


Politecnico di Torino

**Politecnico di Torino**

Master's Degree in Cybersecurity

A.a. 2025/2026

Graduation Session October 2026

# LLM-based Intent-to-Policy Translation in Kubernetes Environments

Supervisors:
Alessio Sacco
Fulvio Valenza

Candidate:
Carolina Ferrari

**Abstract**

Containerized applications have become a fundamental component of modern software systems, with Kubernetes serving as a standard platform for orchestrating them at scale. This widespread adoption has increased the need for effective network security policies that control communication between workloads within a cluster. Defining such policies, however, requires knowledge of low-level YAML formats that vary across Container Network Interface plugins, along with a precise understanding of the underlying network model. Most operators know what they want to enforce, few know exactly how to write it.

This thesis addresses this gap by extending an existing tools, KubeTeus, with a multi-LLM pipeline for intent-based network policy management. The proposed approach allows operators to express security requirements in natural language and automatically translates them into enforceable Kubernetes network policies. The pipeline combines several LLMs with distinct roles: one interprets the natural language intent and produces a structured representation, another generates the corresponding policy configuration, a further model validates and corrects the generated policy and a final component generates tests to verify its behavior directly in the cluster. The system supports both Cilium and Calico and does not require the operator to manually write or configure network policies.

The proposed approach was evaluated on a benchmark of 100 intents covering different levels of specificity and both supported CNI plugins. The pipeline achieved a success rate of approximately 96%, with most observed failures arising from test infrastructure limitations or transient model communication errors rather than from incorrect policy generation. Policy generation, enforcement and testing were completed within one to two minutes, with a token consumption ranging from 15k to 35k tokens depending on the complexity of the intent, with more generic intents generally requiring more tokens. Overall, the results demonstrate the feasibility of combining natural language interfaces with automated validation and testing for intent-based network policy management.

The work shows that a multi-LLM architecture can reduce the technical effort required to translate security intent into enforceable Kubernetes policies while maintaining reliable performance across different networking environments.

# Table of Contents

**List of Tables** v

**List of Figures** vi

**1 Introduction** 1

**2 Background** 4
2.1 Docker . . . 4
2.2 Kubernetes . . . 6
2.2.1 Kubernetes Network Policies . . . 7
2.2.2 Calico . . . 10
2.2.3 Cilium . . . 12
2.2.4 From Native Limitations to CNI-Based Enforcement . . . 15
2.3 Security Applications . . . 16
2.4 Intent Based Networking . . . 18
2.5 Large Language Models . . . 19

**3 Related works** 21
3.1 Nimbus . . . 22
3.2 Kagent . . . 23
3.3 KubeTeus . . . 24
3.3.1 KubeTeus Architecture . . . 25

**4 System Architecture** 26
4.1 Architectural Design . . . 27
4.2 Main Components . . . 28
4.2.1 Cluster Discovery Layer . . . 28
4.2.2 Intent Queue and Request Handling . . . 29
4.2.3 Cluster Context Enrichment . . . 29
4.2.4 Reasoning Service . . . 29
4.2.5 Enforcement and Feedback Layer . . . 30

4.3 Comparison with the Original Architecture . . . . . 30

**5 Multi-Agent Reasoning Pipeline** 32
5.1 Architecture . . . . . 33
5.1.1 Graph Structure . . . . . 33
5.1.2 Shared State . . . . . 34
5.1.3 Retry Mechanism . . . . . 35
5.2 Agents . . . . . 35
5.2.1 Intent Translation . . . . . 35
5.2.2 Policy Generation . . . . . 36
5.2.3 Policy Validation . . . . . 37
5.2.4 Testing . . . . . 38
5.3 Implementation Details . . . . . 38
5.3.1 Policy Semantics . . . . . 39
5.3.2 Handling Model Failures . . . . . 39
5.3.3 Model and Provider Configuration . . . . . 40
5.4 Relationship with the Controller . . . . . 41

**6 Runtime Verification, Enforcement and Feedback Loop** 42
6.1 Controller Architecture and Execution Model . . . . . 43
6.1.1 Controller-Runtime Integration . . . . . 43
6.1.2 Initialization and Cache Management . . . . . 43
6.2 From Reasoning Output to Cluster State . . . . . 44
6.2.1 Direct Policy Application . . . . . 44
6.2.2 Policy Validation . . . . . 45
6.2.3 Conflict Detection . . . . . 46
6.2.4 Enforcement . . . . . 48
6.2.5 Runtime Testing . . . . . 48
6.2.6 Rollback . . . . . 49
6.2.7 Recording Outcomes . . . . . 49
6.3 Discussion . . . . . 50

**7 Evaluation** 52
7.1 Kagent . . . . . 52
7.2 Nimbus . . . . . 54
7.3 KubeTeus . . . . . 55
7.4 Aegis . . . . . 56
7.5 Evaluation Overview . . . . . 58
7.5.1 CNI Coverage . . . . . 59
7.5.2 Human Intervention Requirements . . . . . 60
7.5.3 Token Consumption . . . . . 61

7.5.4 Execution Time . . . . . 63
7.5.5 Comparative Results . . . . . 65
7.5.6 Overall Comparison Across Metrics . . . . . 66

**8 Conclusions and Future Work** 68
8.1 Limitations . . . . . 69
8.2 Future Work . . . . . 69

**Acronyms** 71

**Bibliography** 73

# List of Tables

5.1 Model configuration used in the different pipeline stages. . . . . 40

7.1 Coverage of the evaluated intents across the supported policy mechanisms. . . . . 54

7.2 CNI coverage of the four tools, as implemented and tested. . . . . 60

# List of Figures


2.1 Docker architecture [5] . . . . . 5
2.2 Kubernetes architecture [9] . . . . . 7
2.3 Overview of the main components of the Calico architecture. The architecture includes components such as Felix, BIRD, confd, the CNI plugin, the datastore, and the Kubernetes controllers. . . . . 10
2.4 Overview of the main components of the Cilium architecture. . . . . 13
2.5 LLM architecture . . . . . 19

3.1 Architecture of Nimbus [27] . . . . . 22
3.2 Architecture of KubeTeus [25] . . . . . 25

4.1 High-level architecture: from intent submission to enforcement and rollback. . . . . 27

5.1 Directed graph of the reasoning pipeline. . . . . 32

7.1 Number of human interventions required per intent by Aegis and Kagent. . . . . 60
7.2 Comparison of tokens used per intent by the evaluated language-model-based systems. . . . . 62
7.3 Time required per type of intent by Aegis and Kagent. . . . . 63
7.4 Comparison of time required per intent by Aegis, Kagent, and KubeTeus. . . . . 64
7.5 Pass rate across the four tools on a shared batch of ten intents, with confidence intervals. . . . . 65

# Chapter 1

# Introduction

Kubernetes has become the de facto standard for orchestrating containerized applications, and its widespread adoption has brought with it an equally widespread problem: controlling how workloads inside a cluster are allowed to talk to each other. Network policies exist precisely for this, yet writing one correctly is harder than the concept suggests. An operator needs to know not just what traffic should pass, but the specific YAML syntax of the Container Network Interface (CNI) plugin that is running on the cluster. For instance, Calico and Cilium overlay their own extensions on Kubernetes' native NetworkPolicy API. The operator who understands the security requirement is rarely the person comfortable turning it into a CNI-specific manifest, and even when the operator happens to be both, the process remains slow and prone to mistakes. This is not merely an inconvenience; recent work simulating attacks against misconfigured Kubernetes clusters has shown that the underlying weaknesses exploited, including missing or overly permissive network segmentation, are typically caused not by unknown vulnerabilities but by avoidable misconfiguration, and that enforcing a correct network policy is often one of the more practical mitigations available [1].

That gap is where this thesis starts. One possible way to address it is to move the interaction with the network to a higher level of abstraction. Instead of requiring operators to know the syntax and configuration model of every CNI they might use, the system can accept a description of the desired network behaviour and translate it into the configuration required by the underlying technology. This is the idea behind Intent-Based Networking (IBN). The operator simply specifies the desired result, without needing to know the configuration required to achieve it.

IBN has been explored in different forms for several years but applying this approach to Kubernetes network policies introduces an additional challenge. A generated manifest must not only be syntactically valid, but also semantically correct when it is enforced on a live cluster. Natural-language intents are inherently ambiguous and may omit details such as ports, protocols, namespaces, or the exact

labels associated with a workload. A system translating these intents therefore has to resolve this ambiguity, either by asking the operator for additional information or by using the information available from the cluster itself.

This thesis takes the second route. Large Language Models (LLMs) suit this kind of translation task well, since they are already reasonably good at turning loosely structured text into structured output. The resulting system is called Aegis, from the ancient Greek for shield, and the name will be used in the rest of the thesis. Aegis presents a multi-model pipeline of several LLMs built on top of the existing tool KubeTeus for generating Kubernetes network policies based on intent. In fact, splitting a complex generation task across several specialized models, each handling a specific stage, has been shown to be more effective than asking a single model to perform the entire task at once. In the constructed framework, one LLM interprets the operator's intent for a specific connectivity requirement and translates it into a structured representation of that intent. The second LLM then generates the corresponding CNI-specific policy from it; the third validates the result and, if needed, corrects it. The fourth writes a Bash script that checks the policy's actual behaviour inside the running cluster. It works with both Cilium and Calico, and is designed so that the operator never has to write the underlying YAML.

Beyond the pipeline itself, a second contribution lies in the adopted evaluation framework. For a generated policy, syntactic correctness or theoretical plausibility is insufficient. The policy must operate reliably when actually enforced. Hence, the pipeline does not terminate at generation; instead, it completes the cycle through live connectivity tests run directly in the cluster. The evaluation uses a test set of one hundred intents per CNI, deliberately composed to range from highly specific directives (complete with port and label selectors) to vague, colloquial descriptions that name only the participating services. Aegis is also compared with three existing tools that address the same problem from different perspectives.

The rest of the thesis follows this structure. Chapter 2 lays out the technical background needed for what follows: Docker, Kubernetes, the Calico and Cilium CNI plugins, intent-based networking, and properties of LLMs relevant to the work. Chapter 3 reviews existing tools aimed at automating Kubernetes network policy generation, including KubeTeus, the system on which this work is based. Chapter 4 describes the overall system architecture and how it extends KubeTeus's original design, Chapter 5 goes into the multi-agent reasoning pipeline, covering how the agents communicate and how the errors are handled. Chapter 6 presents runtime verification, enforcement, and the feedback loop that connects the reasoning pipeline back to the actual cluster state. Chapter 7 describes the evaluation results, comparing Aegis against Kagent, Nimbus, and KubeTeus on different metrics: CNI coverage, human intervention required, token consumption, time needed by each intent, and pass rate. The thesis closes with Chapter 8, which summarizes the main findings, discusses the limitations of the current system, and outlines possible

directions for future work.

# Chapter 2

# Background

This chapter introduces the technical background necessary to understand the work presented in this thesis. It begins with an overview of containerization technologies, namely Docker and Kubernetes, which form the foundation of modern cloud-native application deployment and orchestration. Particular attention is given to Kubernetes' security model, with a focus on network policies as the primary mechanism for controlling traffic between workloads. Since native Kubernetes NetworkPolicies present several limitations in terms of expressiveness and observability, the chapter then examines two of the most widely adopted Container Network Interface (CNI) plugins, Calico and Cilium, comparing their architectures and their extended policy enforcement capabilities. Building on this networking background, the chapter introduces Intent-Based Networking (IBN) as a paradigm for abstracting network configuration complexity through high-level, natural language intents. Finally, it provides an overview of Large Language Models (LLMs), highlighting the architectural principles and capabilities that make them suitable for structured generation tasks. Together, these topics provide the necessary foundation for the pipeline developed in this thesis that translates a single natural language network intent into a valid, enforceable Calico or Cilium policy, without requiring the user to write or understand the underlying YAML.

## 2.1 Docker

Docker [2] is an open-source platform for developing, building, sharing and running containerized applications. It provides a lightweight and portable environment for packaging and executing software across different systems.

Originally developed as an internal project at dotCloud, Docker was a natural extension of the technologies the company had created to support its cloud business. It was designed to address the problem commonly known as dependency hell, where

applications depend on different versions of libraries or software components that may conflict with one another. By packaging an application together with all its dependencies inside a container, Docker ensures that it can run consistently across different environments [3].

Containers run on top of the operating system kernel, shared with the host machine, in an isolated environment. Unlike virtual machines, containers share the host operating system kernel rather than virtualizing the underlying hardware. In this way, they require fewer resources and can be created and started much faster.

In a container-based environment, where applications are virtualized and isolated from one another, Docker provides an additional deployment layer that simplifies the creation, distribution, and execution of containers [4].

Docker is composed of different main components:

- Docker Client
- Docker Server/Daemon
- Docker Images
- Docker Registries
- Docker Containers

Docker follows a client-server architecture. Requests issued by the Docker Client are sent to the Docker Server, which processes them accordingly. The client and the server can run on the same machine or communicate remotely with a Docker Server hosted on a different system. The Docker daemon is responsible for managing all the different Docker objects.

Docker Images are templates used to create containers that contain the application code, dependencies, libraries, and configuration required to run an application. They can be built from existing base images or created from scratch by "committing changes" to an existing container or by using a Dockerfile, which contains the instructions required to build an image (Docker build). Images are stored in

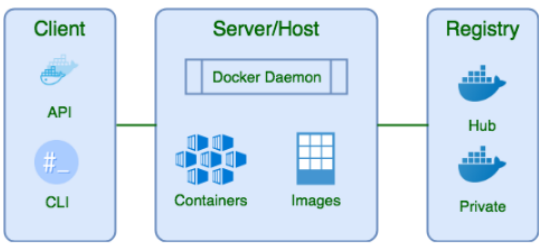



**Figure 2.1:** Docker architecture [5]

Docker Registries, which can be either public or private. The default public registry is Docker Hub, where public images are published and made available to users worldwide.

Finally, Docker Images are used to create Docker Containers. Each container includes all the dependencies required by the application and runs in a lightweight and isolated environment, ensuring consistency across different execution platforms.

Docker also provides networking capabilities that allow containers to communicate with each other. It supports different networking modes, or users can also create custom networks to manage communication between containers. For this purpose, each container can be assigned an IP address.

To manage multi-container applications, Docker provides Docker Compose, which simplifies the orchestration of multiple containers and allows users to define and manage the entire application stack through YAML files [6].

Docker is often used together with Kubernetes. While Docker allows applications to be packaged into portable containers, Kubernetes automates the deployment, scaling, management and self-healing of containerized applications across clusters [6].

## 2.2 Kubernetes

Kubernetes [7], also known as K8s, is an open-source platform for orchestrating containerized applications. It automates the deployment, scaling and management of workloads across a cluster. A Kubernetes cluster is composed of a control plane and a set of worker nodes. The control plane maintains the desired state of the cluster, while the worker nodes provide the computing resources on which workloads are executed. The basic execution units in Kubernetes are the *Pods*. A Pod represents one or more containers that share the same network namespace and can communicate with each other through the loopback interface. Each Pod is assigned an IP address, and communication between Pods is therefore based on the Kubernetes cluster network. Pods are ephemeral resources: over the lifetime of an application, they may be created, deleted, or rescheduled. For this reason, network policies normally refer to workloads through labels rather than relying directly on their IP addresses. Kubernetes resources are defined by declarative manifests, most commonly written in YAML. A manifest describes the desired state of a resource, and Kubernetes continuously reconciles the system to preserve that state. Namespaces, in turn, provide a logical way to separate groups of resources and workloads within the same cluster. Labels provide key-value metadata that can be associated with Kubernetes objects and are particularly relevant to network policies because they allow policies to select groups of Pods without depending on their dynamically assigned IP addresses. The control plane exposes the Kubernetes API

through the *kube-apiserver*. The cluster state is stored in *etcd*, while the scheduler assigns Pods to suitable nodes and controllers continuously reconcile the actual state of the cluster with the desired state [8]. On each worker node, the *kubelet* is responsible for managing the lifecycle of Pods assigned to that node.

