addresses, ports, and protocols (TCP, UDP, etc.), and at layer 7 for application-level protocol data. Furthermore, policies are often defined based on the security identity of the source and destination endpoints, allowing Cilium to filter packets based on the labels and metadata associated with the communicating pods.

Calico

Data plane(s). Calico makes use of both iptables for high-performance IP forwarding and ipsets for efficient firewall routing. Calico Enterprise also supports eBPF for enhanced

performance, lower latency, and features that are not feasible with iptables, and the significant advantage of replacing the functionality of Kubernetes kube-proxy with native service handling. For Kubernetes clusters running on Windows nodes, Calico provides support for the Host Network Service (HNS). Additionally, Calico supports an integration with Vector Packet Processing (VPP) to accelerate the networking performance of Kubernetes clusters utilizing Calico significantly.

Control plane(s). *Felix* is a central agent that runs as part of the calico-node daemonset on each node in the Kubernetes cluster. It is responsible for managing all other Calico components on the node and for offering networking, network policy, and IP address management capabilities. The BGP client (BIRD) retrieves the routes inserted by Felix and distributes them to other nodes within the Calico deployment. *confd* is responsible for monitoring the Calico datastore for any changes related to BGP configuration and global default settings. The Calico API server provides a Kubernetes-native interface for managing Calico resources directly using the *kubectl* command-line tool. *calicoctl* is a command-line interface specifically designed for creating, reading, updating, and deleting Calico objects.

K8s Native Network Policy Support. Calico offers full support for the enforcement of native K8s network policies. Also, for broader, cluster-wide default deny requirements, Calico `GlobalNetworkPolicy` does exist.

Network Authorization Policy Support. Calico network policies can be applied to multiple types of endpoints, including pods, virtual machines, and host interfaces. Also, one key enhancement is the ability to define policy ordering and priority. Calico network rules support `Allow`, `Deny`, `Pass`, and `Log` actions.

Load Balancing. Calico does not directly implement load balancing. The Calico Ingress Gateway is based on Envoy, which provides a standardized and vendor-neutral approach to managing external traffic entering a Kubernetes cluster.

Security features. Calico network policies can be applied to a broader range of endpoints compared to native network policies. Furthermore, Calico introduces enhancements such as the ability to define the order or priority in which policies are evaluated, the inclusion of explicit deny rules, and more flexible options for matching network traffic. Calico provides the `GlobalNetworkPolicy` CRD to apply network policy to the entirety of the cluster instead of being namespace-restricted. Calico also presents the concept of policy tears. These tiers provide an additional layer of organization and ordering for global policies. When a policy with a *Pass* action matches traffic, instead of immediately allowing or denying the traffic, it skips the remaining policies within the current tier and proceeds to evaluate policies in the next tier. Calico leverages labels and namespaces to implement dynamic segmentation policies, so that they are not tied to static IP addresses, which can change frequently in dynamic container environments. Within a policy, Calico offers the

possibility to filter both Layer 3/4 packets (TCP, UDP, ICMP, CIDR rules) and Layer 7 ones (HTTP, DNS). Calico supports WireGuard encryption on the host-to-host portion of the network path. This means that when a pod on one node communicates with a pod on another node, the traffic is encrypted as it travels between the underlying host machines, but the traffic between a pod and the host it resides on remains unencrypted. Furthermore, Calico supports the `Log` actions to record network events that are filtered based on the specified rules in a policy.

Network Isolation. Calico's network isolation feature is provided by the network policies, offering namespace, pod, and host isolation. Also, Network Segmentation can be achieved in Calico by using its IP Address Management (IPAM) capabilities in conjunction with network policies to divide a large network CIDR into smaller blocks of IP addresses and assign these blocks to different nodes in the cluster.

Packet Filtering. Felix, the Calico agent running on each node, is responsible for programming the necessary iptables and ipsets rules based on the defined network policies. Calico also offers an eBPF-based filtering mechanism.

Weave Net

Data plane(s). Layer 2 overlay achieved through UDP encapsulation. It operates in two modes. Sleeve mode, operating in user space, captures network packets on the Linux bridge using *pcap* devices and encapsulates them within UDP via the *wRouter* component. Fastpath mode operates within the kernel space, utilizing Open vSwitch's (OVS) *odp* for VxLAN encapsulation and forwarding. Instead of directly forwarding packets, the *wRouter* manages them through *odp* flow tables, significantly boosting throughput.

Control plane(s). There are *wRouters* on each host to establish full mesh TCP links with one another. These links are the foundation for synchronizing control information using a Gossip protocol, a distributed communication method used in networking and distributed systems that allows nodes to exchange information with randomly selected peers periodically, making information propagate exponentially through the network without central coordination.

K8s Native Network Policy Support. Weave Net provides support for Kubernetes native network policy enforcement thanks to the Network Policy Controller (*weave-npc*), which actively monitors the Kubernetes API for any `NetworkPolicy` objects defined within the cluster.

Network Authorization Policy Support. Not supported.

Load Balancing. Weave Net implements *weaveDNS*, an integrated DNS-based load balancer, facilitating dynamic service discovery and load balancing by distributing DNS queries for a given service name across the IP addresses of the underlying containers using a round-robin strategy.

Security features. Weave Net utilizes NaCl encryption for sleeve traffic, which operates in user space, and IPsec ESP encryption for fast datapath traffic, which operates in kernel space. As previously discussed, Weave Net also offers full support for Kubernetes native network policies.

Network Isolation. Basic network isolation is provided by network policies. Weave Net also offers the "isolation-through-subnets" technique, which enables the hosting of multiple isolated applications on a single Weave network by creating separate subnetworks within the overlay.

Packet Filtering. Basic packet filtering is supported based on network policies. In Fastpath mode, Weave Net also utilizes the Open vSwitch (OVS) datapath for packet forwarding. OVS offers more advanced packet filtering.

Romana

Data plane(s). Romana handles traffic employing a fully routed network model in which each endpoint is assigned a real, routable IP address. These addresses are configured directly on the underlying cluster hosts. Romana supports a topology-aware IPAM system to assign IP addresses based on an understanding of the underlying network topology, including the arrangement of hosts, racks, and availability zones.

Control plane(s). *romana-etcd* provides the crucial function of accessing etcd storage. The *romana-daemon* is a central, long-running service that acts as the core orchestrator for the entire Romana system. The *romana-listener* is a specialized background service designed to monitor the stream of events originating from the Kubernetes API Server. *romana-agent* is a local agent that runs on all Kubernetes nodes. It installs the CNI tools and configuration necessary to integrate Kubernetes CNI mechanics with Romana and manages node-specific configuration for routing and policy.

K8s Native Network Policy Support. Full support for native Kubernetes network policies is offered by bridging the gap between Kubernetes' policy model and its internal mechanisms by seamlessly translating the Kubernetes `NetworkPolicy` objects into its internal policy format.

Network Authorization Policy Support. Not supported.

Load Balancing. Not supported.

Security features. The topology-aware IP Address Management (IPAM) system assigns natively routable IP addresses to endpoints, eliminating the need for overlays or tunnels. Network isolation in Romana is primarily achieved through the implementation of iptables Access Control Lists (ACLs), which are configured based on the defined network policies.

Network Isolation. Romana provides network policy-based isolation, also supported by topology-aware IPAM.

Packet Filtering. Packet filtering in Romana is handled with network policies.

Network Service Mesh

Data plane(s). The *Forwarder* is responsible for creating and managing both client-side and endpoint-side network interfaces, selecting the appropriate connection mechanisms to facilitate communication, collecting operational statistics from these interfaces, and performing load balancing for multiple endpoints that provide the same Network Service. The *vWire* represents a bidirectional, virtual connection established between a Network Service Client (NSC) and a Network Service Endpoint (NSE) and serves as the dedicated pathway for all communication between the connected client and endpoint.