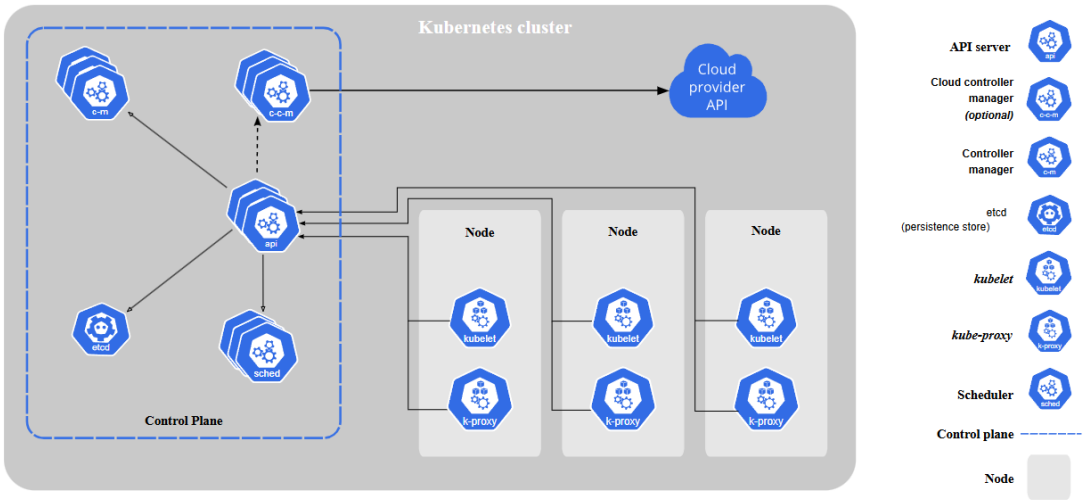



**Figure 2.2:** Kubernetes architecture [9]

Kubernetes follows a client-server architecture (shown in Figure 2.2), where the control plane acts as the central server that manages the cluster and the the worker nodes execute workloads. Each Node runs two main components: the *kubelet*, which receives and manages commands related to container execution, and the *kube-proxy*, which is responsible for configuring network rules and ensuring that traffic is correctly routed to the pods. From a networking perspective, Kubernetes defines a model in which each Pod receives its own IP address and Pods are expected to be able to communicate across nodes without requiring application-level changes. The implementation of this network model is delegated to networking plugins through the Container Network Interface (CNI). A CNI plugin is responsible for configuring the network connectivity required when Pods are created or removed and may also provide additional functionality, including network policy enforcement [10].

### 2.2.1 Kubernetes Network Policies

Kubernetes NetworkPolicy provides a declarative mechanism for controlling network traffic to and from Pods. The NetworkPolicy resource specifies the traffic that is permitted for the selected Pods, while the actual enforcement is delegated to the network plugin installed in the cluster. Kubernetes itself defines the policy API but does not implement the enforcement mechanism. Therefore, the capabilities

and behaviour available in practice depend partly on the CNI implementation used by the cluster [11].

A NetworkPolicy is scoped to a particular namespace and employs a *podSelector* to identify the Pods it governs. When the selector is left empty, the policy applies to every Pod in that namespace. It can regulate inbound and outbound traffic, referred to as *ingress* and *egress* respectively, and these two directions are evaluated separately when determining whether a Pod should be considered isolated. For instance, a policy that selects a group of Pods by a specific label can specify that only traffic originating from another labelled group of Pods is allowed.

The source or destination of traffic can be further restricted using a combination of Pod selectors, namespace selectors and IP blocks. Ports and protocols can also be specified to restrict the allowed traffic to particular transport-layer connections.

A simplified example is shown in Listing 2.1.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend
  namespace: application
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: frontend
      ports:
        - protocol: TCP
          port: 8080
```

**Listing 2.1:** Example of a Kubernetes NetworkPolicy allowing ingress from frontend Pods.

In this example, the policy applies to Pods labelled `app=backend` in the `application` namespace. It allows ingress traffic from Pods labelled `app=frontend` on TCP port 8080. The policy does not directly identify Pods by IP address. Instead, selectors are evaluated against the labels associated with the workloads.

**Network Policy Semantics** The semantics of Kubernetes NetworkPolicy are based on isolation rather than on explicit allow and deny actions. If no

NetworkPolicy applies to a Pod for a particular direction, that Pod is non-isolated in that direction and traffic is allowed by default, subject to any other network controls implemented by the cluster. Once a policy selects a Pod for ingress or egress, the Pod becomes isolated for that direction. Only traffic explicitly allowed by the applicable policies is then permitted [11].

Ingress and egress isolation are independent. A Pod can therefore be isolated for ingress while remaining non-isolated for egress, or the opposite. For example, a policy containing only an `Ingress` type restricts incoming connections without automatically restricting outgoing connections. Similarly, an egress policy does not by itself restrict incoming traffic.

The effects of multiple Kubernetes NetworkPolicies are additive. If several policies select the same Pod for the same direction, the traffic allowed by those policies is combined. The policies therefore do not have an evaluation order in which one policy overrides another. Instead, the effective set of allowed connections is the union of the connections allowed by the applicable policies [11].

This behaviour can be illustrated by considering two policies selecting the same backend Pods. If the first policy allows traffic from Pods labelled `app=frontend` and the second allows traffic from Pods labelled `app=monitoring`, both sources are permitted. Adding the second policy does not revoke the permission introduced by the first one.

A connection between two Pods is also affected by the policies applied to both endpoints. For a connection from a source Pod to a destination Pod, the source's egress policy and the destination's ingress policy can independently restrict the connection. Consequently, an allowed connection requires the relevant egress and ingress controls to permit the traffic [11].

The `policyTypes` field identifies whether a policy applies to ingress, egress or both. If the field is omitted, Kubernetes derives the policy types from the rules contained in the policy. An explicit egress-only policy can therefore be expressed by specifying `Egress` in `policyTypes`. An empty ingress or egress rule list can be used to isolate the selected Pods while allowing no traffic in that direction [11].

The selectors used by NetworkPolicy provide three main ways of identifying peers. A `podSelector` selects Pods according to their labels. A `namespaceSelector` selects namespaces according to their labels. When the two selectors are used together, the resulting peer is restricted to Pods matching the Pod selector in namespaces matching the namespace selector. An `ipBlock` can instead be used to specify an IP address range using CIDR notation [11].

Kubernetes NetworkPolicy operates primarily at the network and transport layers. Rules can restrict traffic according to IP ranges, ports and transport protocols such as TCP, UDP and SCTP. The standard NetworkPolicy API does not provide application-layer rules such as restricting an HTTP request according to its method or URL path. It also does not define an explicit `Deny` action or a

policy ordering mechanism. These characteristics limit the types of network intents that can be represented directly using the native policy model.

These limitations are relevant when a high-level intent requires behaviour that is not directly represented by the Kubernetes NetworkPolicy API. For this reason, network plugins can extend the Kubernetes policy model with additional policy resources and enforcement mechanisms. Two such implementations are Calico and Cilium, discussed in the following sections.

### 2.2.2 Calico

Calico is a networking and network policy solution for Kubernetes and other environments. It integrates with Kubernetes through the CNI and provides both network connectivity and policy enforcement. Its architecture is modular and includes components responsible for networking, policy programming and interaction with the Kubernetes control plane [12].

The main components involved in a Calico deployment are illustrated in Figure 2.3. Felix is the per-node agent responsible for programming routes and access control lists required to provide connectivity and enforce network policy. BIRD can be used to distribute routes between nodes through BGP, while the Calico CNI plugin integrates Calico with the Kubernetes pod lifecycle. The exact set of components depends on the deployment and dataplane configuration.

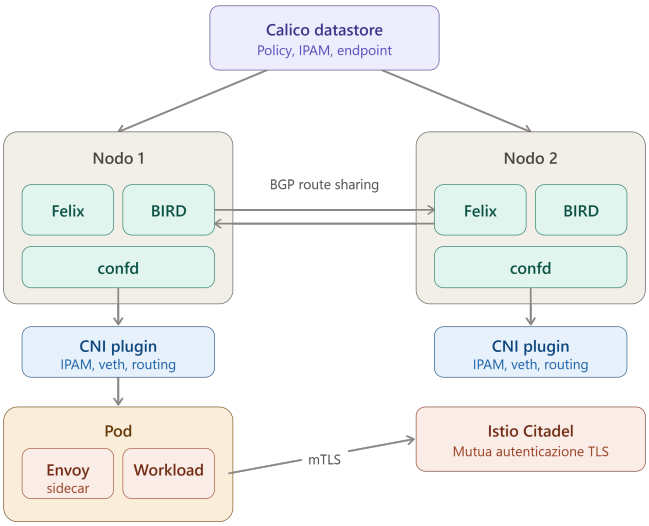



**Figure 2.3:** Overview of the main components of the Calico architecture. The architecture includes components such as Felix, BIRD, confd, the CNI plugin, the datastore, and the Kubernetes controllers.

One of the main components involved in policy enforcement is Felix. Felix runs on the nodes hosting workloads and programs the dataplane with the routes and access control rules required to provide connectivity and enforce policy. Depending on the configured dataplane, Calico can use mechanisms such as Linux iptables or eBPF to implement the required rules [12].

Calico extends the Kubernetes NetworkPolicy model with its own policy resources. The two main resources relevant to workload policy are the namespaced `NetworkPolicy` and the non-namespaced `GlobalNetwork Policy`. A Calico `NetworkPolicy` applies to endpoints within its namespace, while a `GlobalNetwork Policy` can apply across namespaces and can also be used for host endpoints [12].

Unlike the Kubernetes NetworkPolicy resource, a Calico policy can explicitly specify the action to take when a rule matches traffic. The main actions relevant to network filtering are `Allow` and `Deny`. Calico also supports actions such as `Log`, which records matching traffic while allowing policy processing to continue, and `Pass`, which transfers processing to a subsequent policy or profile according to Calico's policy model [12].

Calico policies use selectors to identify the endpoints to which a policy applies. The `selector` field defines the endpoints selected by the policy, while rules can contain selectors and other criteria for identifying the source or destination of traffic. Policies can therefore express relationships between groups of workloads using labels rather than relying only on IP addresses.

The basic structure of a Calico policy is similar to the Kubernetes model, ingress and egress are specified independently. A policy can contain an `ingress` section, an `egress` section, or both. Rules can restrict traffic using protocols, ports, selectors and IP networks. Calico also provides additional matching mechanisms, including namespace and service-account selectors [12].

A simplified Calico policy is shown in Listing 2.2.

```
apiVersion: projectcalico.org/v3
kind: NetworkPolicy
metadata:
  name: allow-frontend
  namespace: application
spec:
  selector: app == "backend"
  types:
    - Ingress
  ingress:
    - action: Allow
      protocol: TCP
      source:
        selector: app == "frontend"
      destination:
```

```
16          ports:
17            - 8080
```

**Listing 2.2:** Example of a Calico NetworkPolicy with an explicit Allow action.

The policy selects backend endpoints and allows TCP traffic from endpoints labelled `app=frontend` to port 8080. The explicit `Allow` action distinguishes this policy language from the native Kubernetes NetworkPolicy model.

Calico also introduces policy ordering. Policies can specify an `order` value, with lower values taking precedence over higher values. This allows the result of a policy set to depend on the order in which policies are evaluated. Ordering is particularly relevant when policies contain both `Allow` and `Deny` actions that could match the same traffic [12].

For Kubernetes Pods, Calico maintains compatibility with the standard Kubernetes NetworkPolicy behaviour when the Kubernetes policy resource is used. In contrast, Calico's own policy resources provide additional semantics through explicit actions and ordering. This distinction is relevant when translating a high-level intent, because the same desired behaviour may require different policy structures depending on whether the target is the Kubernetes NetworkPolicy API or a Calico-specific resource.

Calico also supports policing of endpoint types beyond Kubernetes Pods. Its policy model can be applied to workload endpoints such as Pods and virtual machines, as well as host endpoints. This allows network controls to be applied at a broader scope than the Kubernetes NetworkPolicy abstraction [12].

### 2.2.3 Cilium

Cilium is an open-source networking and security solution for Kubernetes that uses eBPF as a fundamental component of its datapath. eBPF allows programs to be attached to several points of the Linux networking stack, enabling Cilium to implement networking, security and observability functions in the kernel [13]. The main components of Cilium are shown in Figure 2.4. The Cilium agent runs on each node and manages the eBPF programs used to control network traffic. The CNI plugin is invoked by Kubernetes during the lifecycle of a Pod and triggers the required datapath configuration. A Cilium deployment includes a Cilium agent running on each node. The agent receives information about workloads and network configuration and manages the eBPF programs used to implement networking and policy enforcement. The Cilium CNI plugin coordinates the configuration required to provide Pods with network connectivity and policy enforcement [13]. Cilium policies can be expressed using Cilium-specific resources. The two resources most relevant to Cilium's extended policy model are `CiliumNetworkPolicy` and `CiliumClusterwideNetworkPolicy`. The former is namespace-scoped, while the

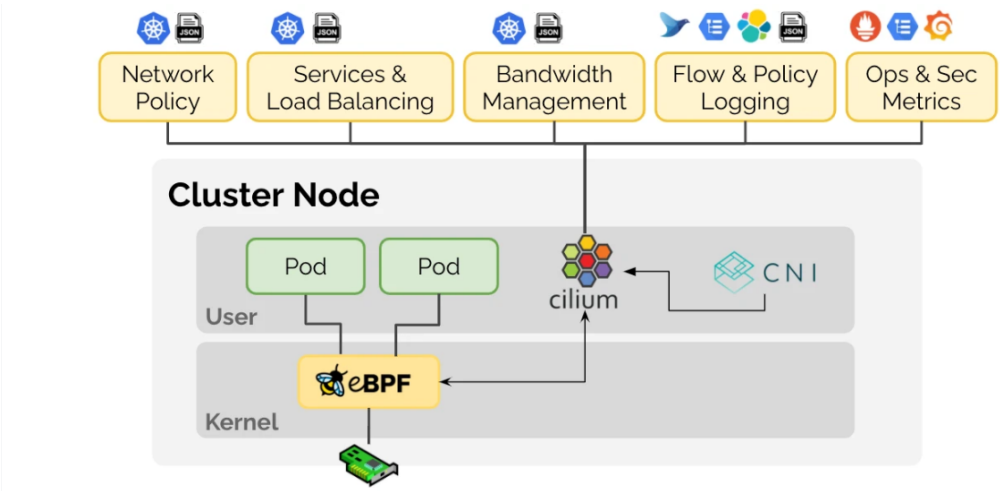



**Figure 2.4:** Overview of the main components of the Cilium architecture.

latter provides cluster-wide policy scope [13]. Cilium uses an identity-based policy model in which endpoints are associated with identities derived from their labels and other Kubernetes information. Policies can therefore select workloads using an `endpointSelector`. This approach allows policy rules to refer to workload characteristics rather than depending directly on dynamically assigned Pod IP addresses. A Cilium policy follows the same basic ingress and egress distinction used by Kubernetes NetworkPolicy, but provides a richer set of matching capabilities. An `endpointSelector` determines the endpoints to which the policy applies, while `ingress` and `egress` rules specify the traffic allowed to or from those endpoints. Rules can identify other endpoints using selectors and can restrict traffic according to ports and protocols [13].

A basic Cilium policy is shown in Listing 2.3.

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: allow-frontend
spec:
  endpointSelector:
    matchLabels:
      app: backend
  ingress:
    - fromEndpoints:
        - matchLabels:
            app: frontend
      toPorts:
        - ports:
            - port: "8080"
              protocol: TCP
```

**Listing 2.3:** Example of a CiliumNetworkPolicy restricting ingress by endpoint and port.

The `endpointSelector` selects the backend endpoints, while `fromEndpoints` restricts the source to endpoints labelled `app=frontend`. The `toPorts` section then restricts the allowed traffic to TCP port 8080.

One of the main extensions provided by Cilium is support for Layer 7 policy rules. Layer 7 rules are associated with Layer 4 port rules and can provide protocol-specific filtering. For HTTP traffic, for example, a policy can restrict requests according to properties such as the HTTP method and URL path. This allows policies to express requirements that cannot be represented by the native Kubernetes NetworkPolicy API [13]. An example of this behaviour is a policy that allows only a specific HTTP operation, shown in Listing 2.4.

```
toPorts:
```

```
  - ports:
      - port: "80"
        protocol: TCP
    rules:
      http:
        - method: GET
          path: "/public"
```

**Listing 2.4:** Example of a Cilium Layer 7 rule restricting HTTP method and path.

This Layer 7 rule can be combined with the Layer 4 `toPorts` rule shown in Listing 2.3, further restricting traffic on port 80 to only `GET` requests targeting the `/public` path. With this type of rule, the policy can distinguish between HTTP requests that use the permitted method and path and other requests sent to the same service port. Cilium therefore provides a policy abstraction that can extend from network-layer controls to application-aware traffic rules [13]. Cilium also provides Hubble, which collects and exposes visibility information derived from the eBPF datapath. Hubble is an observability component built on top of Cilium that provides visibility into network flows and can be used to inspect communication between workloads. This is separate from the policy definition itself, but it is relevant to understanding how Cilium exposes information about the behaviour of an enforced policy [13].

### 2.2.4 From Native Limitations to CNI-Based Enforcement

Kubernetes NetworkPolicy, Calico and Cilium provide overlapping functionality, but they differ substantially in the expressiveness and semantics of their policy models. Kubernetes NetworkPolicy offers a standard abstraction for controlling Pod-to-Pod and Pod-to-network traffic through selectors, ports and transport protocols, but it lacks explicit allow and deny actions, policy ordering and application-layer awareness. Calico extends this model with explicit actions, policy ordering and additional endpoint and selector types, while Cilium extends it further through identity-based endpoint selection and Layer 7 policy capabilities. These differences are directly relevant to the problem addressed by this thesis. A natural-language intent describes a desired behaviour without necessarily specifying how that behaviour should be represented in a particular policy language. The target CNI therefore affects the representation that must be generated: an intent requiring an explicit deny rule can be represented directly using Calico's policy actions, while an intent containing HTTP-specific conditions can make use of Cilium's Layer 7 policy model. For this reason, the policy generation process cannot treat Calico and Cilium as interchangeable YAML formats. Although both operate on Kubernetes workloads and support common concepts such as ingress, egress, selectors and

ports, their policy resources expose different semantics and capabilities. The active CNI consequently forms part of the context required to generate an appropriate policy. The limitations of the native Kubernetes NetworkPolicy model are not merely theoretical: they directly motivate the choice, in this thesis, to target Calico and Cilium policy objects rather than the native `NetworkPolicy` resource. A system that aims to translate a single high-level intent into a complete, enforceable security posture cannot rely on a model that lacks explicit deny rules, ordering semantics or Layer 7 awareness, since generating native manifests would mean discarding, by construction, most of the expressiveness an intent might actually require. For this reason, the pipeline developed in this thesis produces Calico and Cilium policy objects directly, rather than the more limited Kubernetes-native format, and applies them to the running cluster as part of the enforcement stage of the system described in Chapter 6. This design is also what ties the present chapter back to the Intent-Based Networking paradigm introduced in Section 2.4: the user submits a single intent, and the system is responsible for selecting the target CNI representation, generating the corresponding manifest and enforcing it, without requiring the user to author or understand the underlying policy language.

## 2.3 Security Applications

The problem addressed in this thesis goes beyond usability. Network policies constitute a security control, and mistakes in authoring them translate directly into weaknesses in a cluster's security posture. Recent empirical work on the ethical exploitation of misconfigured Kubernetes clusters supports this reading: across container breakout, RBAC misuse, lateral movement and dashboard hijack scenarios, every simulated exploit relied on a known and avoidable misconfiguration rather than on a zero-day vulnerability [1]. In that study, lateral movement between pods was among the attack types most effectively mitigated by network policies, and the authors recommend deny-all defaults with selective exceptions in Calico or Cilium as a priority remediation step. Several security scenarios illustrate the concrete benefits of translating natural-language intents into enforceable Calico or Cilium policies, rather than simply avoiding manual YAML authoring.

**Vulnerability containment.** When a CVE affects a library or an image used by a running workload, the priority is usually to restrict what that workload can reach until a patched version is deployed. An intent such as "isolate the payment-service pod, allow only traffic from the monitoring namespace" can be turned into an enforced restrictive policy in the time it takes to state the requirement, without an operator having to write and validate a CNI-specific manifest under pressure. During an active incident, the speed of containment matters as much as

its precision. The 2018 breach of Tesla's Kubernetes environment [14] shows what such weaknesses can lead to: attackers entered through an exposed administrative console and deployed cryptomining workloads inside the cluster before the exposure was found and closed. No network policy was missing in that case, but the episode shows how an exposed control surface can give an attacker an initial foothold and room to operate.

**Restricting communication paths.** A recurring requirement in production clusters is that workloads should reach only what they explicitly need. The same principle takes different forms depending on the boundary it is applied to. At the cluster edge, internal services handling sensitive data should not be able to open connections to the internet: the intent "no pod in the payments namespace should be able to reach the internet, except the payment-gateway service" corresponds to a default-deny egress policy with a single narrow exception.

Between workloads, the concern is instead lateral movement. Without NetworkPolicies, pods can generally communicate freely, so compromising one is often enough to reach the rest; this pattern appears in Kubernetes-targeted cryptojacking campaigns [15], where malware with initial access to a container scans the internal network for reachable kubelets before spreading. A rule such as "the frontend should never be able to initiate connections to the database directly" narrows the paths available to an attacker who already holds a foothold. At the namespace level, clusters shared between environments or tenants, such as staging and production or several customers in a multi-tenant deployment, require strict boundaries between workloads that must never communicate, expressible in a single general statement such as "pods in the staging namespace must never communicate with pods in the production namespace". In each case the policy is tedious to author by hand, easy to get wrong, and straightforward to express as an intent, which is why the three cases are treated here as one class of requirement rather than as separate ones.

**Misconfiguration as an attack vector.** Many security incidents in cloud-native environments stem from misconfiguration rather than from flaws in the software: network policies that are absent, too permissive, or simply wrong. Red Hat's 2024 State of Kubernetes Security Report [16], a survey of 600 security professionals, found that the large majority of organizations running Kubernetes had experienced at least one security incident in the previous year, with a substantial share traced to detected misconfigurations rather than to vulnerabilities in application code. The pipeline presented here performs semantic validation and runtime testing before a policy is considered final, so a policy is checked not only for syntactic correctness but also for its actual effect on cluster traffic. This reduces the risk of enforcing a policy that appears secure but does not provide the intended protection.

These scenarios point to requirements related to containment, least privilege, segregation, and correct enforcement that are central to running a Kubernetes cluster securely, and that manual policy authoring satisfies only inconsistently. The pipeline in this thesis is aimed at that gap rather than at convenience alone.

## 2.4 Intent Based Networking

Intent-Based Networking (IBN) is an innovative network management paradigm that separates implementation complexity from the management layer. It emerged as modern networks grew in scale and became increasingly dynamic. In this scenario, traditional management based on manual commands and detailed configuration is no longer sustainable. Recent years have seen technologies such as 5G and the Internet of Things foster innovative applications across various industrial domains. Cloud deployments, meanwhile, have become increasingly widespread, requiring greater computing capacity, support for distributed services, and the interactive behavior demanded by these new applications. IBN was proposed by the IETF [17] to address these challenges: it abstracts away the complexity of network management so that users can concentrate on service objectives rather than low-level implementation details. Users express their requirements as intents: high-level objectives that describe the desired network behavior, instead of configuring the network manually. Such intents are typically expressed in natural language, and the IBN system is responsible for translating them into the corresponding network configuration.

A network intent therefore represents a high level of abstraction, where the desired outcome is expressed rather than its implementation, relying on semantic rules instead of low-level commands [18]. Depending on the level of detail, intents can be classified as descriptive or prescriptive. A descriptive intent expresses a general objective, such as “allow traffic between X and Y”, while a prescriptive intent carries more specific information, for instance “from X:10.0.0.2 to Y:10.0.0.1 set rule=allow” [18]. The core idea of IBN, in both cases, is to separate users and applications from technology-specific configuration syntax: the network itself carries out whatever actions are needed to satisfy the stated intent. This higher level of abstraction also benefits network providers, as it allows the system to automate configuration and continuously check whether the declared objective is still satisfied. To reach this, IBN combines automation and orchestration mechanisms with machine learning techniques, moving toward networks capable of managing themselves with minimal human intervention. The intent is defined once by an administrator, typically through a centralized interface, after which the system takes over: it monitors the network continuously and reacts to any deviation from the intended state by applying corrective actions on its own. The

work presented in this thesis follows exactly this loop, applied to Kubernetes network security. A single natural language intent is submitted by the user; an LLM-based pipeline translates it into a Calico or Cilium policy manifest, which is then applied directly to the running cluster. No manual YAML authoring and no manipulation of low-level CNI rules are required at any point. The system is, in this sense, a concrete instantiation of the IBN paradigm described above, narrowed to the specific problem of Kubernetes network policy generation and enforcement.

## 2.5 Large Language Models

The ability to process and generate human language has long been one of the hardest problems in artificial intelligence. Early approaches relied on hand-crafted rules and statistical models that worked reasonably well for narrow tasks but struggled to generalize. The landscape changed fundamentally in 2017 with the introduction of the Transformer architecture [19], which replaced recurrent processing with a mechanism called self-attention. Rather than reading a sequence word by word, a Transformer model attends to all tokens in a sequence simultaneously, learning which parts of the input are most relevant to each other. This parallelism made training on much larger datasets tractable and unlocked a qualitative jump in language understanding. Large Language Models are neural networks built on this architecture and trained on text corpora of unprecedented scale [20]. The defining characteristic is size: billions of parameters, trained on hundreds of billions of tokens drawn from books, websites, code repositories and scientific literature. At this scale, models can develop emergent capabilities that were not explicitly trained for, including analogical thinking and the ability to follow complex instructions expressed in plain language [21].

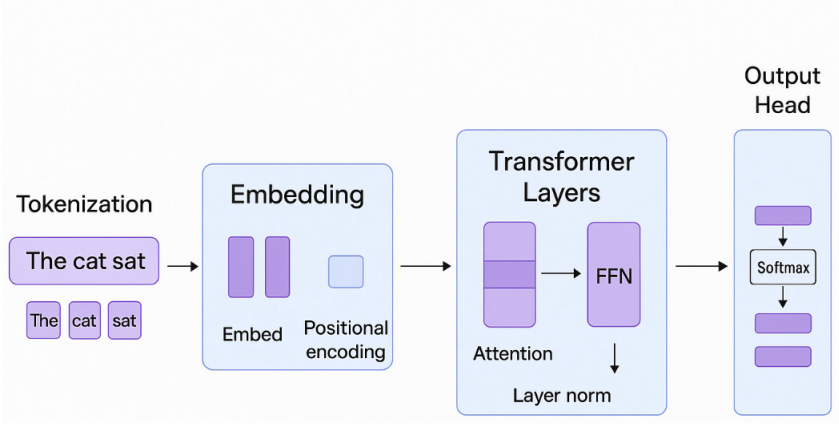



**Figure 2.5:** LLM architecture

From a technical point of view, an LLM takes a sequence of tokens as input and produces a probability distribution over the next token, automatically generating text one token at a time. Despite this relatively simple objective, the representations learned during pre-training capture rich semantic and syntactic structure. As a result, LLMs prove effective not only for open-ended text generation but also for structured output tasks, including the generation of configuration files, code, and formal specifications from natural-language descriptions [22]. Prompting represents the primary interface through which this capability is accessed. Instead of retraining or fine-tuning a model for a specific task, a prompt can steer a pre-trained LLM toward the desired behavior simply by describing the task, providing examples, or constraining the expected output format [23]. This aspect is especially relevant for tasks that require deterministic, machine-readable output: a single, broadly scoped prompt often produces inconsistent results, whereas decomposing a complex generation task into a sequence of smaller, narrowly scoped prompts, each handled by a dedicated call, generally yields more reliable and verifiable outputs [23]. The relevance to this work is direct. The pipeline developed in this thesis relies on LLMs to translate informal natural language intents into syntactically valid and semantically correct Calico and Cilium network policy manifests. This is a structured generation task where LLMs have shown consistent capability: the model must understand the intent, map it to a known schema, and produce output that is both parseable and functionally correct [22]. The multi-model design adopted in this work, using separate models for translation, generation, validation and test script production, reflects the same principle: smaller, focused prompts tend to produce more reliable outputs than a single monolithic call [23].

This reliability concern is not incidental to the design of the pipeline: it is the reason a dedicated validation stage exists at all. An LLM producing a Calico or Cilium manifest is not guaranteed to produce output that is syntactically well-formed, let alone one that enforces the security posture the user actually intended. Schema violations, malformed selectors or subtly incorrect rule ordering are all plausible failure modes of a generative approach applied to a structured, safety-relevant artifact such as a network policy. This motivates treating generation and validation as separate stages in the pipeline, rather than trusting the raw output of a single model call before it is enforced in the live cluster.

# Chapter 3

# Related works

Several approaches have been proposed for translating high-level security intents into enforceable policies in Kubernetes clusters, and they differ quite a bit in how much flexibility they give up in exchange for predictability.

Nimbus [24] sits at one end of that spectrum. It maps predefined identifiers to specific network intents, each of which resolves to a pre-built YAML configuration. Behaviour is fully deterministic as a result, every generated manifest is known in advance but only intents already implemented in the system can be handled.

A different line of work turns to Large Language Models instead, trading some of that determinism for the ability to express requirements far more freely. Kubeteus [25] and the framework in [26] both fall into this category, though they attack the problem from noticeably different angles.

KubeTeus [25] generates Kubernetes NetworkPolicies directly from natural-language specifications, using live cluster information pulled through the Kubernetes API to enrich the user's prompt before an LLM ever sees it. The policy synthesis becomes cluster-aware this way. A fine-tuned LLM handles the interpretation and generation step, after which the result gets validated and enforced. Two modes are supported: one driven by explicit user intents, another driven by whatever resources are already deployed in the cluster.

[26] goes after a related but distinct problem: the general lack of structured annotations in Kubernetes manifests, and what that costs when trying to generate configuration automatically. Their pipeline combines classical NLP with deep learning to pull relevant metadata out of large YAML datasets. Files get preprocessed into structured representations first, which then feed relevance-ranking based on TF-IDF and embeddings. Graph Neural Networks model relationships between configuration elements on top of this, GPT-based models turn those relationships into natural-language descriptions of configuration intent, and a fine-tuned LLM finally produces the YAML manifest matching what the user asked for. The target here is configuration synthesis in general even though not security intent enforcement

specifically, which is where this work diverges from it.

The next sections go into Nimbus, kagent and KubeTeus in more depth. The framework in [26], despite being relevant, is left out of the experimental evaluation simply because no public implementation exists to test against.

## 3.1 Nimbus

Nimbus [24] is an open-source security automation framework built around the idea of separating what a security objective is from how it gets enforced. Kubernetes deployments typically rely on a patchwork of tools for this, admission controllers, network policy engines, runtime security systems, each with its own rule format and its own configuration differences. Nimbus address this fragmentation by allowing users to declare requirements as Kubernetes-native `SecurityIntent` resources rather than writing rules by hand for every engine involved.

### Nimbus Architecture

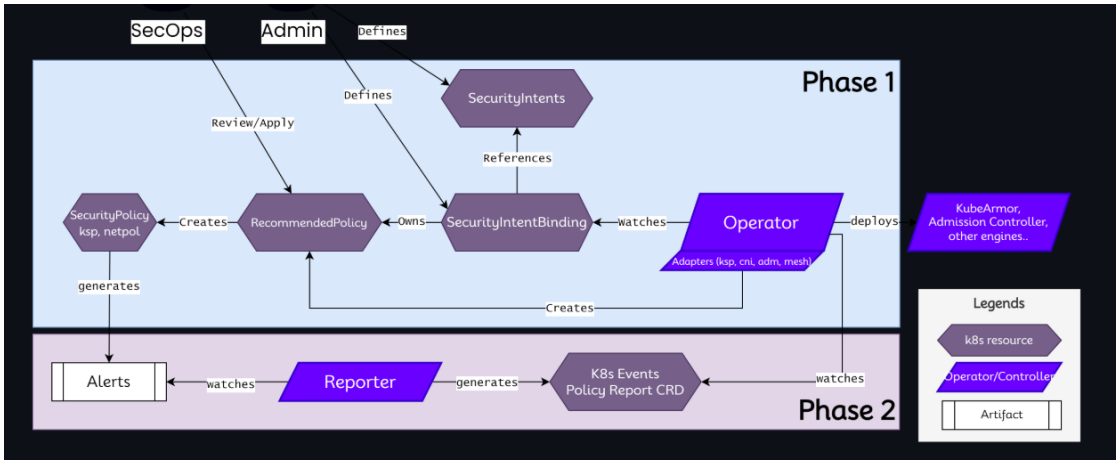



**Figure 3.1:** Architecture of Nimbus [27]

The architecture (Figure 3.1) follows the standard Kubernetes operator pattern, custom resources, controllers and adapters working together to turn a high-level intent into something enforceable.

Users start by defining `SecurityIntent` resources at the top layer. Each reference points to one of a finite set of predefined intent IDs, mapped to a specific enforcement strategy [27]; it does not accept an arbitrary natural-language description. An intent on its own says nothing about scope, so a `SecurityIntentBinding` attaches it to concrete workloads through ordinary Kubernetes label selectors.

Controllers reconcile these resources continuously. One validates and normalizes the intent definitions. Another handles workload selection: it works out exactly which resources a given intent touches. Nimbus then combines this state into an intermediate abstraction, `SecurityPolicy`, whose rules are assembled from each intent's ID, action, and parameters. This representation stays agnostic to any particular enforcement engine. That separation keeps intent processing apart from the mechanics of implementation.

Domain-specific adapters turn this into actual, enforceable objects: KubeArmor, Kyverno, native NetworkPolicy, k8tls. Each adapter checks whether an intent ID falls within its supported set. If it does, the adapter runs a fixed translation routine based on switch-case logic and hardcoded templates. A single intent can therefore expand into several concrete policies across different enforcement layers. That is what enables a coordinated defence-in-depth strategy: admission control, network restrictions, and configuration checks all triggered together. Nimbus also tracks the status of what it generates and updates the fulfilment state accordingly, although this visibility remains fairly coarse compared with true runtime verification.

Two limitations fall directly out of this design. Free-form security intents are simply not something Nimbus can handle; only the finite set of identifiers already wired into its adapters is supported and anything new means implementing and hardcoding a new mapping. And there is no natural-language understanding or runtime inference anywhere in the loop: the link between intent and enforcement policy is fixed at design time, never inferred at runtime.

## 3.2 Kagent

Kagent [28] takes a completely different shape. It is a general-purpose agentic framework for cloud-native environments. It was originally built at Solo.io, it is now a CNCF sandbox project. Kagent provides a platform for running autonomous agents inside a Kubernetes cluster rather than implementing one predefined intent-to-policy pipeline.