Control plane(s). The Network Service Manager (NSMGR) is the primary control entity on each node. When an NSC requests a connection to a Network Service, the local NSMGR accepts this request and queries the API Server, which serves as the service registry, for available NSEs. The Network Service Registry (NSR) is a central component that stores information about all the key elements of the NSM infrastructure, including the defined Network Services, the registered Network Service Endpoints, and the running NSMGR instances. For seamless integration with Kubernetes environments, NSM includes an Admission Webhook. This Kubernetes component automatically injects NSCs into Kubernetes pods based on specific NSM annotations present in the pod or namespace specifications.

K8s Native Network Policy Support. Not supported.

Network Authorization Policy Support. authorization mechanism based on workload identity, leveraging the same Spiffe ID that is used for Layer 7 communication and extending this framework of cryptographic identity-based auditability down to Layer 3 of the network stack. Whether a client is permitted to establish a connection to a specific Network Service is determined by defined policies.

Load Balancing. depends on the specific implementation.

Security features. NSM enforces the Zero Trust principle by default, mandating verification for every connection and transaction and granting users and services the minimum necessary access to perform their functions. Each NSC is independently authenticated with *SPIFFE* ID and must be explicitly authorized to establish a connection with the desired Network Service. The *vWire* between an NSC and an NSE ensures that network traffic originating from a specific client is exclusively routed to its intended endpoint and vice versa. NSM also makes use of mTLS to verify the identity of the communicating parties and ensure that the data exchanged is protected from eavesdropping and tampering. NSM allows for the creation of multiple independent and mutually unaware Registry Domains. Each Registry Domain can manage its own set of Network Services and Endpoints, providing a form of administrative isolation between different groups of services or tenants. Furthermore, NSM allows for the composition of security features into Network Services, e.g., an Intrusion Prevention System (IPS) can be selectively inserted into the traffic path between clients and the virtual L3 network based on client labels, offering enhanced security for specific communication flows.

Network Isolation. *vWires* an isolated Layer 3 connection dedicated to a specific NSC and NSE. NSM also achieves network isolation by decoupling Network Services from the underlying Runtime Domain, allowing workloads from different environments, such as separate companies connected to a shared NSM, to collaborate on specific tasks without exposing their entire network infrastructure. NSM also supports the deployment of multiple independent Registry Domains to create logical isolation boundaries between different sets of services or tenants.

Packet Filtering. depends on the specific implementation.

Istio

Data plane(s). Istio data plane is composed of Envoy proxies, which are deployed as sidecars alongside each microservice instance. These proxies intercept and control all network communication between the microservices within the mesh, and collect and report telemetry data on all mesh traffic.

Control plane(s). *Istiod* offers service discovery, configuration management, and certificate management. It takes high-level routing rules that control traffic behavior and translates them into Envoy-specific configurations, which are then propagated to the sidecar proxies at runtime. *Pilot* monitors the high-level routing rules and security policies configured within Istio and translates them into low-level instructions that each Envoy proxy can understand and enforce. *Citadel* focuses on ensuring the security of the service mesh. It oversees security policies for Envoy proxies, handling tasks such as user authentication, certificate management, and ensuring the encryption of network data through mTLS. *Galley* acts as the configuration validation, ingestion, processing, and distribution component.

K8s Native Network Policy Support. Not supported.

Network Authorization Policy Support. Authorization rules in Istio are specified using a specific CRD, `AuthorizationPolicy`, and operate at Layer 7. By default, if no `AuthorizationPolicy` is applied to a workload, all requests to that workload are allowed. Istio Authorization Policies support both `ALLOW` and `DENY` actions, as well as `AUDIT` and custom actions to allow integration of external authorization systems or custom logic.

Load Balancing. Different load balancing algorithms are supported. By default, Istio employs a least requests load balancing policy, where traffic is sent to the instance with the fewest active requests, aiming to distribute the load evenly. Other strategies available are round-robin, random, consistent hash, e.g...

Security features. Istio automatically enables mTLS encryption for all communication within the service mesh to ensure that all data transmitted between services is encrypted in transit. Istio also provides identity management with X.509 certificates formatted according to the *SPIFFE* standard. Istio uses the `AuthorizationPolicy` resource to define access rules, specifying the selector, an action (ALLOW, DENY, AUDIT), and the rules to enforce, then the Envoy sidecar proxies evaluate these policies for every incoming request and enforce the decision. Istio also supports the `PeerAuthentication` resource for authentication, by specifying the selector and the mTLS mode (PERMISSIVE, STRICT, DISABLE).

Network Isolation. Sidecar configurations can be used to limit the reachability of services and fine-tune the set of ports and protocols that an Envoy proxy will accept. These configurations can be applied to all workloads within a specific namespace or targeted to particular workloads using workload selectors based on labels.

Packet Filtering. Packet filtering is handled by Envoy proxies.

Cert-manager

Data plane(s). Cert-manager does not implement a data plane; instead, it extends the Kubernetes API by using CRDs to define certificate authorities (`Issuers` and `ClusterIssuers`) and to manage certificate requests through the `Certificate` resource.

Control plane(s). The *Controller* takes actions to ensure the desired state is maintained, driving the certificate issuance and renewal processes. The *Webhook* acts as an admission controller, validating and defaulting Cert-manager API resources upon creation or update. *cainjector* monitors Kubernetes Secrets containing CA certificates and ensures that other resources, such as deployments and stateful sets that rely on these CA certificates for trust, are automatically updated when the CA certificate changes. *acmesolver* handles challenges

required by ACME-based Issuers. *startupapicheck* runs at the startup of Cert-manager components to verify that the Cert-manager API is available and functioning correctly.

K8s Native Network Policy Support. Not supported.

Network Authorization Policy Support. The optional *CertificateRequestPolicy* custom resource to automatically approve or deny CertificateRequests based on specific criteria is supported.

Load Balancing. Not supported.

Security features. Cert-manager extends the Kubernetes API by introducing several CRDs, such as `Certificate`, `Issuer`, `ClusterIssuer`, `CertificateRequests` to define the desired state of certificates and the sources from which they should be obtained. Cert-manager securely manages and stores sensitive information, primarily private keys and issued certificates, within Kubernetes Secrets, which provide a secure mechanism for holding sensitive information within the cluster. When Cert-manager issues a certificate, the resulting private key and signed certificate are stored as key-value pairs within a Secret in the same namespace as the corresponding `Certificate` resource. Beyond the use of Kubernetes Secrets, Cert-manager also supports alternative methods for private key storage through Container Storage Interface (CSI) drivers. When these drivers are used, the private key is generated on-demand, just before the application starts, and it never leaves the node where the application pod is running. The Cert-manager webhook operates as an admission controller within Kubernetes, intercepting requests to the Kubernetes API server for the creation or modification of Cert-manager resources. The webhook requires a TLS certificate that the API server trusts to establish a secure HTTPS connection.

Network Isolation. Cert-manager can be configured to use namespace-scoped `Issuer` resources, providing isolation between different namespaces and ensuring that certificates are only issued within the intended scope.

Packet Filtering. Not supported.

OVN-Kubernetes

Data plane(s). The main component is Open vSwitch (OVS), which is responsible for making packet forwarding decisions based on a set of rules defined by the OpenFlow protocol. For communication between nodes in the cluster, OVN-Kubernetes utilizes Generic Network Virtualization Encapsulation (GENEVE) tunnels to create an overlay network.

Control plane(s). in the centralized control plane setup, the *ovn-kube-master* runs on the Kubernetes control plane nodes, monitoring objects like namespaces, pods, services, and network policies from the API; it converts these changes into OVN logical entities (switches, routers, ACLs) and stores them in the *Northbound Database* (NBDB). Meanwhile, the *Southbound Database* (SBDB) maintains network state and switch port bindings for OVS, with *ovn-northd* translating NBDB entries into SBDB data. The *ovn-kube-cluster-manager*, running in the *ovn-kube-control-plane* pod, oversees cluster-wide tasks. In the interconnect (distributed) control plane setup, the NBDB/SBDB are hosted on each worker node within the *ovn-kube-node* pod. Here, *ovn-kube-controller* takes on the node-specific responsibilities of *ovn-kube-master*, watching local Kubernetes objects, translating them into OVN entities, and storing them in the local NBDB, while allocating pod IP addresses. A local *northd* translates these entries into logical flows in the local SBDB, and *ovn-controller* on each node retrieves the flows. Though distributed, cluster-wide configurations remain coordinated through the *ovn-kube-cluster-manager*.