Three abstractions structure the framework: *tools*, *agents* and the *framework layer*. Tools are MCP-style functions exposing cluster capabilities to an agent, such as retrieving pod logs, querying Prometheus metrics or generating Kubernetes resources. Agents are autonomous components that combine one or more tools with iterative planning to work through multi-step operational tasks, ranging from diagnosing connectivity issues to managing progressive rollouts. They can be organized into teams, in which a planning agent breaks down a high-level task and assigns its parts to specialized agents; furthermore, they can identify and communicate directly with one another via an Agent-to-Agent (A2A) protocol. The framework layer itself, built on Google's Agent Development Kit, exposes agents

as first-class Kubernetes custom resources, definable and controllable through ordinary GitOps workflows, and supports both Go and Python runtimes alongside third-party orchestration tools like LangGraph and CrewAI.

There's an obvious point of contact with the intent-driven systems already discussed: kagent is also about turning high-level goals into concrete cluster actions, and like KubeTeus it can pull live cluster state into its reasoning through the tool layer. But its scope is far wider and correspondingly less specialised. There is no dedicated pipeline here for network policy synthesis, validation, conflict resolution or automated testing and rollback the way KubeTeus implements them. Such a workflow would need to be built as a custom agent or tool on top of kagent's generic planning and tool-invocation machinery, rather than relying on anything pre-built. kagent's agents are, by design, meant for open-ended reasoning across very different operational domains (observability, traffic management, security and so on) while KubeTeus stays deliberately narrow.

Two agents in kagent's current catalogue matter directly for the comparison later in this thesis. `kagent/cilium-policy-agent` generates a CiliumNetworkPolicy manifest given a natural-language intent. However it never queries the target cluster and has no way of resolving pod labels, namespaces or existing services on its own. Actually enforcing and checking that manifest falls to a separate agent, `kagent/k8s-agent`, which does have direct API access and can apply and inspect cluster resources. Getting from intent to an enforced, verified policy therefore means orchestrating two independent agents rather than following a single pipeline. Since the generation agent has no cluster awareness at all, any ambiguity in the original intent is left for the human operator to sort out.

## 3.3 KubeTeus

KubeTeus [25] is an intent-to-policy generation framework for Kubernetes presented at IEEE INFOCOM 2025. It targets a fundamental usability gap: translating high-level security objectives into network policies that are both syntactically correct and semantically meaningful without requiring the user to know Kubernetes internals or CNI-specific rule formats.

Where Nimbus is rule-based, KubeTeus takes arbitrary natural language, such as *"deny egress traffic from the frontend service"* and produces an enforceable policy tailored to the CNI plugin installed in the cluster.

### 3.3.1 KubeTeus Architecture

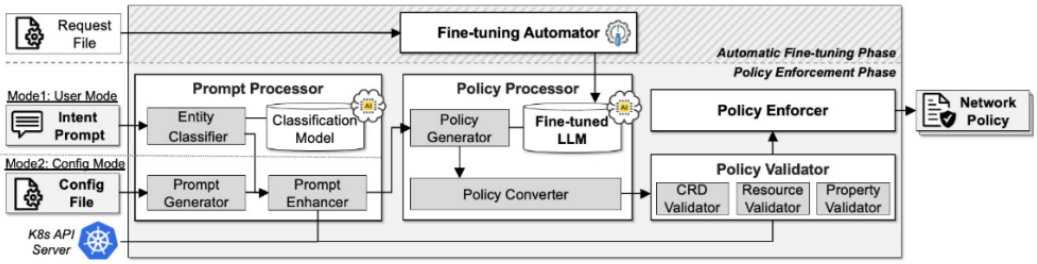



**Figure 3.2:** Architecture of KubeTeus [25]

KubeTeus follows a multi-stage pipeline architecture (Figure 3.2), where each stage progressively transforms a natural-language intent into a validated and applied Kubernetes policy.

The first stage is the **Entity Classifier**, a fine-tuned BERT-based token classification model that reads the raw intent string and extracts semantics entities such as pod names, actions (`allow`, `deny`), directions (`ingress`, `egress`), ports, protocols, and namespace references.

In the second stage, the **Prompt Processor** is implemented as a Kubernetes controller component. It enriches these entities with live cluster data, such as pod labels, container ports, associated services, environment variables, and the active CNI, all retrieved through the Kubernetes API. The result is a detailed and grounded prompt for the language model.

That prompt goes to a **Large Language Model**, served through a locally deployed Flask instance, which generates the network policy in YAML. The original implementation used fine-tuned CodeGemma and DeepSeek Coder variants, trained specifically on Kubernetes network policy data to improve generation accuracy.

A **policy converter** sits between generation and validation, turning the LLM's raw output into a typed, CNI-specific representation. For instance, this representation can be a native Go `CiliumNetworkPolicy` object, with its `endpointSelector`, `ingress`, and `egress` fields populated, before being sent to the validation phase.

Validation itself runs in several steps: a CRD syntax check against the target CNI schema, a resource existence check confirming that the referenced Pods and services actually exist in the cluster, and a property check verifying ports and protocols against live workload configuration.

Only once all of these checks pass does the **Enforcer** component apply the policy to the cluster through the Kubernetes API.

# Chapter 4

# System Architecture

Aegis follows a two-layer architecture that separates the control logic, interacting with the cluster, from the reasoning, based on language models. The first layer, implemented as a Kubernetes controller, receives user intents, observes the cluster state, orchestrates their processing and enforces the policies generated by the reasoning layer. The second layer is an external service that translates natural-language intents into network policies and associated verification scripts. This separation is intentional. The reasoning component relies on language models that may be computationally expensive and may require different inference backends, while the Kubernetes controller should remain lightweight and reactive. Moving the reasoning logic outside the controller makes the two components relatively independent: the reasoning backend can be changed without redesigning the cluster-facing logic, while the controller remains responsible for the parts of the workflow that require direct access to Kubernetes. The interaction between the two layers is driven by HTTP requests. Incoming intents are accepted by the controller and placed into a bounded queue, where they are processed sequentially. Before an intent is sent to the reasoning service, the controller enriches it with information obtained from the cluster, including the CNI plugin and information about the running Pods. The reasoning service then transforms the intent through a sequence of specialised language-model stages. Once a policy has been generated and validated, it is returned to the controller, which is responsible for reconciling it with the current cluster state, enforcing it and subsequently verifying its runtime behaviour.

Thus, the architecture is not merely a translator from natural language to YAML; it forms a closed-loop workflow. The cluster state is observed before reasoning, a candidate policy is generated and validated, the resulting configuration is applied to the cluster, its actual behaviour is tested and, if necessary, reverted. Figure 4.1 gives an overview of this workflow. The internal structure of the reasoning stage, the details of enforcement and rollback are presented in the following chapters.

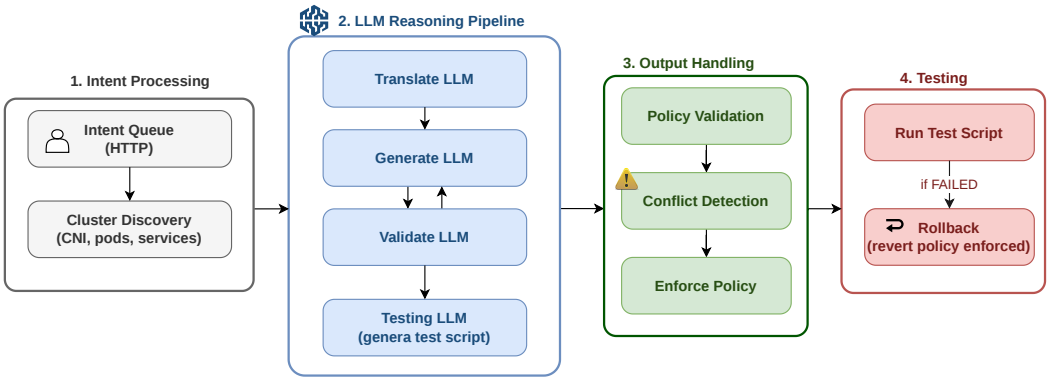



**Figure 4.1:** High-level architecture: from intent submission to enforcement and rollback.

## 4.1 Architectural Design

The architecture is informed by limitations observed in the original system and by the operational requirements imposed by live network policies.

A primary design choice concerns request processing. Incoming intents are enqueued in a bounded queue and processed sequentially by a dedicated worker. This arrangement separates request reception from execution, while serial processing ensures that policy changes occur in a predictable order. This predictability is crucial because policy generation and enforcement are not purely computational operations, as they ultimately mutate the state of a running cluster. The concurrent processing of multiple intents would complicate the determination of which policy was active at any given time and could lead to interference among independent requests.

The second decision relates to the boundary between the language-model reasoning and the policy enforcement. The original KubeTeus architecture included an intermediate policy conversion step between model generation and enforcement. During development, this conversion was found to be lossy, particularly for certain ingress and egress information. Consequently, the policy converter was removed. The reasoning service now generates the policy directly in the dialect required by the active CNI plugin, and the resulting YAML is passed to validation and enforcement without an intermediate transformation into another policy representation.

A third consideration is the decomposition of the reasoning process into specialised stages. Rather than relying on a single language model to interpret an intent, generate a policy, validate it and produce a test script in one monolithic step, these responsibilities were separated into distinct stages. This decomposition facilitates inspection of each stage's output and provides explicit points for feedback

when a generated policy is rejected. The internal organisation of this pipeline is detailed in Chapter 5.

The fourth decision is to ground the reasoning process in information collected from the actual cluster. That labels were not assumed, namespaces, CNI configuration, or command-line tools are available merely because they appear plausible from the intent. The controller discovers these properties and supplies them as context to the reasoning service. This is particularly important for runtime testing, where a syntactically correct script is not useful if the selected container does not contain the command required to execute it.

Finally, feedback was incorporated during reasoning and after enforcement. A policy rejected by validation can be returned to the generation stage for correction. A policy that passes the reasoning stages but fails its runtime test can be rolled back automatically. The model's output is therefore treated as a candidate throughout the system, not as an authoritative decision.

## 4.2 Main Components

The architecture comprises several components that cooperate to process security intents from submission to runtime verification. The following subsections describe these components and their responsibilities, covering cluster discovery, request handling, context enrichment, language-model-based reasoning, policy validation, policy enforcement and feedback. Together, these components implement the closed-loop workflow introduced above.

### 4.2.1 Cluster Discovery Layer

The cluster discovery layer supplies the reasoning service with information about the environment in which the generated policy will operate. The controller observes running workloads and collects data including pod labels, exposed ports, available networking tools and the active CNI plugin. The CNI type is particularly critical because it determines the policy dialect expected by the generation and validation stages. The reasoning service therefore receives the detected CNI as part of the contextual information associated with each request.

The discovery layer also maintains information about the tools available inside running Pods. At discovery time, the controller probes the containers for relevant networking utilities and caches the results in memory. The cache is refreshed periodically; an on-demand refresh can be triggered when the information is stale at the time an intent arrives. This prevents the reasoning service from having to rely on assumptions about the software installed inside individual containers. In essence, this component acts as an information provider. It does not decide which

policy should be generated; rather, it establishes the environmental context against which subsequent reasoning and testing are performed.

### 4.2.2 Intent Queue and Request Handling

User intents enter the system through an HTTP endpoint and are placed into a bounded `IntentQueue`. A dedicated worker consumes the queue and invokes the intent-processing logic sequentially. This mechanism is the main entry point for policy requests. Although the controller is integrated with the Kubernetes controller-runtime framework, the operational intent-processing path is driven by the HTTP endpoint and the queue, not by a reconciliation event. Thus, the `Reconcile` method does not constitute the primary execution mechanism of the system as it was in Kubeteus.

The queue also establishes a clear separation between accepting a request and processing it. An HTTP request does not directly execute the reasoning pipeline; it is first converted into a queued unit of work, allowing the controller to maintain a controlled execution order.

### 4.2.3 Cluster Context Enrichment

Before an intent reaches the reasoning service, the controller enriches it with information from the discovery layer. The resulting request includes both user-provided data and contextual information about the cluster. This enrichment is necessary because natural language alone is often insufficient to determine an executable policy. An intent may refer to a generic entity such as “the backend” or “the database”, whereas the actual cluster may contain several applications with different labels. Similarly, an intent typically does not specify which networking utilities are installed in a source container. Therefore, cluster context is treated as an explicit input to reasoning rather than as information that the language model is expected to infer. The quality of the resulting policy still depends on the model’s interpretation of the request, but that interpretation is constrained by information describing the environment in which the policy will be applied.

### 4.2.4 Reasoning Service

Once an intent has been enriched with cluster context, it is handed to the reasoning service, which produces both a candidate network policy and the script that will later be used to verify the policy. The service is organised as a sequence of specialised language-model stages for intent translation, policy generation, policy validation and test-script generation. Generation and validation are connected by a bounded feedback loop: a rejected candidate is not discarded but returned to the

generation stage together with the validator's feedback, for up to three attempts. If all attempts are exhausted, the pipeline proceeds with the best candidate obtained within the retry budget, regardless of the validation outcome.

At the level of detail relevant to this chapter, the reasoning service was treated as a single component: it receives an enriched intent and cluster context and returns a policy together with a test script, or an indication that no usable result could be produced within the retry budget. The internal organisation of the four stages, the shared state that connects them, the CNI-specific prompting strategy, and the choice of language models and inference providers are described in full in Chapter 5.

### 4.2.5 Enforcement and Feedback Layer

The reasoning service does not directly determine the final state of the cluster. Once a candidate policy and its test script have been produced, responsibility returns to the controller, which reconciles the candidate with the cluster's current state, applies it, and subsequently checks whether its observed effect matches the requested behaviour.

This stage completes the feedback loop introduced at the beginning of this chapter. Before enforcement, the controller checks the target namespace for policies that could conflict with the new one and removes them. The policy is then applied and its associated test script is executed against the live cluster. If the test indicates that the enforced behaviour does not match the intent, the controller removes the policy through an automatic rollback, avoiding the persistence of a potentially incorrect configuration. Thus, a policy can be accepted by the reasoning service and still fail at runtime if its actual effect diverges from the intent. The complete mechanics of conflict resolution, enforcement, testing, and rollback are treated separately in Chapter 6.

## 4.3 Comparison with the Original Architecture

The original KubeTeus architecture followed a considerably flatter processing model. An intent was translated and used to generate a policy, which was then passed through a policy conversion stage before enforcement. Semantic checking relied more heavily on static rules and the system had limited awareness of the actual cluster state. Moreover, there was no runtime mechanism capable of determining whether an enforced policy produced the behaviour requested by the user.

The architecture presented in this chapter introduces several changes to that model.

First, the reasoning process is explicitly grounded in the cluster. Pod labels, available tools, ports, namespaces and the active CNI plugin are treated as contextual inputs rather than assumptions made by the language model.

Second, the intermediate policy converter has been removed. The language model generates the CNI-specific policy directly, and the raw YAML is preserved through validation and enforcement. This reduces the number of transformations between reasoning and execution and avoids the loss of rule information observed in the original conversion step.

Third, the single-shot reasoning process has been replaced by a sequence of specialised stages. Natural-language intent is first translated into a structured representation, a policy is then generated, and the generated policy is subjected to a bounded validation-and-correction loop. The testing stage produces the executable verification script only after this process.

Fourth, the system no longer ignores policies already present in the cluster. Conflict detection is performed before enforcement so that a newly generated policy is not evaluated in isolation from residual configuration.

Finally, the architecture extends beyond generation and enforcement by introducing runtime verification and rollback. The generated policy is applied to the live cluster and its effect is subsequently tested. If the observed behaviour does not satisfy the expected outcome, the controller can remove the policy rather than leaving a potentially incorrect configuration active.

Taken together, these changes transform the original system from a relatively direct intent-to-policy translator into a closed-loop architecture. The language model remains central to interpreting the user's request and constructing the policy, but it operates within a larger system that provides environmental context, validates its output, controls how that output reaches the cluster, and ultimately checks whether the resulting behaviour is consistent with the original intent.

# Chapter 5

# Multi-Agent Reasoning Pipeline

This chapter goes inside the reasoning component introduced previously, laying out how a natural-language intent is transformed, step by step, into a network policy and a runtime verification script. The focus here is on how the language-model agents are organised, what state they share, and how the feedback mechanism refines a generated policy that does not meet the requirements.

The pipeline runs as a directed graph built with LangGraph. Rather than a fixed sequence of independent function calls, the graph models the actual transitions possible between stages. The graph consists of four nodes: `translate`, `generate`, `validate` and `testing`. Translation always runs first, then generation, then validation, which is the only stage that can redirect execution: if the policy fails and the retry budget still has room, control goes back to generation; otherwise it moves on to testing.

Figure 5.1 shows the resulting control flow.

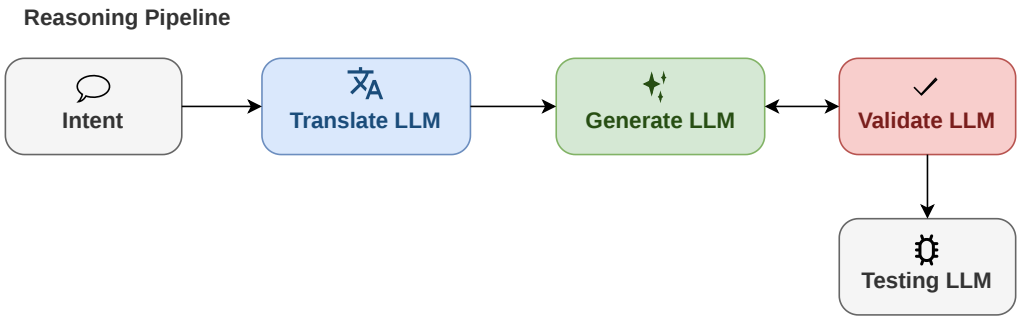



**Figure 5.1:** Directed graph of the reasoning pipeline.

Nothing here runs in parallel by design. Testing never overlaps with generation or validation; it only starts once the generation-validation loop has actually terminated, which guarantees the test artefact is produced only after the policy has either been accepted or run out of correction attempts. The rest of the chapter works through the same pipeline at three levels: graph structure and shared state, what each agent is responsible for, and the implementation choices behind the whole thing.

## 5.1 Architecture

This section describes how the reasoning pipeline is organised as a directed graph and how information is maintained throughout its execution. It first introduces the structure of the graph and its conditional transition, then describes the shared state used by all nodes and finally explains how the bounded retry mechanism controls the generation-validation loop.

### 5.1.1 Graph Structure

The reasoning graph is deliberately small. The nodes correspond to distinct responsibilities:

- `translate` reads the natural-language intent and produces a structured technical representation;
- `generate` builds a candidate network policy from the translated intent and cluster context;
- `validate` checks the generated policy against the requested behaviour and decides whether it should be accept or correct;
- `testing` writes the script the controller will later use to verify the policy at runtime.

The entry point of the graph is the translation node. A direct edge runs from translation to generation, another from generation to validation. The only conditional branch in the whole graph sits on the edge leaving validation, governed by the `should_continue` function: it returns either `generate` (another generation pass) or `testing` once the loop is complete. In code, this is represented as follows:

```
graph.set_entry_point("translate")
graph.add_edge("translate", "generate")
graph.add_edge("generate", "validate")
```

```
graph.add_conditional_edges(
"validate",
should_continue,
{"generate": "generate", "testing": "testing"}
)

graph.add_edge("testing", END)
```

There is only one feedback edge, placed exactly where the system has enough information to judge whether a policy needs revision. Translation does not get repeated on a failed validation, the semantic reading of the intent has not changed just because one particular candidate was wrong. Testing, likewise, sits outside the correction loop entirely; it only touches the candidate once generation and validation have completed.

### 5.1.2 Shared State

All agents operate on a common state represented by the `PolicyState`. The state holds both what came in with the original request and whatever intermediate results the pipeline has produced so far: the original intent, the translator's structured representation, the current policy candidate, validation results, how many generation attempts have happened and the cluster context the controller collected:

- `intent`: the original natural-language request;
- `technical_intent`: the structured interpretation of the request;
- `yaml`: the current generated policy;
- `validation`: the result of the latest validation step;
- `attempt`: the number of policy-generation attempts performed;
- `cluster_context`: information about the current cluster environment.

The state acts as the single source of information throughout the graph. Rather than passing individual values between functions through an increasing number of arguments, every node receives the shared state and updates only the fields relevant to its own operation. This is what makes the feedback loop simple to implement: when a policy gets rejected, the generation node receives the existing state with the original intent, the technical representation, the previous YAML, the validation feedback and the cluster context all still sitting there. The next attempt fixes the existing candidate instead of starting from scratch.

The use of explicit state fields also provides some protection against malformed model output. LLMs may return explanatory prose where structured data was expected, incomplete YAML, or an empty response. Keeping intermediate artefacts in well-defined fields allows the implementation to detect missing or unusable values before they propagate into the next stage.

### 5.1.3 Retry Mechanism

An attempt counter in the shared state bounds the generation-validation loop. When `should_continue` sees a failed validation but room left in the attempt budget, it routes back to `generate`; once validation passes, or the budget runs out, execution moves on to `testing`. That's a hard guarantee against runaway model calls; the generator and the validator disagreeing indefinitely is simply not possible here. A rejected candidate doesn't get thrown away. It might already have correct selectors, ports, namespace details – regenerating everything from scratch would waste that. So the previous YAML and the validator's feedback both carry into the next attempt, letting the model the existing candidate rather than generate a new one from scratch. Running out of retries without an accepted candidate isn't treated as a hard failure inside the graph – the pipeline just proceeds to testing with whatever the best candidate was within budget. It's a pragmatic trade-off: more iterations mean more chances to fix a bad policy, but also more latency and token cost, and there's no guarantee extra rounds converge anyway.

## 5.2 Agents

The following subsections describe the four agents that form the reasoning pipeline. Each agent has a specific responsibility, all working on the same shared `PolicyState` but with different purpose within the overall workflow. What follows follows the execution order: intent interpretation, generation, validation and finally producing the runtime test.

### 5.2.1 Intent Translation

Translation is the entry point of the reasoning process. It receives the original natural-language intent and converts it into a structured representation that the rest of the pipeline can actually use.

This stage was introduced to separate natural-language interpretation from policy generation. Asking the policy generator to infer the meaning of an informal request while simultaneously deciding the appropriate Kubernetes policy structure makes it more difficult to determine whether an error originates from semantic

interpretation or policy generation. The translation stage provides an intermediate representation between these two tasks.

What it produces includes the source application, the target, port, protocol, action, namespace, policy type, plus a fuller natural-language description of what's actually wanted.

For example, an intent such as:

*block traffic from the frontend to the database on port 5432*

is transformed into a representation that identifies the frontend as the source, the database as target, port `5432`, and `deny` as the intended action.

A more ambiguous intent, such as,

*the frontend shouldn't be reachable from outside*

requires the translator to infer that the relevant direction is ingress and that the source should be interpreted broadly, rather than referring to a single named application.

The distinction between source and target is particularly important for the later testing stage. The entity a policy protects is not necessarily where the test connection should originate. For an ingress-deny request, the target is the object being protected, while the source is where the test traffic needs to come from.

The translator is instructed to return JSON rather than a free-form paragraph. This change was introduced because an earlier version returned plain text and required subsequent stages to recover fields with regular expressions, which broke easily whenever an intent had exceptions or did not follow the expected sentence shape.

Structured output provides downstream nodes explicit fields to work with, while still leaving room for a natural-language description wherever something cannot be reduced to the fixed schema. The implementation does not assume that the model will always comply with the requested format. If the model returns invalid JSON or surrounds the structured output with extra text, a fallback parser, based on the older pattern-matching approach, attempts to recover the relevant information. The fallback is limited to recognisable things like explicit namespaces, application labels, port numbers, and common allow/deny phrasing.

Whether translation went through the model or the fallback, the resulting internal representation looks the same either way, so the other agents do not need to know how the translation was obtained.

### 5.2.2 Policy Generation

The generation agent receives the translated intent, the cluster context, and the CNI the controller detected, and is responsible for producing an actual policy

from them. The prompt used here is specific to whichever CNI is active, Cilium and Calico are given separate instructions, since the resource types and semantics exposed are different. A single generic prompt turned out to be unreliable. The model could produce a syntactically acceptable policy in the wrong dialect or combine concepts from the two systems. The CNI is therefore an explicit part of the generation context, and the model writes the policy directly in the dialect expected by the active cluster, avoiding an intermediate conversion layer, used in earlier versions of the system.

The generation has two operating modes. On the first attempt, the model receives the technical intent and the available cluster context and is asked to produce a policy that satisfies the requested behaviour. If validation rejects that, generation runs again, this time with the previous YAML and the validation result already sitting in state, so the model is asked to fix the existing candidate rather than start over.

The attempt counter, tracked explicitly in shared state, is what the function `should_continue` checks to keep the loop from running forever.

Even though the model is instructed to use the namespace associated with the request, the system does not rely entirely on that instruction. After generation of YAML, the namespace is set programmatically on every generated document. This is a deliberate separation between what the model decides semantically and what gets enforced deterministically. The model determines the structure and the content of the policy, while properties that can be guaranteed by the controller are applied in code instead.

The same logic applies to extract the YAML out of the response. Generation is expected to return a single policy document, and implementation strips away any surrounding explanatory text before submitting the result to later stages.

### 5.2.3 Policy Validation

The validation agent checks the current policy candidate against the original intent and the cluster context. Validation is also CNI-specific, since judging whether a policy is semantically right requires understanding the policy language itself, not just checking that the YAML is syntactically valid. Thus, it receives the same CNI information passed to the generator.

The validator is asked to determine whether the policy actually represents the requested behaviour. This includes checking selectors, directions, ports, actions and how the policy relates to the applications named in the intent. Its output decides the next step in the graph. If validation passes, execution moves to testing; if it fails, execution loops back to generation, assuming the retry limit has not been reached.

The translation does not run again here for the same reason mentioned earlier.

The semantic reading of the intent does not change because one candidate was wrong. Similarly, the testing stage is not entered until the generation-validation branch has finished.

Validation also produces the information needed for a correction. When a candidate fails, the feedback gets passed along with the previous YAML into the next generation attempt. In this way, the generator has both the original request and the concrete reason for rejecting the policy.

### 5.2.4 Testing

The testing agent is the last node in the graph. It is responsible for writing a Bash script that checks the network behaviour represented by the intent. The agent receives the structured intent and cluster context rather than the generated YAML itself. The purpose of the test is to verify the expected behaviour described by the user, not just simply reproducing whatever the policy generator happened to produce. Cluster context contains information about networking utilities available in the source Pods. The agent is explicitly instructed to use an available tools rather than assume a common command like `curl` exists.

Tool selection logic follows a fixed order of preference: `nc` or `ncat` first, then Bash-based `/dev/tcp`, then HTTP-oriented tools like `wget` or `curl`, and other available tools as a fallback.

This information comes from the discovery layer covered in the previous chapter, so the reasoning service never has to probe the cluster itself. The testing stage also needs to select a concrete source pod when the intent refers to a generic source such as "any". The implementation uses cluster context to pick a reasonable candidate rather than leaving it up to the model. The selected pod satisfies the relevant constraints the intent requires and must have an appropriate networking tool.

This matters especially for UDP. A pod that can do TCP connectivity checks is not automatically fit for a UDP test, so tool availability is checked for the specific protocol during selection. If no suitable pod or tool can be found, the test is supposed to report that verification is not available rather than run something unreliable. A failed test and a test that could not be executed do not carry the same meaning.

## 5.3 Implementation Details

The rest of this chapter covers the implementation choices that support the reasoning workflow described above. The subsections describe how security-policy semantics are represented in the prompts and how different kinds of "deny" behaviour are distinguished, how the system contains model failures the mechanism used to

contain them and how models and providers were assigned to each of the four stages during evaluation.

### 5.3.1 Policy Semantics

The generation and validation prompts contain explicit guidance for several semantic distinctions that cannot be handled by syntax alone.

A distinction is made between a specific deny, an implicit deny, and a total deny. Consider an intent stating that "only the frontend should reach the cache", the behaviour can be represented with a single allow rule for the frontend, relying on the policy semantics to deny traffic that is not explicitly allowed. An intent saying "the backend specifically must *not* reach the cache" means something different. The desired behaviour is that one connection needs to be blocked while everything else stays as it was. A generated policy should distinguish the explicit deny from the remaining traffic rather than converting the request into a complete block.

Finally, an intent such as "nobody should be able to reach the cache" requires an unconditional deny, adding any catch-all allow here would contradict the requested behaviour.

Distinguishing between these cases at the prompt level, rather than relying on the validator to catch the difference after generation, was found to reduce the number of correction cycles needed before a policy passed validation. This distinction is particularly relevant for Calico, where the placement of the rules and the relationship between allow and deny actions directly shape the outcome. During development, additional prompt constraints were introduced for cases involving bidirectional communication, port placement, and deny/allow combinations. None of this guarantees correctness, but they reduce the number of semantically ambiguous candidates that reach the validation stage.

### 5.3.2 Handling Model Failures

The reasoning pipeline assumes, from the beginning, that model output will not always be well-formed. Failures can occur at every level: malformed JSON from the translator, incomplete YAML from the generator, an unusable response from the validator. Most of these failures are handled locally. Translation falls back to pattern extraction, generation retries when nothing usable comes out, and a failed validation just becomes part of the graph state, which can trigger another generation attempt. This approach avoids tying the controller to the internal details of individual model failures. From the controller perspective, the reasoning service either hands back a policy and a test script, or reports that it could not produce anything usable. The internal graph determines how many attempts are made before reaching that outcome.

The retry limit also provides a practical termination guarantee. A disagreement between the model stages cannot result in a correction loop, the number of API calls is bounded as the attempt count is explicitly tracked.

### 5.3.3 Model and Provider Configuration

The agents are implemented independently of a single language-model provider. A model can be assigned to each stage through the client configuration, allowing the reasoning workload to be distributed across different inference backends. This turned out to matter in practice, once rate limits started showing up from running several stages against the same provider at once. The final setup used various models and providers for different roles. Table 5.1 summarises the models, together with their parameter counts and the inference backend through which each was accessed.

**Table 5.1:** Model configuration used in the different pipeline stages.

| **Stage** | **Model** | **Parameters** | **Provider** |
|---|---|---|---|
| Translation | `llama-3.3-70b-versatile` | 70B (dense) | Groq |
| Generation | `minimax-m2.5:cloud` | 230B total / 10B active (MoE) | Ollama Cloud |
| Validation | `qwen/qwen3-32b` | 32B (dense) | Groq |
| Test Generation | `qwen-coder-next` | 80B total / 3B active (MoE) | Ollama Cloud |

The four models differ considerably in scale and architecture, and this heterogeneity is relevant to the practical implementation of the pipeline. The two MoE-based models, used for policy generation and test-script production, expose a comparatively small number of active parameters per token despite their larger total parameter counts. This made them suitable for the corresponding workloads through Ollama Cloud. The two dense models used for translation and validation were accessed through Groq. None of this is meant to claim any one model is intrinsically the right fit for its stage. Instead, it was the result of the models and inference providers available during development and evaluation, as well as a qualitative preference for a code-driven model for test generation.

The model assignment is therefore a configuration choice rather than a fundamental property into the architecture. What remains stable is the division of responsibilities between stages and the state interface through which they communicate. That separation is also what makes experimentation easier. A different model

can be assigned to an individual stage without modifying the graph structure or the controller interface.

## 5.4 Relationship with the Controller

The reasoning graph is deliberately isolated from Kubernetes-specific enforcement. It receives the information needed to reason about the cluster and returns the artefacts required by the next stage, but it does not itself apply the generated policy.

The controller remains responsible for operations that require direct access to the Kubernetes API, including conflict detection, policy application, runtime execution of the generated test script and rollback. This separation is particularly useful for runtime verification. The testing agent generates a script based on expected behaviour and available tools, but it does not execute the script itself. Execution takes place only after the policy has been applied to the real cluster.

The boundary therefore separates two forms of responsibility:

- the reasoning service figures out *what should be enforced* and *how it can be tested*;
- the controller decides *whether and how those artefacts are applied to the live cluster*.

Keeping this split intact means the language-model component is never the one directly modifying the cluster state.

The next chapter moves outside the reasoning graph and describes how the controller takes these generated artefacts and uses them to modify, test, and, when necessary, restore the state of the Kubernetes cluster.

# Chapter 6

# Runtime Verification, Enforcement and Feedback Loop

The preceding chapter concluded with the generation of a validated candidate policy and a verification script. However, at that point, no modification has yet been applied to the cluster. The present chapter addresses the subsequent stages of the workflow: how the policy is applied, how its behaviour is verified at runtime and how the system reacts when the observed behaviour deviates from the user's original intent.

This stage is of central importance because a syntactically valid and semantically plausible policy remains only a necessary condition for correctness. Such a policy may still conflict with the existing cluster configuration, or may exhibit unintended effects when exposed to the actual runtime environment. A second layer of verification is therefore introduced here: the generated configuration is applied to the live cluster, and its effects are observed directly rather than inferred from static analysis. Intents are processed sequentially, one at a time. Requests are enqueued and handled in order, which prevents concurrency-related interference among conflict detection, enforcement, testing, and rollback operations, all of which modify the same policy state.

The chapter first describes the controller's execution model and its integration with the Kubernetes runtime, then traces the complete path of a single intent from reasoning output through enforcement, verification, rollback and outcome recording.

## 6.1 Controller Architecture and Execution Model

This section describes how the controller handles execution once the reasoning service has produced its output. It starts with the request-processing model and its relationship to controller-runtime, followed by initialisation and cache management, both of which are relevant because the operations described later interact directly with the live cluster.

Intents arrive via a `POST /intent` endpoint on port 9090. They are placed into a bounded queue with a capacity of fifty entries and are consumed sequentially by a dedicated worker. Each intent is processed under a ten-minute context timeout: this duration is sufficient for several rounds of the generation-validation loop and the subsequent runtime test, yet short enough to prevent a stalled request from blocking the queue indefinitely. Sequential processing is particularly important here. Conflict detection, enforcement, testing and rollback all operate on live network configuration. The handling of one intent at a time ensures that concurrent operations do not interfere with overlapping policies and provides the feedback mechanism with a stable and predictable execution order.

### 6.1.1 Controller-Runtime Integration

The controller is built on controller-runtime, though the main intent-processing path does not rely on Kubernetes reconciliation events. In the original design, the standard `Reconcile` loop served as the nominal entry point, reachable via a `SetIntent` method. In practice, however, this method was never invoked by the path actually used to process intents.