K8s Native Network Policy Support. the OVN-Kubernetes controller watches for changes to `NetworkPolicy` resources in the Kubernetes API server. When a new network policy is created or an existing one is updated, the controller translates the policy into corresponding OVN ACL rules, which are stored in the NBDB. The OVN controller running on each node reads the ACL rules from the SBDB.

Network Authorization Policy Support. Not supported.

Load Balancing. OVN-Kubernetes directly implements Kubernetes Services and support for `EndpointSlices` using its own native OVN Load Balancers to ensure efficient routing of traffic for both internal communication between pods within the cluster and external traffic destined for services exposed outside the cluster.

Security features. OVN-Kubernetes implements Kubernetes Network Policies through OVN ACLs, which allows for fine-grained control over network traffic at the pod level. Each new `NetworkPolicy` in OVN-Kubernetes leads to the creation of a port group, and pods matching the policy's `podSelector` are added to this group. Subsequently, ACLs are applied to these port groups to enforce the defined rules. OVN-Kubernetes supports the encryption of network traffic within the cluster using IPsec in Transport mode, encrypting the payload of the IP packet. In "Full" mode, all pod-to-pod traffic between nodes is encrypted, and optionally, traffic to external hosts can also be encrypted after additional configuration. "External" mode allows for encrypting only traffic destined for external IPsec endpoints. OVN also leverages Kubernetes RBAC to control access to network-related resources and can be configured to log network traffic, providing visibility into network activity.

Network Isolation. OVN's logical networks allow each namespace to have its logical switches and routers. Kubernetes Network Policies, translated into OVN ACLs,

let administrators define allowed communication paths with fine-grained control. Also, GENEVE-based overlay networking additionally encapsulates pod traffic, isolating it from the physical network.

Packet Filtering. ACLs can be configured to filter network traffic based on a wide range of criteria, including source and destination IP addresses, port numbers, and network protocols.

KubeArmor

Data plane(s). KubeArmor utilizes eBPF to observe and intercept system calls made by containers and pods. This allows it to monitor network-related system calls like `connect()`, `bind()`, `listen()`, `accept()`, etc., without requiring modifications to the application code or the container image. KubeArmor integrates with Linux Security Modules (LSMs) such as AppArmor, SELinux, and BPF-LSM. When a network-related system call is made, KubeArmor's eBPF probes can trigger policy checks against these LSMs. Based on the defined policies, the LSM can either allow or deny the operation.

Control plane(s). The KubeArmor Controller runs as a deployment within the Kubernetes cluster and watches for the creation, update, and deletion of KubeArmor policy CRDs. When changes occur, the controller translates these policies into eBPF and LSMs. The KubeArmor Agent runs as a DaemonSet on each node in the cluster and is responsible for receiving policy updates from the KubeArmor Controller, loading and enforcing the policies using eBPF and configuring the relevant LSMs, monitoring system events, and providing logs and telemetry data.

K8s Native Network Policy Support. Not supported.

Network Authorization Policy Support. KubeArmor supports its form of network authorization policy through its `KubeArmorPolicy` CRD that allows defining policies with criteria based on process identity, destination IP address, protocol and port, and system call type.

Load Balancing. Not supported.

Security features. KubeArmor defines different CRDs to cover different needs. The command `KubeArmorPolicy` enforeces policies on pods, `KubeArmorHostPolicy` focuses on the host machines, and `KubeArmorClusterPolicy` can be used to apply cluster-wide policies. These policies allow to application of restrictions on process execution, file access, (basic) network activity, and system calls usage. KubeArmor policies allow setting Allow, Deny, or Audit actions when the specified condition in the policy is met. The Audit mode allows for recording events without blocking them.

Network Isolation. by controlling network-related system calls, KubeArmor can prevent containers from establishing unauthorized network connections, thus isolating them from unintended targets. KubeArmor can also enforce policies at the host level via the `KubeArmorHostPolicy`.

Packet Filtering. Not supported.

Category	Kube-router	Flannel	Cilium
Data plane(s)	IPVS, iptables, IP Routing	VxLAN, UDP, Host-GW,WireGuard	eBPF
Control plane(s)	Network Routes Controller, Network Policy Controller, Service Proxy Controller	flanneld daemon, etcd, K8s APIs	Cilium Agend, Cilium Operator, Cilium CLI, Cilium CNI, Hubble
K8s native network policy support	Yes	-	Yes
Network authorization policy support	-	-	Identity-based. RBAC support
Load balancing	DSR, IPVS/LVS	-	XDP and DSP for intra-cluster, Cilium ClusterMesh for inter-cluster
Security features	Network policies and BGP	WireGuard, IPsec	L3, L4 and L7 rules. Identity-based policies. IPsec and WireGuard encryption for node-to-node traffic. Fine-grained control
Network isolation	Basic	Basic	Identity-based isolation and security groups
Packet filtering	Basic	-	Identity-based, L3/4/7 and XDP-accelerated filtering

Table 3.1: Comparison table 1

Category	Calico	Weave Net	Romana
Data plane(s)	iptables, ipsets, Windows HNS, eBPF, VPP	Layer 2, UDP encapsulation, Sleeve and Fastpath mode	Routed network model IPAM
Control plane(s)	Felix Daemon, confd, BIRD, calicoctl	wRouters, Gossip protocol	romana-etcd, romana-daemon romana-listener, romana-agent
K8s native network policy support	Yes	Yes	Yes
Network authorization policy support	Endpoint-based, ordering, priority, actions	-	-
Load balancing	-	weaveDNS	-
Security features	Fine-grained control, Default-deny beahviour	Traffic encryption, access control, authentication and authorization	Firewalling, topology-aware IPAM
Network isolation topology-aware IPAM	Network policies	Virtual overlay network	Network policy-based
Packet filtering	Workload Endpoints, Policy Enforcement	Network policy-based, OVS	Network policy-based

Table 3.2: Comparison table 2

Category	Network Service Mesh	Istio	Cert-manager
Data plane(s)	Forwarder, vWire	Envoy Proxies	Interaction with K8s API
Control plane(s)	NSMGR, NSR, Admission Webhook	Istiod with Pilot and Citadel	Controller, Webhook, cainjector, acmesolver startupapicheck
K8s native network policy support	-	-	-
Network authorization policy support	Spiffe ID, policy-driven	AuthoriazionePolicy CRD	Optional CertificateRequestPolicy
Load balancing	Depending on implementation	Built-in traffic management features	-
Security features	Per-workload granularity, Spiffe ID, additional components like IPS	Mutual TLS, X.509, SPIFFE	Automatic certificate renewal, ACME support, PK management, RBAC, external issuer support, certificate revocation
Network isolation	NS-RD decoupling, vWire, multiple NSRs	Fine tuning of sidecar configurations	Namespaced-scoped Issuer resources
Packet filtering	Depending on implementation	Envoy proxies	-

Table 3.3: Comparison table 3

Category	OVN-Kubernetes	KubeArmor
Data plane(s)	Open vSwitch (OVS), OpenFlow, GENEVE tunnels, overlay network	System calls interception, eBPF, LSMs
Control plane(s)	ovn-kube-master, Kubernetes API, NBDB, SBDB, ovn-northd, ovn-kube-cluster-manager, ovn-kube-node, ovn-kube-controller, IPAM	KubeArmor Agent, KubeArmor Controller
K8s native network policy support	Yes	-
Network authorization policy support	-	Endpoint-based
Load balancing	OVN Load Balancers, Kubernetes Services, EndpointSlices, internal/external traffic, service routing	-
Security features	ACLs, GENEVE tunneling, IPsec, TLS, Kubernetes RBAC, traffic logging	Runtime protection, file integrity monitoring, privileged container control, host security, policy-based security
Network isolation	Logical networks, namespace-level switches/routers, OVN ACLs, GENEVE-based overlay, fine-grained control	Network-related system calls Host policies
Packet filtering	ACLs, source/destination IP, port/protocol filtering	-

Table 3.4: Comparison table 4

3.2 Kubernetes Abstract Model

The subsequent section of this work delivers a comprehensive analysis of the implementation of the abstract model for the Kubernetes resources and the automated Python scripts designed for parsing Kubernetes YAML manifests bidirectionally as JSON objects.