The actual processing flow proceeds through the HTTP endpoint, the bounded queue and the worker described above. The `Reconcile` method has consequently been reduced to a no-operation implementation, retained only because `ctrl.NewControllerManagedBy` requires a type that satisfies the reconciler interface. It is worth making this explicit, as it clarifies how the system actually operates: despite being built on controller-runtime, intent processing is not triggered by changes to any Kubernetes resource. Work enters the system through an explicit HTTP request and is subsequently handled by the controller's own queue and worker.

### 6.1.2 Initialization and Cache Management

The controller maintains a cache of information discovered from running pods, as described in the previous chapter. During development, an issue was identified in `SetupWithManager`: the cache refresh ran synchronously at setup time, before controller-runtime's internal cache had started. This produced the runtime error

`the cache is not started, can not read objects`; a timing-dependent failure that occurred specifically when the refresh was triggered before the manager had completed initialisation. The issue was resolved by wrapping the cache refresh and its periodic ticker inside a `manager.RunnableFunc`, registered via `mgr.Add`. Execution now waits until the manager has started and the cache is safe to access. The discovery mechanism itself remains unchanged; this modification ensures that it interacts with the controller-runtime cache only when the runtime is ready.

## 6.2 From Reasoning Output to Cluster State

Once the reasoning service returns a policy and, where applicable, a test script, control returns to `processIntent` on the controller side. This function orchestrates all the steps covered in this chapter: policy validation, conflict resolution, enforcement, runtime testing, rollback and outcome recording, all executed sequentially for each intent.

The generation phase is additionally protected by a controller side retry mechanism, independent of the internal generation-validation loop. Up to three attempts are made, with a two second interval between them, in case the reasoning service fails to return a usable result. Timeouts, empty responses and transient malformed JSON are treated as recoverable rather than fatal errors. If all three attempts fail, or if none produces at least one usable network-policy document, the controller halts processing and records the failure without modifying the cluster. This constitutes an additional safety barrier between reasoning and enforcement: no policy is applied unless reasoning has produced a concrete enforceable artifact.

### 6.2.1 Direct Policy Application

A significant departure from the original system concerns how generated policies are applied to the cluster. The earlier implementation routed all policies through a `policyconverter`, translating model output into typed Go structures before enforcement. This conversion proved to be lossy for certain policy structures. In particular, more elaborate Calico rules combining source and destination selectors with port constraints did not always survive the transformation intact.

The converter has therefore been removed. Raw YAML produced by the reasoning service is sent directly to the Kubernetes API through the client, with no intermediate representation. This approach is feasible because the generation stage is already configured for the specific CNI in use. The model writes the policy directly in the expected dialect and validation operates on the same representation that is subsequently submitted for enforcement.

Eliminating the conversion layer not only simplifies the pipeline but also removes a class of errors in which the correct model results suffered a loss of information

during the transfer to the cluster.

### 6.2.2 Policy Validation

Before a generated policy is applied to the cluster, the controller performs a validation stage to verify both the structure of the generated resource and its consistency with the current cluster state. This stage is executed after policy generation and before conflict detection and enforcement. The validation process is coordinated by the `Validator` function and consists of CRD validation, resource validation and property validation.

The validation pipeline opens with the `CRDValidator`. Its first task is to normalise each generated policy into a uniform `map[string]interface` representation, regardless of whether the input arrives as a raw YAML string or a pre-structured object. From there, the validator inspects the policy's kind field to determine which JSON schema applies. `CiliumNetworkPolicy` resources are checked against the Cilium schema, whereas `NetworkPolicy` resources require a closer look at their apiVersion. When the Calico API is detected, validation switches to the Calico schema instead. Any policy kind that the validator does not support, along with any Kubernetes NetworkPolicy that falls outside the recognised API versions, is rejected at this initial filter stage.

If the CRD validation succeeds, the controller proceeds with the `ResourceValidator`. This component verifies that the generated policy is consistent with resources currently available in the cluster. It first checks the presence and validity of the main resource fields, including `kind`, `metadata` and `spec`. The remaining checks depend on the policy type. For Cilium policies, the validator extracts the labels contained in the `endpointSelector` and uses them to identify the pods targeted by the policy. For standard Kubernetes NetworkPolicy resources, the corresponding labels are obtained from the `podSelector`. Calico policies are handled differently because their selectors are expressed as Calico selector strings rather than Kubernetes label maps. For this reason, label-based pod validation is skipped for Calico policies.

The validator also checks the namespace specified in the policy. It verifies that the namespace exists and that its status is valid. When labels are available for the policy selector, it additionally checks whether pods matching those labels exist in the target namespace. This provides a consistency check between the generated policy and the workloads currently present in the cluster. If no labels are available, the pod existence check is skipped. For Cilium policies, an additional check is performed on service references. The validator verifies the services associated with the policy against the services available in the corresponding namespace. This allows invalid service references to be detected before the policy reaches the enforcement stage.

Initially, after the Resource validation, the `PropertyValidator` was invoked.

Currently the property specifications of both Calico and Cilium are handled in the CRD stage. Thus there are no additional checks in this component.

An important aspect of the validation pipeline is its ordering. The `Validator` first performs the CRD checks for all generated policies. If these checks produce errors, the function returns immediately and the resource validation stage is not executed. If the generated policies pass this first stage, `ResourceValidator` is executed sequentially, with errors from both components accumulated and returned to the controller.

Overall, the validation stage provides a boundary between policy generation and enforcement. It combines schema-level checks with checks against the current cluster state and, where applicable, validation of policy-specific properties. Policies that produce validation errors are not passed to the subsequent conflict detection and enforcement stages, reducing the risk of applying malformed policies or policies that reference resources inconsistent with the current cluster state.

### 6.2.3 Conflict Detection

Before a generated policy is applied, the controller checks whether existing policies in the target namespace may interfere with the policy being introduced. This step is necessary because policies left by previous operations may still affect the same workloads and traffic paths. Simply applying the new policy without considering the existing configuration could therefore produce behaviour that differs from the one intended by the current request.

The conflict detection is performed by `DetectAndResolveConflicts`, which runs after validation and before enforcement. The function first parses the policies generated for the current request and retrieves the policies already present in the target namespace for the CNI identified earlier in the `processIntent`. Both the new and existing policies are converted into a common internal representation containing the target pod selector, ingress and egress rules, peer selectors, actions and ports. This allows the comparison to be performed independently of the different policy formats used by Cilium and Calico. A policy is considered relevant to the comparison only when it targets the same set of pods as the newly generated policy. For this purpose, the controller compares the label selectors of the two policies. The selectors must contain the same key-value pairs. The policies that target different workloads are therefore not treated as conflicting. Once the target is the same, the controller compares ingress rules with other ingress rules and egress rules with other egress rules. The comparison between individual rules considers both the selected peers and the ports affected by the rule. Two rules are considered to overlap when their peer selectors refer to the same set of peers, or when one of them applies to any peer. The same principle is used for ports: the rules overlap when they contain at least one common port and protocol, or when one of the rules

does not specify any port restriction and therefore applies to all ports. Rules that do not overlap in either their peers or their ports are not considered conflicting.

Two cases are distinguished once overlapping rules have been identified. The first is a *residual policy*. This occurs when the new and existing rules apply to the same direction, target the same workloads and overlapping peers and have the same action. In this situation, the existing policy is not contradictory to the new one, but it represents an older rule enforcing the same behaviour. It is removed to prevent the previous policy from remaining active alongside the newly generated configuration. The second case is a *direct conflict*, which occurs when the overlapping rules specify opposite actions. For example, an existing `deny` rule affecting the same traffic targeted by a new `allow` rule represents a direct conflict. Keeping both policies active could make the resulting behaviour dependent on the policy semantics and ordering of the specific CNI rather than on the current intent.

When a residual policy or a direct conflict is detected, the existing policy is removed before the new policy is enforced. The controller records the policy name, namespace, CNI type, policy kind and reason for the removal in a `ConflictResolution` object. This information is returned to the caller and can therefore be included in the controller's execution records, making the conflict-resolution step traceable rather than treating policy deletion as an opaque operation. The two CNI implementations require an additional distinction. For Cilium, the conflict detector can explicitly compare both allow and deny rules because the parser handles `ingress`, `egress`, `ingressDeny` and `egressDeny`. For Calico, the same comparison is performed using the corresponding ingress and egress rules and their explicit actions. In addition, Calico policies may overlap without constituting a direct conflict that can be resolved simply by deleting one policy. To handle this case, Aegis can assign a lower `order` value to the generated policy, giving it higher priority in the Calico policy evaluation. This mechanism is used for overlapping cases that remain after the explicit conflict-removal step, such as a broader residual deny-all policy. The deletion operation is also treated according to the desired final state rather than only according to the immediate API response. If a policy that has been identified as conflicting is already absent when the deletion is attempted, the relevant condition is still satisfied; the conflicting policy is no longer present in the namespace. This makes the conflict-resolution step tolerant of cases in which a previous operation has already removed the policy.

Overall, conflict detection provides a separation between the newly generated configuration and policies left by earlier operations. The controller does not assume that the target namespace contains only policies generated by the current request. Instead, it examines the existing configuration, identifies policies that affect the same workloads and overlapping traffic, and removes or prioritises them before enforcement. This reduces the possibility that residual configuration changes the behaviour being tested during the subsequent runtime verification.

No attempt is made to merge or reconcile conflicting rules. The newly generated policy simply takes precedence, under the assumption that it reflects the most recent user intent. This keeps conflict resolution deliberately simple, avoiding the introduction of an additional composition layer between reasoning and enforcement.

### 6.2.4 Enforcement

Once conflict detection has completed, the generated policies are passed to `enforcer.Enforcer` for application to the cluster. The enforcer operates directly on the generated YAML rather than converting it into an intermediate typed representation. This preserves the policy produced by the reasoning service and avoids introducing another transformation between validation and enforcement. The enforcer accepts both individual policies and collections of policy documents. It also supports multi-document YAML, where several Kubernetes resources are contained in the same string and separated by `---`. Each document is parsed into an `unstructured.Unstructured` object and applied through the Kubernetes client.

Policy application handles both creation and update. The enforcer first attempts to create the resource. If a policy with the same name already exists, it retrieves the existing object, copies its `resourceVersion` to the generated object, and performs an update instead. This allows the controller to replace an existing policy without requiring a separate conversion or resource-specific implementation for each supported CNI. Successful application only establishes that the Kubernetes API accepted the policy. It does not provide evidence that the resulting network behaviour matches the original intent. Runtime verification is therefore performed as a separate step after enforcement.

### 6.2.5 Runtime Testing

When a test script is provided, the controller executes it against the live cluster via `tester.RunTestScript` immediately after enforcement. The test script is written to a temporary file and executed with `bash`. Its combined output is retained together with any execution error. The purpose of this step is different from the validation performed by the reasoning pipeline. The reasoning agent evaluates whether the generated policy appears consistent with the structured intent, whereas the runtime test observes the behaviour produced by the policy after it has been applied to the actual cluster.

The test result is classified into three categories:

- **Passed**: the script executes successfully and reports the expected connectivity behaviour, whether the intent requires a connection to be allowed or blocked;

- **Failed**: the test executes but reports behaviour that is inconsistent with the requested intent;
- **Skipped**: meaningful verification cannot be performed, for example because the selected source pod does not provide a suitable networking tool or because the required source or target resource cannot be found.

The distinction between failed and skipped outcomes is important for the interpretation of the experimental results. A failed test provides evidence that the enforced policy did not produce the expected behaviour. A skipped test does not provide evidence about policy correctness; instead, it indicates that the test environment did not allow the intended check to be performed. Treating both cases as failures would therefore mix policy errors with limitations of the verification environment.

The tester also handles the case in which connectivity tools such as `curl`, `nc` or `wget` are unavailable in the selected container. These cases are treated as skipped rather than failed, allowing the policy to remain active when no meaningful runtime observation could be obtained.

### 6.2.6 Rollback

A failed runtime test triggers an automatic rollback through `enforcer.Rollback`. The rollback operates on the same YAML documents that were applied and removes the corresponding Kubernetes resources from the cluster. Multi-document policies are split into their individual resources before deletion. The rollback mechanism is intentionally separate from the test itself. The tester determines whether the observed behaviour is consistent with the intent, while the enforcer is responsible for restoring the cluster when the test indicates that the enforced policy should not remain active. A policy that is already absent during rollback is not considered an error. The relevant final state is that the policy is no longer present in the cluster, so an `IsNotFound` response is treated as a successful removal. Other deletion errors are propagated and recorded.

Rollback is therefore not assumed to be infallible. If an error occurs while removing a policy, the controller records the rollback failure together with the original test outcome. This makes the failure visible rather than treating the unsuccessful cleanup as if it had completed normally. In such a case, manual intervention may still be required to restore the expected cluster state.

### 6.2.7 Recording Outcomes

After runtime testing and, when necessary, rollback, the controller creates a `TestResultRecord` containing the information associated with the processed intent.

The record includes the original natural-language intent, the generated YAML documents, the test script, the raw test output, any execution error, the classified outcome, rollback information and a timestamp. `LogTestResult` persists these records to separate files for passed, failed and skipped executions. Keeping the three categories separate makes it possible to distinguish successful verification from failed verification and from cases where verification could not be performed.

The stored record is deliberately more detailed than a simple pass or fail value. In addition to the final outcome, it preserves the artefacts that led to that outcome, including the generated policy and the test script. The raw test output is also retained, making it possible to inspect the reason for a failure or to identify cases in which the test environment affected the result. This information is useful both during development and during the experimental evaluation. During development, the records provide a persistent trace of the policy and test execution without requiring the same intent to be executed again against the cluster. When an intent does not pass the test, the generated YAML and the script serve as diagnostic artefacts. The operator may use them to locate the failure and subsequently enforce the policy directly, or apply corrective modifications to the existing configuration, without requiring the generation of a new policy from scratch. For the evaluation, the same records provide the basis for calculating success, failure and skip rates and for inspecting individual cases when analysing the results.

## 6.3 Discussion

The enforcement and verification mechanisms described in this chapter do not replace the reasoning pipeline. They operate after policy generation and provide a controlled path between the model's output and the live Kubernetes cluster. Their purpose is not to make the language model more accurate, but to limit the consequences of an incorrect or incomplete policy once it reaches the enforcement stage.

The separation of responsibilities is therefore deliberate. The reasoning service determines how the user's intent should be translated into a network policy and how the resulting behaviour can be tested. The controller, instead, performs the operations that directly affect cluster state: it checks for conflicting existing policies, applies the generated YAML, executes the test script, and removes the policy when runtime verification fails.

This distinction also separates two different forms of validation. The validation stage in the reasoning pipeline evaluates the candidate policy using the structured intent and the information available to the model. Runtime testing takes place only after enforcement and evaluates the behaviour produced by that policy in the actual cluster. A policy may therefore pass reasoning-time validation and still

fail its runtime test if its interpretation of the intent does not correspond to the behaviour observed in the environment.

The same principle applies to conflict detection. A generated policy may be internally consistent while still interacting unexpectedly with policies left by previous operations. Detecting and removing relevant residual or conflicting policies before enforcement reduces this source of interference and makes the subsequent runtime test more representative of the policy being evaluated.

Rollback provides the final safety mechanism in this sequence. When runtime verification identifies a failed outcome, the policy is removed rather than being left active in the cluster. Persistent result logging then records both the observed behaviour and the cleanup attempt, preserving information about what happened even when the execution does not complete successfully.

Taken together, these mechanisms turn the final part of the system into a controlled transition from a model-generated candidate to an observed cluster state. The model remains responsible for the flexible interpretation of natural-language intents, while enforcement, testing and rollback are handled by deterministic controller-side operations. This separation does not eliminate the possibility of incorrect policy generation, but it provides explicit checks around the point where such errors could otherwise affect the live cluster.

The next chapter presents the experimental evaluation of the resulting system, examining both the effectiveness of the generated policies and the end-to-end behaviour of the intent-processing pipeline.

# Chapter 7

# Evaluation

This chapter evaluates Aegis and the other tools presented in Chapter 3, in order to assess and compare how effectively the proposed pipeline translates natural-language security intents into enforceable network policies. The evaluation is performed on Kubernetes clusters using both Cilium and Calico as CNIs. It considers different levels of intent specificity, from detailed requests that explicitly define technical constraints to more generic requests that leave part of the intended behaviour unspecified.

The evaluation considers several aspects of the system. First, it measures whether the tools can correctly generate and enforce the expected policies across different types of intents. Second, it analyses the amount of human intervention required during the process, since one of the main goals of Aegis is to handle natural-language requests without requiring the operator to manually translate them into technical policy specifications. Finally, time to enforce and token consumption are considered as an indication of the computational cost associated with the language-model-based reasoning process.

The evaluation is divided into two complementary parts. The first part analyses Aegis independently, using a larger set of intents to characterise its behaviour across different levels of specificity and for both supported CNIs. The second part compares Aegis with existing approaches, namely Kagent, Nimbus and KubeTeus. This comparison focuses on the capabilities that are relevant to the problem addressed by this thesis, including policy generation, support for different networking technologies, human intervention, time and successful execution of the resulting policies.

## 7.1 Kagent

Kagent does not provide a single component dedicated to the complete task considered in this thesis. Instead, it provides a collection of agents, each associated

with a specific capability. The `kagent/cilium-policy-agent` is the component relevant to this evaluation. It accepts a natural-language security intent and produces a CiliumNetworkPolicy manifest. The agent does not use information from the target cluster during policy generation, it does not enforce the generated policy itself, nor does it perform runtime verification of its effect. These operations can instead be performed through another agent provided by the framework.

The lack of cluster context can lead to problems when the intent refers to applications using names that do not correspond directly to their Kubernetes labels. For example, asking the agent to *"Allow the frontend to access the redis cache"* produces a syntactically valid CiliumNetworkPolicy selecting a pod labelled `app: redis`, if no further confirmation is given by the operator. The cluster used in this evaluation does not contain such a pod. The cache workload is labelled `app == "cache"`. Since the policy-generation agent does not inspect the cluster, it has no information with which to resolve this difference. The generated policy is therefore consistent with the wording of the request but not with the actual configuration of the target cluster. This distinction is relevant when comparing Kagent with Aegis, since Aegis can obtain information about the running cluster before generating the policy.

In kagent, therefore, human interaction becomes necessary when the intent does not contain enough information to construct the required policy. If the target is described using a generic expression such as "the cache", or if the namespace or port is not specified, the `cilium-policy-agent` can request clarification before continuing. The operator is responsible for providing or confirming information that cannot be obtained from the agent itself. A similar issue arises when bidirectional communication is required. To obtain both the egress rule on the source application and the corresponding ingress rule on the destination, the user has to explicitly specify that communication is required in both directions. The agent does not infer this requirement from the original request.

Once a policy has been generated and, where necessary, corrected by the user, enforcement can be delegated to the separate `kagent/k8s-agent`. This agent can apply the generated manifest and interact with the cluster to inspect the resulting configuration.

The complete Kagent workflow therefore involves more than the policy-generation agent alone. Moving from a natural-language request to an enforced configuration can require coordination between different agents and one or more human interventions.

## 7.2 Nimbus

Nimbus follows a substantially different approach from the language-model-based systems considered in this evaluation. Rather than interpreting an open-ended natural-language request, it operates on a fixed vocabulary of predefined intent identifiers. Each identifier is associated with a hand-written mapping to one or more enforcement adapters.

In this workflow, the user does not specify the required behaviour via natural language. Instead, the input consists of a `SecurityIntent` and a `SecurityIntentBinding`, which collectively denote one of the intent types pre-implemented by Nimbus. These intents are stored as files in a dedicated directory of the tool. The user needs to apply a given intent by pasting its corresponding input into the terminal. The corresponding policy is then produced through the predefined mapping, without an inference step.

**Table 7.1:** Coverage of the evaluated intents across the supported policy mechanisms.

| Intent ID | KubeArmor | NetPol | Kyverno | k8tls |
|---|---|---|---|---|
| `dnsManipulation` | ✓ | ✓ | | |
| `escapeToHost` | ✓ | | ✓ | |
| `swDeploymentTools` | ✓ | | | |
| `unAuthorizedSaTokenAccess` | ✓ | | | |
| `preventExecutionFromLogsFolders` | ✓ | | | |
| `denyExternalNetworkAccess` | | ✓ | | |
| `cocoWorkload` | | | ✓ | |
| `virtualPatch` | | | ✓ | |
| `assessTLS` | | | | ✓ |
| `disallowChRoot` | ✓ | | | |
| `disallowCapabilities` | ✓ | | | |

Table 7.1 lists the eleven identifiers currently implemented and the adapters to which they are routed. KubeArmor is used for runtime controls, Kyverno operates at the admission-control layer, k8tls provides TLS assessment and the NetPol adapter generates standard Kubernetes `NetworkPolicy` objects.

Because this evaluation focuses on network-policy generation, only the identifiers routed through the NetPol adapter are relevant to the comparison. These are `dnsManipulation` and `denyExternalNetworkAccess`. For instance, applying

`dnsManipulation` end-to-end produced both a `KubeArmorPolicy` restricting writes to `/etc/resolv.conf` and a `NetworkPolicy` constraining DNS egress to `kube-dns`. The resulting configuration behaved as intended.

No language-model inference is involved in this process. The corresponding configuration is generated from predefined mappings. This gives Nimbus predictable behaviour for the security properties that are explicitly represented in its vocabulary.

This determinism is one of Nimbus's main strengths. An implemented identifier produces the same configuration independently of model sampling or prompt formulation. At the same time, the same design decision limits the range of requests that the system can process. Nimbus cannot generate a policy for an intent that is not represented by one of its predefined identifiers.

A request that does not correspond to one of the implemented identifiers is therefore outside the supported scope rather than being interpreted approximately. Extending this scope requires adding new intent definitions and the corresponding adapter logic. Nimbus trades the flexibility of open-ended intent interpretation for a fixed and predictable set of supported security behaviours.

## 7.3 KubeTeus

Evaluating the original KubeTeus implementation required several adaptations because some of the components used in the published configuration are not publicly available. The original configuration relies on fine-tuned models hosted in private Hugging Face repositories, including `cclabadmin/codegemma-7b-it-network`. Since these models could not be accessed, a substitute model was provided through the same command-line option exposed by the original tool. This change affected model selection only, while the structure of the remaining pipeline was left unchanged.

A second limitation concerns the first stage of the KubeTeus pipeline. Its original design uses a BERT-based entity classifier to extract structured information such as action, direction, pod names, ports and protocols from the input intent. The public repository does not include the training data, training scripts or trained weights for this classifier, although the project documentation indicates that these components should be available. Reimplementing the classifier was not attempted because doing so would have introduced a new component and would therefore no longer constitute an evaluation of the original implementation. Instead, a manually constructed JSON object matching the documented output schema of the classifier was supplied directly to the remaining pipeline. This allowed the downstream stages to be evaluated while keeping the unavailable natural-language-to-entity extraction stage outside the experiment. The comparison with Aegis at this stage is consequently limited, since the two systems were not evaluated under exactly

the same conditions.

The available implementation also revealed a structural problem in the policy-processing pipeline. The `policyconverter` module produces an object based on Cilium's native schema, including fields such as `endpointSelector`, `ingress`, and `egress`. The downstream `PropertyValidator`, however, expects a different representation based on fields such as `selector` and `rules`.

The conversion function `toMap` confirms that this difference is introduced between the two stages rather than being caused by a particular model output. As a result, a generated policy could not pass this validation step using the available implementation. The corresponding check was disabled for the evaluation, while the validators related to namespace and resource existence remained active.

Even after this modification, enforcement was not consistently successful. Several generated policies were rejected by the Kubernetes API server, while other manifests lacked fields required by the Kubernetes schema. Across a sample of ten intents with different levels of complexity, four reached enforcement without a blocking error.

This result should be interpreted as a qualitative observation rather than as a statistically robust success rate. The sample is small, one validation stage had to be disabled and the model output is stochastic. Repeating the same ten intents could therefore produce a different result. These limitations are considered again when comparing the four systems on the common batch of ten intents.

## 7.4 Aegis

The evaluation of Aegis was performed under controlled conditions using the Cilium and Calico clusters described in the previous chapters. One hundred intents were executed twice: once against a Cilium cluster and once against a Calico cluster. Runtime outcomes were classified automatically by the testing mechanism described in Chapter 6.

The intents were organised according to the amount of technical information explicitly contained in their wording. Three groups were considered:

- **Specific** intents state the port, protocol, namespace and exact label;
- **Intermediate** intents identify the applications involved but leave some technical details to be inferred;
- **Generic** intents use informal or general descriptions of the applications and provide little technical information;

This classification makes it possible to examine not only whether Aegis succeeds, but also how the amount of information provided by the user affects the behaviour

of the pipeline. In particular, it allows token consumption and pass rate to be considered together rather than as independent measurements.

**Specific** intents achieved a pass rate of 100% on both Cilium and Calico and required the lowest token consumption among the three intent groups. These results are consistent with the information contained in the requests. When the port, protocol, namespace, and label are already specified, the translation stage has fewer values to infer and the generation stage has less ambiguity to resolve.

**Intermediate** intents achieved a pass rate of 98% on both Cilium and Calico. Their token consumption was slightly higher than for specific intents, but remained below that of generic intents. In this case, the translation and discovery stages become more important, as references such as ”the cache” or ”the backend” have to be associated with concrete cluster information before the policy can be generated. One intent in the Intermediate batch was skipped during the evaluation because one of the selected pods did not have any available tool for executing the generated test. The generated YAML was subsequently inspected manually and found to be correct. The intent was therefore considered a successful result.

**Generic** intents produced the lowest pass rates and the highest token consumption. The pass rate was 95% on Cilium and 93% on Calico, corresponding to 2 failures out of 40 intents on Cilium and 3 failures out of 40 on Calico. One additional intent was skipped for the same reason observed in the Intermediate batch, namely the absence of an available testing tool in one of the selected pods. Manual inspection confirmed, also in this case, that the generated configuration was correct, and the intent was considered successful.

Overall, the pass rate was 97% when considering only intents for which the runtime test could be executed. When the skipped intents are included in the evaluation as cases requiring manual verification, the overall pass rate decreases to 96%.

The difference between the three groups follows the way Aegis processes the request. Less information is provided by the user in the generic case, so more information has to be inferred from the cluster context and from the wording of the request. This increases the work performed by the translation and generation stages and can also result in additional validation attempts.

The failures in this group do not necessarily have a single cause. In some cases, the cluster may contain several plausible candidates for an informal reference such as ”the database”. In other cases, the wording itself may be too ambiguous for the translator to identify the intended relationship correctly. The discovery layer can provide additional information about the cluster, but it cannot remove ambiguity that is already present in the user’s request.

The results therefore show a relationship between input specificity, token consumption and pass rate. More explicit intents are generally easier to process and require fewer tokens. Generic intents require more reasoning and show a small

but measurable decrease in runtime success. This behaviour is relevant to the comparison with Kagent, where a similar lack of information is instead handled through interaction with the human operator. The comparison between Cilium and Calico provides an additional observation. The same ordering is maintained for both CNIs: specific intents achieve the highest pass rate and lowest token consumption, generic intents achieve the lowest pass rate and highest token consumption, and intermediate intents remain between the two. This suggests that the observed trend is primarily related to the information contained in the intent rather than to a single policy CNI. However, the absolute results differ slightly between the two clusters. During the Calico experiments, asynchronous propagation through the Felix agent occasionally resulted in a test being executed before the newly applied policy had reached the dataplane. In these cases, the policy was subsequently verified to be correct independently, and the transient propagation issue was not counted as a policy-generation failure.

This distinction is relevant because the evaluation aims to measure the ability of Aegis to generate and enforce the intended security behaviour, rather than the propagation delay of the underlying networking implementation. The observed Calico cases therefore affected the timing of the test rather than the correctness of the generated policy.

## 7.5 Evaluation Overview

The remaining sections compare Aegis, Kagent, Nimbus and KubeTeus along four dimensions: CNI coverage, human intervention, token consumption and pass rate. These measurements do not describe exactly the same aspect of the systems, so they are considered separately rather than combined into a single score.

Whenever the interface of the evaluated tool supports it, the specificity level of the original intent is used as a common reference point. For this evaluation, the same three groups used for Aegis are adopted: specific, intermediate and generic. This classification is mainly used to analyse Aegis and Kagent, since their interfaces accept natural-language requests and can therefore be tested with different levels of specificity. It is not directly applicable to Nimbus, whose interface is based on predefined intent identifiers, and only partially applicable to KubeTeus because of the adaptations required for its unavailable components. KubeTeus accepts only well-formatted JSON with all the information required.

Before comparing performance, the next subsection establishes the CNI coverage of the four systems. This provides the necessary context for interpreting the subsequent measurements, since the systems do not all target the same policy mechanisms.

### 7.5.1 CNI Coverage

The four tools do not operate over the same CNI scope. This difference needs to be considered before comparing their performance, since the policy resources they generate and the amount of CNI-specific reasoning required are not identical across systems.

Nimbus has the narrowest scope in this comparison, but this follows directly from its design. Its NetPol adapter produces standard Kubernetes `NetworkPolicy` objects and does not target a CNI-specific policy resource. The adapter therefore operates through the CNI-agnostic Kubernetes networking API rather than through a specific CNI's custom resources.

Kagent has a broader framework-level scope. The pre-built `cilium-policy-agent` evaluated here targets CiliumNetworkPolicy, while the framework also provides agents for Kubernetes resources and other tools such as Prometheus, Helm and Istio. These additional agents are not themselves CNI implementations. The framework nevertheless allows additional agents to be defined and connected to different tools, meaning that a CNI-specific policy agent could in principle be implemented following the same approach.

This evaluation is limited to the pre-built Cilium policy agent. Broader CNI coverage should therefore be understood as a property of the framework's extensibility rather than as a capability directly evaluated in this thesis.

KubeTeus presents a difference between its documented and reproducible scope. The published material describes support for Cilium, Calico, Weave and Antrea. Inspection of the available source code, however, shows that the downstream policy generation and validation logic evaluated here is specifically implemented for Cilium. Equivalent generation and validation paths for the other three plugins are not present in the available implementation. The KubeTeus results reported in this chapter should therefore be interpreted as a Cilium-only evaluation. This distinction is important when comparing the four systems, since the documented scope of a tool is not necessarily equivalent to the functionality that can be reproduced with the available implementation.

Aegis supports both Cilium and Calico through separate CNI-specific generation and validation prompts, as described in Chapter 5. Extending the system to another CNI would require an additional generation and validation configuration together with the corresponding CNI-detection branch, while the overall graph structure could remain unchanged.

Table 7.2 summarises the CNI coverage considered in this evaluation.

**Table 7.2:** CNI coverage of the four tools, as implemented and tested.

| Tool | Implemented and tested |
|---|---|
| Nimbus | Kubernetes-native NetworkPolicy only |
| Kagent | Cilium (pre-built agent) |
| KubeTeus | Cilium only |
| Aegis | Cilium, Calico |

The difference in scope should be kept in mind when interpreting the results that follow. The tools are not evaluated under identical architectural constraints, and some comparisons therefore describe practical capabilities rather than controlled measurements of equivalent systems.

### 7.5.2 Human Intervention Requirements

Figure 7.1 reports the average number of human interventions required per intent for Aegis and Kagent. The measurements are based on the same ten intent batches used for this comparison, with error bars representing one standard deviation across repeated runs.

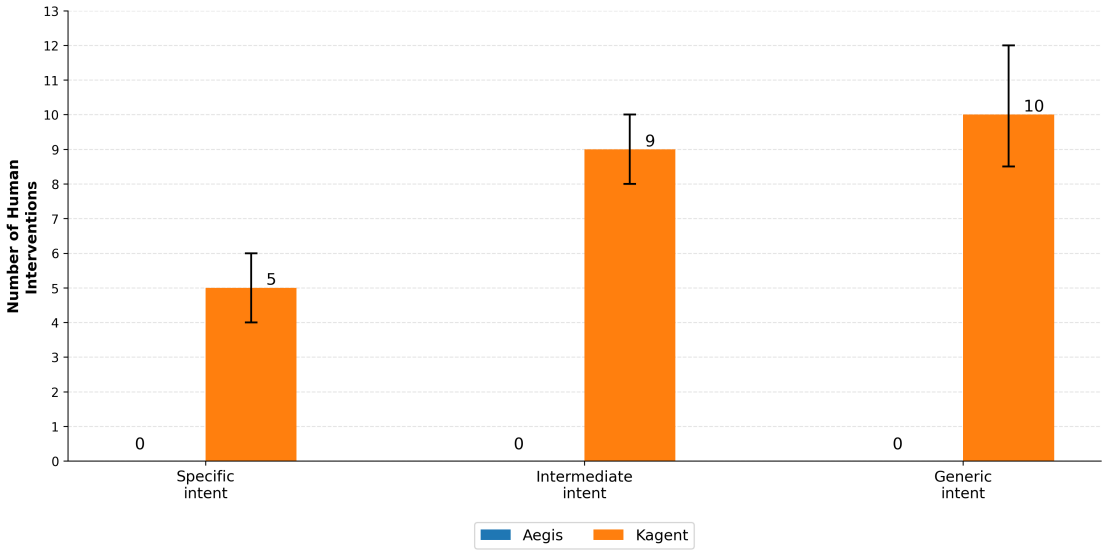



**Figure 7.1:** Number of human interventions required per intent by Aegis and Kagent.

Aegis required no human intervention across the three intent groups. Ambiguity in the wording is handled within the pipeline using the cluster information collected by the discovery layer. The system therefore does not pause execution to request

clarification from the operator. The absence of human intervention does not mean that ambiguity has no cost. As shown in Section 7.4, generic intents require more tokens and have a slightly lower pass rate. In this case, the cost of missing information is reflected in additional computation and, in some cases, in an incorrect result rather than in a request for clarification.

Kagent shows the opposite trend. For specific intents, five interventions were recorded on average. The number increased to nine for intermediate intents and reached ten for generic intents. This behaviour is consistent with the lack of cluster awareness in the evaluated policy-generation agent. Even when the request is technically detailed, the agent may request confirmation of information or of the steps required to complete the operation. As the request becomes less specific, more information has to be supplied by the human operator because the agent cannot resolve the ambiguity by inspecting the running workloads.