3.2.1 Model details

The model is composed of a series of Python dataclasses divided into two major sections:

- **Main Entities**: these entities model the Kubernetes resources that are identified by the presence of the *"kind"* field inside the YAML manifest. Main entities represent a distinct resource type in the Kubernetes API and include core resources (Pods, Services, Deployments, etc.), extended resources (NetworkPolicies, CRDs, etc.), and ecosystem-specific resources (Cilium, Calico, etc. components). A main entity can be composed of both mandatory and optional parameters. Required fields common to all main entities are *"apiVersion"* and *"kind"*. An example of a main entity is *"DaemonSet"* (Listing 3.1).

Listing 3.1: Example of a main entity: DaemonSet class

```
class DaemonSet {
    -apiVersion: str
    -template: PodTemplate
    -metadata: Dict[str, str | Dict[str, str]]
    -selector: Dict[str, str]
    -status: Optional[Dict[str, int]]
    -updateStrategy: Optional[Strategy]
    -minReadySeconds: Optional[int]
    -revisionHistoryLimit: Optional[int]
    -kind: str
}
```

- **Sub-Entities**: these are nested resources that are referenced within *"main entities"* but do not exist as independent resources with their own *"kind"* attribute. A *"sub entity"* provides a structured representation for complex structures and commonly reused patterns within Kubernetes objects. In the same way as main entities, sub-entities can make use of both mandatory and optional parameters. An example of a sub entity, referred to in the *"DaemonSet"* example (Listing 3.1), is the *"Strategy"* class (Listing 3.2).

Listing 3.2: Example of a sub entity: Strategy class

```
class Strategy {
    -type: str
    -rollingUpdate: Optional[RollingUpdate]
}
```

A sub-entity can make use of other sub-entities, like in the case of *"Strategy"*, which makes use of the *RollingUpdate* entity (Listing 3.3).

Listing 3.3: RollingUpdate class

```
class RollingUpdate {
    -maxSurge: Optional[int]
    -maxUnavailable: Optional[int]
}
```

Figures 3.1, 3.2, and 3.2 are simplified representations of some class of the model, highlighting the separation between main entities and sub-entities, and their relations within the model.

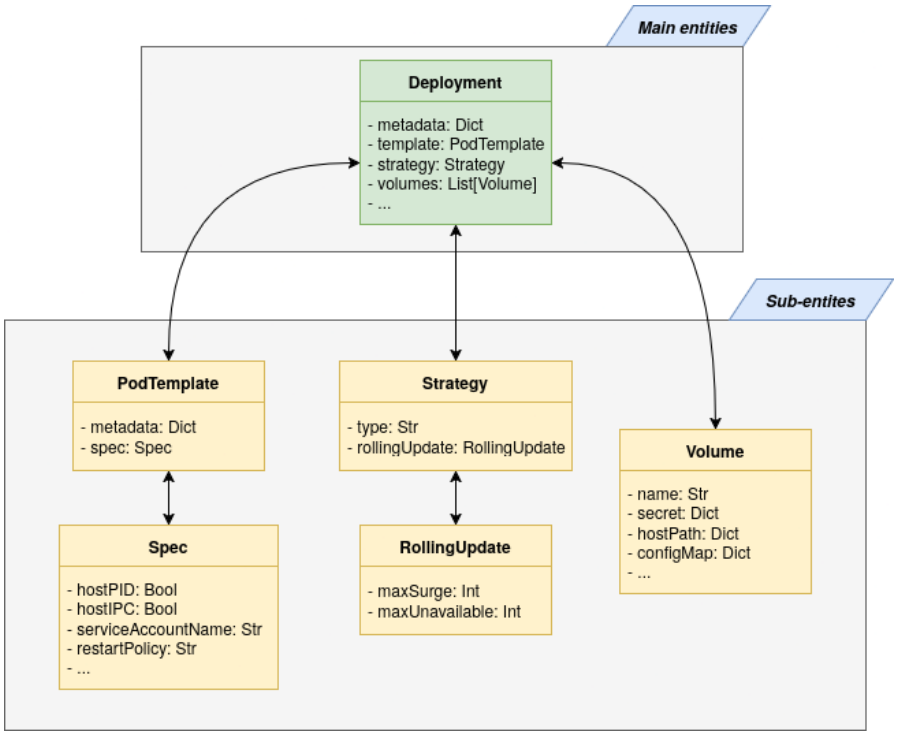

Figure 3.1: Deployment class

3.2.2 Automation Framework

This framework aims to facilitate the conversion process of the manifests of a cluster using a certain solution to a desired one. Within the Kubernetes ecosystem, resources are specified using YAML manifests, which encompass all required parameters for resource definition. The objective of this framework is to streamline the process of analyzing Kubernetes manifests and produce a valid instance of the model that can describe the

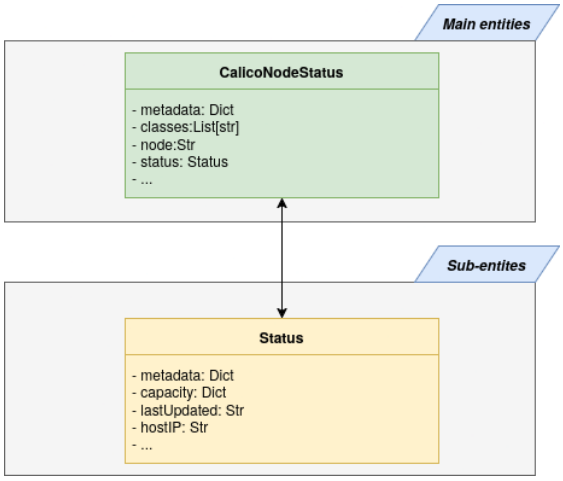

Figure 3.2: CalicoNodeStatus class

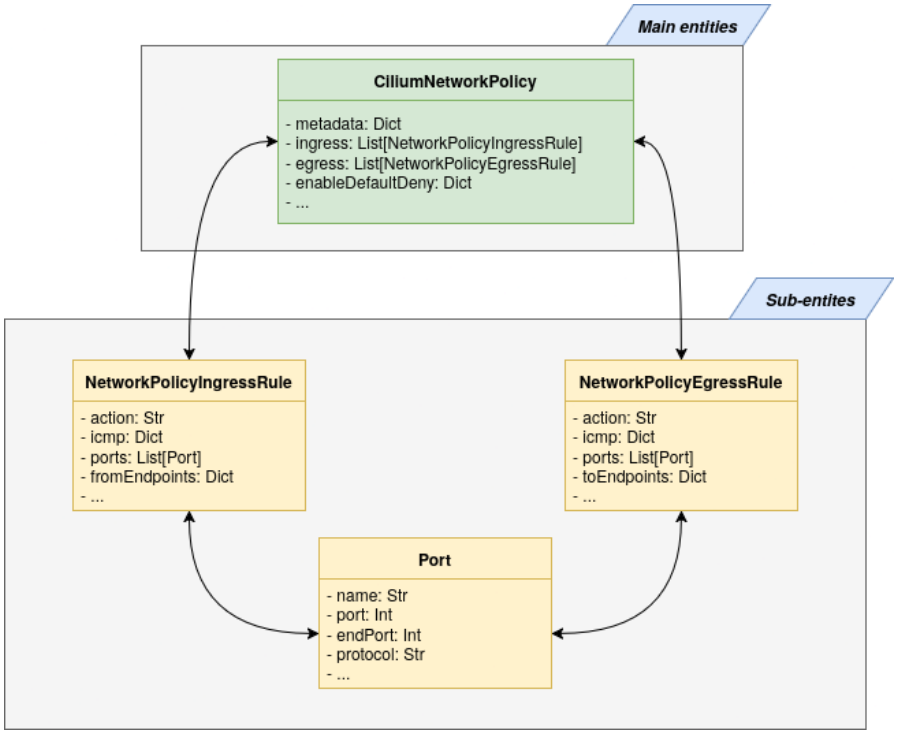

Figure 3.3: CiliumNetworkPolicy class

resources with abstract classes. Contextually, the framework is also able to compute resources expressed in terms of the abstract model and output valid Kubernetes manifests that can be instantiated on a different tool compared to the starting one.

Feature analyses: importing configurations from existing networks

This script analyzes a YAML manifest representing a Kubernetes resource. Based on the input data, it generates a JSON file that captures an instance of the abstraction model that can describe the input resource with generic classes. Furthermore, the script identifies any term within the manifest that cannot be correlated with a resource defined by the model (Listing 3.4), by comparing the keys of the provided field with the ones expected by the model. The output file that is produced represents a generalization of the input resources that allows for the conversion to a different tool than the one used in the input manifests.

The listing **??** represents an example of an instance of the model in the form of a JSON object obtained as a result of the processing of the *Feature analysis* script.

Listing 3.4: check_unexpected_keys function

```
def check_unexpected_keys(data, expected_keys):
    if isinstance(data, list):
        for item in data:
            check_unexpected_keys(item, expected_keys)
    else:
        unexpected_keys = set(data.keys()) - set(expected_keys)
        if unexpected_keys:
            print(f"[!] Unexpected keys: {unexpected_keys}")
```

Main processing The central function of the script involves extracting all fields within the parsed manifest and differentiating between the *"spec"* keys and the other *"top level"* keys (such as kind and metadata). Utilizing the *kind_to_class* (Listing 3.5) and *kind_to_subclass* (Listing 3.5) dictionaries, it constructs the appropriate model instances for each resource type. It establishes the hierarchical relationships among these objects, facilitating the instantiation of the model.

Listing 3.5: kind_to_class dictionary

```
kind_to_class = {
    ...
    'GatewayClass': GatewayClass,
    'DestinationRule': DestinationRule,
    'ServiceMonitor': ServiceMonitor,
    'PodMonitor': PodMonitor,
    ...
}
```

Listing 3.6: kind_to_subclass dictonary

```
kind_to_subclass = {
    ...
    'roleRef': RoleRef,
    'subjects': Subject,
```

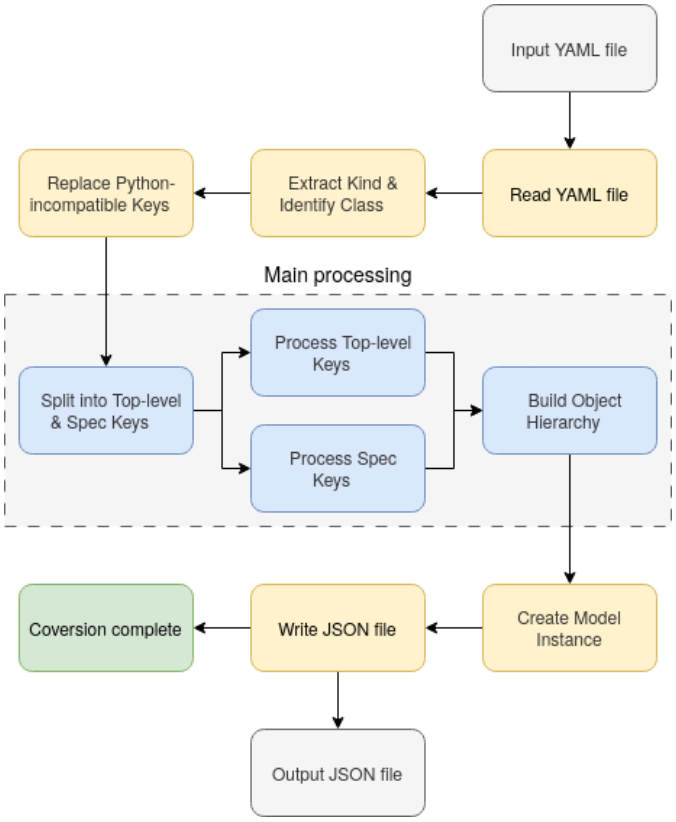

Figure 3.4: Feature analysis workflow

```
    'trafficPolicy': TrafficPolicy,
    'endpoints': Endpoint,
    ...
}
```

Feature translation: converting abstract configurations

This script is designed to generate a valid YAML Kubernetes manifest, translated to the desired output tool, from a JSON file that embodies an instance of the abstract model.

Main processing The primary function of the script involves the pre-processing of top-level keys by modifying the *top_level_keys* data structure, which encompasses all conceivable top-level keys of the model. These modifications are executed by eliminating specific terms from *top_level_keys*, contingent upon the type of resource (indicated by the *kind* parameter) currently being processed. After this step is completed, the input data is split between top-level keys and spec ones, and everything is combined into a single dictionary.

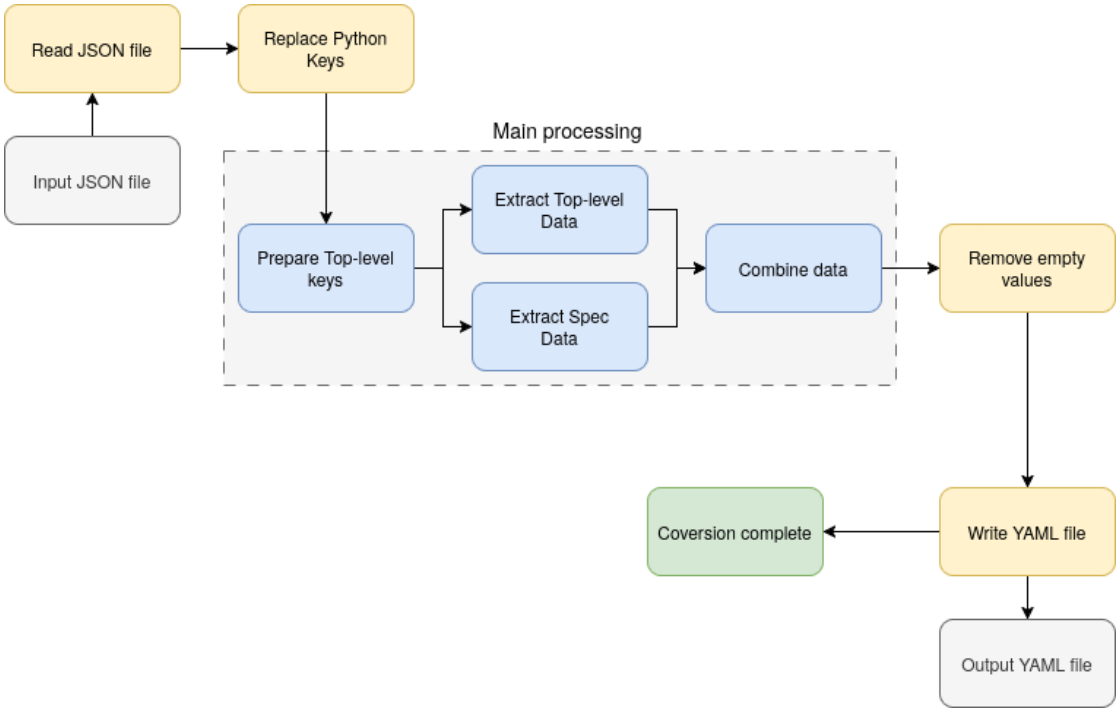

Figure 3.5: Feature translation workflow

Listing 3.7: Example of high-level description label

```
[
    {
        "apiVersion": "cilium.io/v2",
        "metadata": {
            "name": "dev-to-kube-apiserver"
        },
        "description": null,
        "endpointSelector": {
            "matchLabels": {
                "env": "dev"
            }
        },
        "ingress": null,
        "ingressDeny": null,
        "egress": [
            {
                "action": null,
                "destination": null,
                "http": null,
                "icmp": null,
                "ipVersion": null,
                "metadata": null,
                "notICMP": null,
                "notProtocol": null,
```

```
                "protocol": null,
                "to": null,
                "ports": null,
                "toEndpoints": null,
                "toPorts": null,
                "toFQDNs": null,
                "toEntities": [
                    "kube-apiserver"
                ],
                "toCIDR": null,
                "toCIDRSet": null,
                "toGroups": null,
                "toNodes": null,
                "toRequires": null,
                "icmps": null,
                "toServices": null,
                "auth": null,
                "authentication": null
            }
        ],
        "egressDeny": null,
        "enableDefaultDeny": null,
        "kind": "CiliumNetworkPolicy"
    }
]
```

Chapter 4

Results and Validation

In this section, the results of the comparative analysis are presented, as well as the validation of the Kubernetes abstract model with the Python-based automated framework. Furthermore, performance considerations related to the processing of the automated framework have been taken into consideration.

4.1 Comparative Analysis

The comparative analyses revealed that the landscape of Kubernetes networking and security tools presents a diverse range of solutions, often with overlapping capabilities reflecting an evolution from basic connectivity to complex traffic management and security enforcement. Initially, CNI plugins like Flannel, Weave Net, and Romana focused primarily on establishing pod-to-pod communication using various data plane techniques such as VxLAN overlays, direct routing, or user-space encapsulation. Over time, many CNIs evolved significantly; Kube-router utilizes IPVS and BGP, Calico integrates BGP with iptables/ipsets and optionally eBPF, OVN-Kubernetes leverages Open vSwitch and GENEVE, while Cilium heavily relies on eBPF for high-performance networking and security. This evolution led CNIs like Calico, Cilium, Kube-router, Weave Net, OVN-Kubernetes, and Romana to incorporate support for Kubernetes Network Policies, adding crucial L3/L4 security features directly at the networking layer, with some like Calico and Cilium even extending these with custom resources for more granular control. Concurrently, service meshes like Istio emerged, operating primarily at Layer 7 using sidecar proxies (Envoy) to provide advanced traffic management, observability, and identity-based security (mTLS, Authorization Policies), but typically not implementing the core CNI functionalities or K8s Network Policies directly. Network Service Mesh focuses on providing L3 network connectivity itself as a programmable service via identity-secure vWires, differing from Istio's application-layer focus by enabling composition of network functions and leveraging SPIFFE identity for L3 authorization rather than relying on sidecars for L7 control. Complementary tools address specific needs: Cert-manager automates the lifecycle of TLS certificates, essential for securing communication often orchestrated by CNIs or service

meshes, while KubeArmor focuses on runtime security, using eBPF and LSMs to enforce policies based on system calls, process, and file activity, offering a distinct layer of defense compared to network-centric policies provided by CNIs like Calico and Cilium or service meshes like Network Service Mesh and Istio. This ecosystem shows a trend where CNIs are expanding into security and even basic service mesh functionalities (especially eBPF-based ones like Cilium), while dedicated service meshes provide richer L7 capabilities, and specialized tools handle orthogonal concerns like certificate management or host/process-level security.

4.2 Automated Framework

To evaluate the automated framework, two distinct scenarios have been taken into account:

- **Java Microservices** [30]: The files contained within the "manifest" directory were chosen for the analysis.
- **Online Boutique** [31]: In the root directory of the repository, the YAML manifests located within the "kubernetes-manifests" and "istio-manifests" directories have been analyzed.

For each operational scenario under investigation, a rigorous validation protocol was implemented. This protocol involved a detailed assessment of the feature analysis stage, focusing on the verification of resources identified by the framework and the resultant model instance, which was serialized in JSON format. Complementary to the functional validation, a quantitative analysis of the framework's computational performance was conducted. This evaluation employed standard temporal metrics to characterize the execution efficiency. The specific metrics considered were:

- **Total Execution Time**: This metric represents the complete wall-clock time elapsed from the initiation to the termination of the function of the framework's process for a scenario instance.
- **Average Execution Time**: Calculated as the arithmetic mean of the execution times · recorded across different files (YAML manifests in the case of feature analysis, JSON objects in the case of translation). This metric offers an indication of the typical or expected temporal cost associated with executing a function of the framework.
- **Standard Deviation (of Execution Time)**: This statistical measure quantifies the degree of dispersion or variability observed in the recorded Total Execution Times relative to the calculated Average Execution Time.

4.2.1 Java Microservices

Figure 4.1 illustrates the feature analysis of the `deploy-cloudant.yaml` manifest. All resources specified within the manifest are accurately identified, resulting in the creation of

an instance of the model, as demonstrated in Listing 4.1. Additionally, Figure 4.2 presents the performance metrics associated with the execution time of the feature analysis process, while Figure 4.3 shows the execution time for the translation from the model instance to a Kubernetes manifest. Listing 4.2 shows the translation result from the model instance (Listing 4.1) to a deployable Kubernetes manifest.

```
Analyzign examples/ibm-java-ms/deploy-cloudant.yaml...
Kind: PersistentVolume
Kind: PersistentVolumeClaim
Kind: Service
Kind: Deployment
File analyzed!
```

Figure 4.1: Java Microservices example

```
Total execution time: 0.001419 seconds
Average execution time: 0.000118 seconds per file
Standard deviation: 0.000059 seconds
Number of files processed: 12
```

Figure 4.2: Java Microservices feature analyses process time computation

```
Total execution time: 0.000901 seconds
Average execution time: 0.000075 seconds per file
Standard deviation: 0.000042 seconds
Number of files processed: 12

Total execution time: 0.002320 seconds
```

Figure 4.3: Java Microservices translation process time computation

Listing 4.1: Model instance as a JSON object

```
[
    {
        "apiVersion": "v1",
        "metadata": {
            "name": "cloudant-pv",
            "labels": {
                "app": "microprofile-app"
            }
        },
        "capacity": {
            "storage": "4Gi"
        },
        "accessModes": [
            "ReadWriteMany"
        ],
        "storageClassName": null,
        "persistentVolumeReclaimPolicy": "Recycle",
        "volumeMode": null,
        "storageos": null,
        "hostPath": {
            "path": "/var/cloudant"
        },
        "gcePersistentDisk": null,
        "kind": "PersistentVolume"
    },
    {
        ...
        "kind": "PersistentVolumeClaim"
    },
    {
        ...
        "kind": "Service"
    },
    {
        ...
        "kind": "Deployment"
    }
]
```

Listing 4.2: Kubernetes manifest obtained from model instance

```
apiVersion: v1
kind: PersistentVolume
metadata:
  labels:
    app: microprofile-app
  name: cloudant-pv
spec:
  accessModes:
  - ReadWriteMany
  capacity:
```

```
    storage: 4Gi
  hostPath:
    path: /var/cloudant
  persistentVolumeReclaimPolicy: Recycle
---
...
kind: PersistentVolumeClaim
---
...
kind: Service
---
...
kind: Deployment
```

4.2.2 Online Boutique

Figure 4.4 illustrates the results derived from the feature analysis executed on the `cartservice.yaml` manifest file. The analysis process successfully identified all resources declared within this manifest. Consequently, a corresponding model instance was generated, the structure of which is detailed in Listing 4.3. To demonstrate the framework's handling of non-supported input, Figure 4.5 presents an example case where a manifest includes a resource definition containing an unsupported key, specifically `number`. Upon encountering this key, the analysis framework issues a warning message. This warning explicitly flags the unsupported key (`[!] Unexpected keys: 'number'`) and indicates the specific resource definition within the manifest where the deviation was detected (`kind: ServiceEntry`). Furthermore, in Figure 4.6, a report about the execution time for the feature analysis process is provided, while Figure 4.7 shows the execution time for the translation from the model instance to a Kubernetes manifest. Listing 4.4 shows the translation result from the model instance (Listing 4.3) to a deployable Kubernetes manifest.

```
Analyzign examples/google-boutique/cartservice.yaml...
Kind: Deployment
Kind: Service
Kind: ServiceAccount
Kind: Deployment
Kind: Service
File analyzed!
```

Figure 4.4: Online Boutique example

```
Analyzign examples/google-boutique/allow-egress-googleapis.yaml...
Kind: ServiceEntry
[!] Unexpected keys: {'number'}
[!] Unexpected keys: {'number'}
Kind: ServiceEntry
[!] Unexpected keys: {'number'}
[!] Unexpected keys: {'number'}
File analyzed!
```

Figure 4.5: Online boutique example with unexpected key

```
Total execution time: 0.002398 seconds
Average execution time: 0.000160 seconds per file
Standard deviation: 0.000093 seconds
Number of files processed: 15
```

Figure 4.6: Online Boutique feature analysis process time computation

```
Total execution time: 0.002065 seconds
Average execution time: 0.000138 seconds per file
Standard deviation: 0.000081 seconds
Number of files processed: 15

Total execution time: 0.004463 seconds
```

Figure 4.7: Online Boutique translation process time computation

Listing 4.3: Model instance as a JSON object

```
[
    {
        "apiVersion": "v1",
        "metadata": {
            "name": "redis-cart",
            "labels": {
                "app": "redis-cart"
            }
        },
        "selector": {
            "app": "redis-cart"
        },
        "ports": [
            {
                "name": "tcp-redis",
                "port": 6379,
                "endPort": null,
                "targetPort": 6379,
                "containerPort": null,
                "nodePort": null,
                "appPort": null,
                "appProtocol": null,
                "hostPort": null,
                "protocol": null,
                "dnsName": null,
                "l7proto": null,
                "l7": null
            }
        ],
        "clusterIP": null,
        "clusterIPs": null,
        "sessionAffinity": null,
        "externalTrafficPolicy": null,
        "ipFamilies": null,
        "ipFamilyPolicy": null,
```

```
        "externalIPs": null,
        "externalName": null,
        "loadBalancerSourceRanges": null,
        "internalTrafficPolicy": null,
        "type": "ClusterIP",
        "status": null,
        "kind": "Service"
    },
    {
        ...
        "kind": "Deployment"
    },
    {
        ...
        "kind": "Service"
    },
    {
        ...
        "kind": "ServiceAccount"
    },
    {
        ...
        "kind": "Deployment"
    }
]
```

Listing 4.4: Kubernetes manifest obtained from model instance

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis-cart
  name: redis-cart
spec:
  ports:
  - name: tcp-redis
    port: 6379
    targetPort: 6379
  selector:
    app: redis-cart
  type: ClusterIP
---
...
kind: Deployment
---
...
kind: Service
---
...
kind: ServiceAccount
---
...
```

```
26 kind: Deployment
```

4.3 Summary of Validation Findings

The validation process confirmed the utility and effectiveness of the developed components. The comparative analysis provided a thorough and useful overview of the Kubernetes networking and security tool landscape, highlighting the evolution and overlapping capabilities of various solutions like CNIs, service meshes, and specialized security tools. The abstract Kubernetes model demonstrated its effectiveness, validated through an automated framework using distinct real-world scenarios (IBM Java Microservices and Google Online Boutique). This framework's feature analyzer accurately identified all specified resources within the test manifests, successfully generating corresponding instances of the abstract model and appropriately handling unsupported keys with warnings. Furthermore, the feature translator reliably converted these abstract model instances back into deployable Kubernetes manifests, confirming the model's representational adequacy and the translator's functional correctness. Performance metrics for both the analysis and translation processes were also systematically measured.

Chapter 5

Conclusion

This thesis addressed the growing complexity within the Kubernetes ecosystem, particularly concerning the diverse landscape of network plugins, operators, and security tools. Through a comprehensive comparative analysis of key solutions like Flannel, Calico, Cilium, Network Service Mesh, and Kube-router, this work highlighted their distinct features, performance aspects, and security implications.

A central contribution of this research is the development of an abstract model designed to represent a wide array of Kubernetes resources in a unified manner. This model provides a foundation for simplifying cluster management and facilitating the migration process between different technological solutions. An automated Python framework was implemented to operationalize this model. This framework successfully demonstrated its capability to parse Kubernetes manifests, translate them into the abstract model representation, and convert the model back into deployable configurations.

The effectiveness of the abstract model and the automation framework was validated using real-world deployment scenarios, including Google Online Boutique and IBM Java microservices. The results confirm the model's ability to encapsulate diverse resource types and the framework's utility in analyzing and translating configurations.

Overall, this work offers valuable insights into the Kubernetes networking and security landscape and provides a practical toolset for managing complexity and improving interoperability within cloud-native environments. Future work could involve expanding the abstract model to encompass an even broader range of Kubernetes resources and extensions, and refining the translation capabilities of the framework to accommodate more complex scenarios.

Appendix A

User manual

A.1 System setup

Below are listed all the operations and components needed to set up and configure the system correctly.

A.1.1 Requirements

The following components are needed to run the project:

- Python version 3.11 and above
- pip version 24.2 and above
- Docker Engine
- Docker Desktop (optional)

A.1.2 Setup repository

The project can be initialized in two ways:

- as a Docker container by running:

```
./docker.sh
```

- directly in a Python virtual environment:

```
./venv.sh
source .venv/bin/activate
```

A.2 Usage

The project can be executed by running the following command:

```
python transforms.py
```

Which presents the following options:

1. `YAML to JSON`, which parses YAML files to JSON ones.
2. `JSON to YAML`, which parses JSON files to YAML ones.
3. `Both`, which executes, in order, option 1 and option 2.

Upon choosing one of the options above, a list of available examples is presented inside the `examples` directory. Once the execution ends, the results can be found inside the `output` directory.
Example:

```
> python transforms.py

    [1] YAML to JSON
    [2] JSON to YAML
    [3] Both
    Enter 1, 2, or 3:

> 3

    - google-boutique
    - kubeflix
    - real-apps
    - bookinfo
    - java-ms
    - cilium
    - calico
    - kube-armor
    Enter folder name:

> calico

    [YAML --> JSON]
    [...]
    Converting examples/calico/cluster-info.yaml to JSON...
    Kind: ClusterInformation
    File converted!

    [JSON --> YAML]
    [...]
    Converting output/yaml_to_json/calico/cluster-info.json to YAML...
    File converted!
```

To add new examples, it is simply necessary to create a new folder inside `examples` and put it inside the YAML files to parse.

When converting from YAML, any unsupported resource or term is displayed in the execution console.

It is also possible to execute the single conversion functions to parse single files, by running the following command and specifying the required paths:

- *YAML to JSON*

  ```
  python yaml_to_json.py /path/to/yaml ./path/to/output_dir
  ```

  Example:

  ```
  python yaml_to_json.py examples/bookinfo/bookinfo.yaml
      ./output/yaml_to_json/bookinfo
  ```

- *JSON to YAML*

  ```
  python json_to_yaml.py /path/to/json ./path/to/output_dir
  ```

  Example:

  ```
  python json_to_yaml.py output/yaml_to_json/bookinfo/bookinfo.json
      ./examples/json_to_yaml/bookinfo
  ```

Appendix B

Developer manual

B.1 Adding new entities

B.1.1 Model expansion

To add new entities to the model, it is necessary to edit the file

`k8s_model.py`

This file is located inside the `model` directory and is divided into two sections:

- *Sub entities*, for all the resources that do not have a `kind` attribute.
  Example:

```
@dataclass
class Cinder:
    volumeID: str = field(default_factory=str)
    fsType: str = field(default_factory=str)
    readOnly: Optional[bool] = field(default_factory=None)
```

- *Main entities*, for all the resources with a `kind` attribute.
  Example:

```
@dataclass
class Job:
    apiVersion: str = field(default_factory=str)
    metadata: Dict[str, str | Dict[str, str]] = field(default_factory=str)
    template: PodTemplate = field(default_factory=PodTemplate)
    kind: str = 'Job'
```

For the new entities, all the top-level members must be specified, with the exception made for the `spec` key, whose internal keys have to be added directly.

To reflect the changes inside the PlantUML schema, it is necessary to edit the file

`k8s_model.puml`

The file is organized like its Python counterpart, adding the new resources to their specific sections and specifying any new relationships between entities. Example:

- *Sub entities*

```
class Cinder {
   -volumeID: str
   -fsType: str
   -readOnly: Optional[bool]
}
```

- *Main entities*

```
class Job {
   -apiVersion: str
   -metadata: Dict[str, str | Dict[str, str]]
   -template: PodTemplate
   -kind: str
}
```

- *Relationships*

```
Job "1" -- "1" PodTemplate
```

The schema images can be manually generated, given that `plantuml` is installed, with the following commands:

- *PNG*

```
plantuml -tpng model.puml
```

- *SVG*

```
plantuml -tsvg model.puml
```

B.1.2 Conversion functions

To make the conversion functions recognize the new entities added to the model, it is necessary to add the new resources to the following mappings at the top of `yaml_to_json.py`:

- `kind_to_class` for the main entites.
- `key_to_subclass` for the sub entities.

Inside the file `json_to_yaml.py`, the variable `top_level_keys` contains all the top-level keys of the model, and any top-level keys added with a new resource have to be added to the data structure.
Example:

```
class BGPFilter {
   -apiVersion: str
   -metadata: Dict[str, str | Dict[str, str]]
   -exportV4: Optional[List[BGPFilterRule]]
   -exportV6: Optional[List[BGPFilterRule]]
   -importV4: Optional[List[BGPFilterRule]]
   -importV6: Optional[List[BGPFilterRule]]
   -kind: str
}

top_level_keys = {
    [...], exportV4, exportV6, importV4, importV6
}
```

Additionally, ad-hoc modifications to `top_level_keys` are needed for any eventual resource that contains a non-top-level key that is, however, present in `top_level_keys`.
Example:

```
class Service {
   -apiVersion: str
   -metadata: Dict[str, str | Dict[str, str]]
   -selector: Optional[Dict[str, str]]
   -ports: List[Port]
   [...]
   -internalTrafficPolicy: Optional[str]
   -type: str
   -status: Optional[Status]
   -kind: str
}

if kind == 'Service':
    top_level_keys_copy.remove('type')
```

Additionally, since the Python naming convention forbids the use of specific terms and characters, a `keys_to_replace` dictionary is present in both `yaml_to_json.py` and `json_to_yaml.py` to substitute the terms in the model with the correct ones.
Example:

```
class ReferenceGrant {
   -apiVersion: str
   -metadata: Dict[str, str | Dict[str, str]]
   -from_: Optional[List[Dict[str, str]]]
   -to: Optional[List[Dict[str, str]]]
   -kind: str
}
```

yaml_to_json.py

```
keys_to_replace = {
    [...],
    'from': 'from_'
}
```

json_to_yaml.py

```
keys_to_replace = {
    [...],
    'from_': 'from'
}
```

Bibliography

[1] Red Hat. *What is virtualization?* URL: https://www.redhat.com/en/topics/virtualization/what-is-virtualization.

[2] IBM. *What Is virtualization?* URL: https://www.ibm.com/topics/virtualization.

[3] Aapo Kalliola, Shankar Lal, Kimmo Ahola, Ian Oliver, Yoan Miche, and Silke Holtmanns. «Testbed for security orchestration in a network function virtualization environment». In: *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2017.

[4] Geeksforgeeks. *Network Functions Virtualization*. URL: https://www.geeksforgeeks.org/network-functions-virtualization.

[5] Cloudflare. *What is software-defined networking (SDN)?* URL: https://www.cloudflare.com/learning/network-layer/what-is-sdn.

[6] Geeksforgeeks. *What is Software Defined Networking (SDN)?* URL: https://www.geeksforgeeks.org/software-defined-networking.

[7] Nathan Sousa, Danny Perez, Raphael Rosa, Mateus Santos, and Christian Esteve Rothenberg. «Network Service Orchestration: A Survey». In: *Computer Communications* 142 (2018).

[8] Introduction to Microservices Architecture. *Medium*. URL: https://medium.com/the-modern-scientist/introduction-to-microservices-architecture-f0c7eefe79f1.

[9] IBM. *What is containerization?* URL: https://www.ibm.com/topics/containerization.

[10] Xu Zhiqun, Chen Duan, Hu Zhiyuan, and Sun Qunying. «Emerging of Telco Cloud». In: *China Communications* 10.6 (2013).

[11] Geeksforgeeks. *Architecture of Docker*. URL: https://www.geeksforgeeks.org/architecture-of-docker.

[12] Docker. *Docker Compose*. URL: https://docs.docker.com/compose/.

[13] IBM. *The evolution of Kubernetes*. URL: https://www.ibm.com/think/topics/kubernetes-history.

[14] Kubernetes. *Kubernetes documentation*. URL: https://kubernetes.io/docs/home/.

[15] Gerald Budigiri, Christoph Baumann, Jan Tobias Mühlberg, Eddy Truyen, and Wouter Joosen. «Network Policies in Kubernetes: Performance Evaluation and Security Analysis». In: *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*. 2021.

[16] Boris Lublinsky, Elise Jennings, and Viktória Spišaková. «A Kubernetes 'Bridge' Operator between Cloud and External Resources». In: *8th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*. 2023.

[17] Shixiong Qi, Sameer G. Kulkarni, and K. K. Ramakrishnan. «Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability». In: *IEEE Transactions on Network and Service Management* 18.1 (2021).

[18] GCore. *Explaining Microservices and Service Mesh with Istio*. URL: `https://gcore.com/learning/decoding-service-mesh-architecture-for-docker`.

[19] Network Service Mesh. *NSM Concepts*. URL: `https://networkservicemesh.io/docs/concepts/enterprise_users/`.

[20] Kube-router. *Introduction*. URL: `https://www.kube-router.io/docs/`.

[21] Flannel. *How it works*. URL: `https://github.com/flannel-io/flannel`.

[22] Cilium. *What is Cilium?* URL: `https://docs.cilium.io/en/stable/overview/intro/`.

[23] Calico. *About Calico*. URL: `https://docs.tigera.io/calico/latest/about`.

[24] Weave Net. *Introducing Weave Net*. URL: `https://rajch.github.io/weave/overview/`.

[25] Romana. *Welcome to Romana*. URL: `https://romana.readthedocs.io/en/latest/welcome.html`.

[26] Istio. *What is Istio?* URL: `https://istio.io/latest/docs/overview/what-is-istio/`.

[27] cert-manager. *Introduction*. URL: `https://cert-manager.io/docs/`.

[28] OVN-Kubernetes. *Overview*. URL: `https://ovn-kubernetes.io/`.

[29] KubeArmor. *KubeArmor*. URL: `https://docs.kubearmor.io/kubearmor`.

[30] IBM. *Enable your Java microservices with advanced resiliency features leveraging Istio*. URL: `https://github.com/IBM/resilient-java-microservices-with-istio`.

[31] Google. *Online Boutique*. URL: `https://github.com/GoogleCloudPlatform/microservices-demo`.