The comparison therefore concerns not only the number of corrections but also where the cost of ambiguity is placed. Kagent transfers part of this cost to the operator, while Aegis handles it within the reasoning pipeline. In Aegis, this results in additional inference and a small decrease in success rate for generic intents, but the system does not require human clarification during execution. From an operational perspective, this difference is relevant. A workflow that asks the user to resolve every missing technical detail can still produce correct policies, as shown by the ten successful Kagent intents in the common batch. However, the operator remains part of the execution loop. Aegis instead attempts to resolve the same type of missing information using the available cluster context.

### 7.5.3 Token Consumption

Token consumption is compared only for systems that rely on language-model inference. Nimbus is therefore excluded from this metric because its evaluated pipeline does not invoke a language model.

Figure 7.2 compares the token consumption of Aegis, Kagent, and KubeTeus. KubeTeus uses approximately 700 tokens per intent, considerably fewer than the other two systems. This lower consumption is consistent with its simpler processing path. The evaluated pipeline does not include a separate translation stage, a generation-validation loop or a dedicated test-generation stage. Fewer model calls naturally result in lower token consumption. This reduction, however, should not be considered independently from the reliability issues observed in Section 7.3.

Aegis shows a stronger dependence on the specificity of the input. Technically explicit intents, where the port, protocol, namespace and label are already provided, require considerably fewer tokens than generic intents. Less information needs to be inferred during translation and generation, so the corresponding reasoning process is shorter. The increase in token consumption for generic intents is therefore not

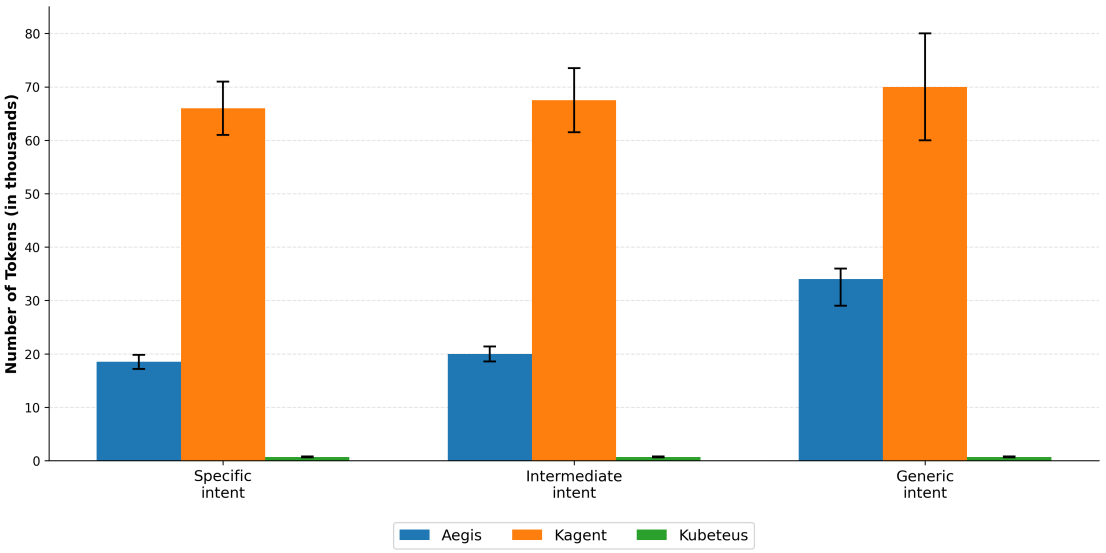



**Figure 7.2:** Comparison of tokens used per intent by the evaluated language-model-based systems.

simply an overhead of the implementation. It reflects the additional work required when the user provides less technical information. The system has to use cluster context to identify the relevant workloads and may also require further reasoning or validation before producing a policy that can be applied.

Kagent's token consumption is comparatively stable across the three intent groups. Part of the explanation is architectural: its workflow includes calls to the separate `k8s-agent` for enforcement and cluster interaction. These operations contribute to the total token count even when the original intent is already highly specific. Aegis does not require an additional language-model call for these operations. Enforcement, runtime testing and rollback are performed by the deterministic controller described in Chapter 6. Consequently, the additional work required for less specific intents is concentrated in the reasoning pipeline rather than being introduced by the enforcement mechanism itself.

The token results should therefore not be interpreted as a direct measure of system quality. KubeTeus uses fewer tokens, but its evaluation also revealed limitations in validation and enforcement. Aegis uses more tokens because it performs more stages and incorporates the cluster context, while the resulting policy is subsequently checked and tested. Token consumption is consequently better understood as a measure of computational cost within the overall workflow rather than as an objective to minimise independently.

### 7.5.4 Execution Time

In addition to human intervention and token consumption, the evaluation considers the time required to process an intent. The measured time includes not only policy generation, but the complete execution of the pipeline.

For Aegis, the execution time is measured using timestamps recorded by the controller. The starting timestamp is taken when the intent is received, while the ending timestamp is recorded after the runtime test has been executed. The measurement therefore covers all the steps involved in processing the intent, including cluster discovery, policy generation and validation, enforcement, and runtime testing.

The execution time varies according to the specificity of the intent. As shown in Figure 7.3, specific intents require approximately 30 seconds, intermediate intents around 1 minute, and generic intents approximately 2 minutes. The increase in execution time is mainly due to additional information that has to be inferred for less specific requests. In particular, these intents require more work during the translation and cluster discovery stages before a concrete policy can be generated and tested.

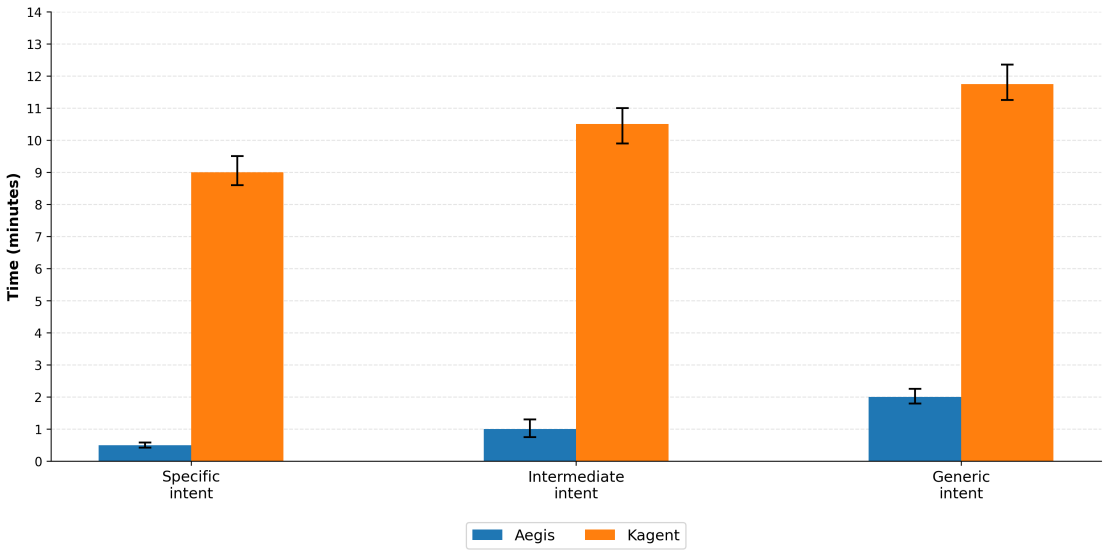



**Figure 7.3:** Time required per type of intent by Aegis and Kagent.

The figure also shows a clear difference between Aegis and Kagent. Aegis remains within a few minutes even for generic intents, whereas Kagent requires several minutes for all three levels of specificity. For Aegis, the increase in time is mainly related to the additional reasoning and cluster information required by less specific requests. For Kagent, the execution time is more strongly affected by the

interaction required from the operator.

For Kagent, the execution time was measured manually and is therefore less precise than the measurements obtained for Aegis. The measured time ranges from approximately 9 minutes for the more specific intents to around 12 minutes for the less specific ones. These values include the additional interactions required to clarify the intent and complete the process. Although the measurements are approximate, the difference compared with Aegis is sufficiently large to show that the Kagent workflow requires considerably more time.

A direct comparison based on intent specificity cannot be performed for KubeTeus. Its evaluation uses structured JSON input rather than the natural-language intents used for Aegis and Kagent, and the same specific, intermediate, and generic classification cannot therefore be applied. Its overall execution time can nevertheless be used as a reference point. For KubeTeus, the processing time is approximately 20 seconds, measured using timestamps around the policy-generation process.

Figure 7.4 provides a general comparison of the execution time of Aegis, Kagent, and KubeTeus, without grouping the results by intent specificity. This comparison gives an overall view of the time required by the three systems while avoiding a direct comparison between input formats that are not equivalent.

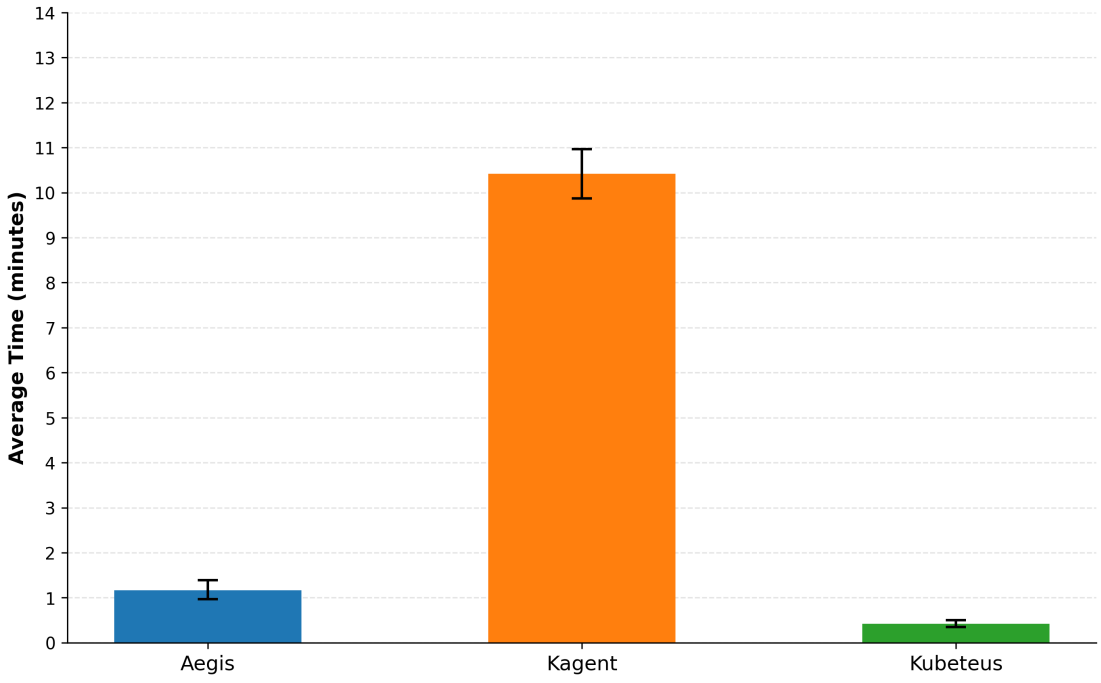



**Figure 7.4:** Comparison of time required per intent by Aegis, Kagent, and KubeTeus.

The results show that Aegis and KubeTeus complete their respective processes within seconds or a few minutes, while Kagent requires substantially more time.

### 7.5.5 Comparative Results

As previously discussed, the four tools were evaluated on a common batch of ten intents, within the limits imposed by the interface and implementation of each system. Figure 7.5 reports the resulting number of successful intents together with confidence intervals.

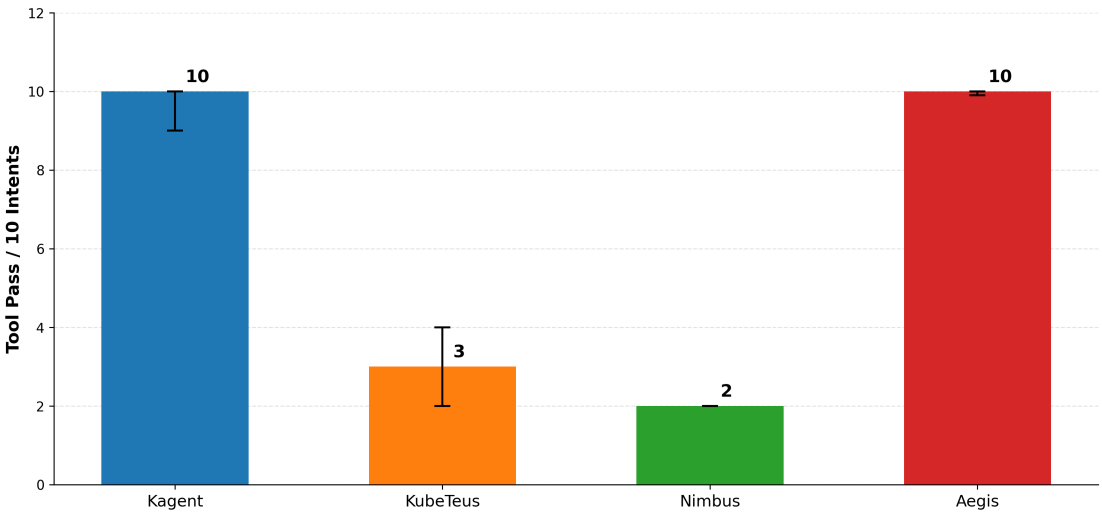



**Figure 7.5:** Pass rate across the four tools on a shared batch of ten intents, with confidence intervals.

Aegis and Kagent both achieved ten successful intents out of ten in this batch. The two results, however, need to be interpreted differently.

The Kagent result represents the outcome after human intervention. Each intent could therefore benefit from operator review and correction before the policy was considered complete. The result measures what can be achieved through the Kagent workflow with human assistance rather than the accuracy of the policy-generation agent in isolation.

The Aegis result was obtained without manual correction. Each intent passed through automated generation and runtime testing. The same mechanism is used in the larger benchmark presented in Section 7.4, where Aegis achieved an overall pass rate of 97% when considering only intents for which the runtime test could be executed.

The confidence intervals also need to be considered when interpreting the ten-intent comparison. A sample of ten observations provides only limited statistical

evidence. A single additional failure would have a substantial effect on the estimated pass rate. The larger Aegis benchmark provides a more stable estimate of its behaviour because it includes one hundred intents for each CNI.

KubeTeus passed three of the ten intents in the shared batch. The causes are consistent with the limitations described in Section 7.3, including the substituted entity-classification stage, the schema mismatch between the policy converter and validator, and errors encountered during enforcement.

Nimbus successfully handled two of the ten intents. These results should not be interpreted in the same way as the failures of the other tools. Nimbus does not attempt to interpret arbitrary natural-language intents. The two successful cases correspond to behaviours covered by its implemented NetPol identifiers, while the remaining requests were outside the scope of the evaluated adapter. For this reason, the pass rate of Nimbus cannot be interpreted simply as a measure of poor policy generation accuracy. A request outside its predefined vocabulary is a limitation of the system's supported scope, not a failed attempt to infer the requested behaviour. The same distinction applies, to a different extent, to the adaptations required for KubeTeus.

### 7.5.6 Overall Comparison Across Metrics

The pass rate obtained on the common batch is only one of the dimensions considered in this evaluation, and it cannot be interpreted in isolation from the other metrics discussed in the previous subsections. Considering human intervention, token consumption, execution time and pass rate together highlights different trade-offs rather than identifying a single system as universally superior. Kagent combines model-based generation with human supervision: the ten-out-of-ten result on the common batch was obtained at the cost of a substantial number of operator interventions and an execution time of several minutes per intent, as reported in Section 7.5.2 and Section 7.5.4. Nimbus provides deterministic behaviour within a fixed vocabulary, with no language-model inference and consequently no associated token cost, but at the price of a supported scope limited to its predefined intent identifiers. KubeTeus follows a comparatively lightweight generation pipeline, with the lowest token consumption and execution time among the language-model-based systems, but the available implementation introduces limitations in entity extraction and validation that are reflected in its low pass rate on the shared batch.

The main advantage of Aegis in this comparison is therefore not simply the ten-out-of-ten result obtained on the small common batch. That result alone would not be sufficient to draw strong conclusions from a statistical perspective. The more relevant observation is that Aegis obtained this result without manual clarification, with no human intervention recorded across any of the three intent groups, and with an execution time that remained within a few minutes even for generic intents.

The same workflow was subsequently evaluated on a larger set of one hundred intents in both Cilium and Calico. Taken together, these results indicate a trade-off between the amount of information supplied by the user and the computational effort required by the system, rather than a uniform advantage across every metric: less specific intents require additional tokens and processing time and show a small decrease in pass rate, but this cost is absorbed within the automated pipeline instead of being transferred to the operator, as is instead the case for Kagent.

Taken together, the experiments show that the main contribution of Aegis is not only the generation of a valid network policy from a natural-language request. The complete workflow combines interpretation, cluster discovery, policy generation, validation, enforcement and runtime testing, while keeping human intervention outside the normal execution path. This allows the system to handle requests that are less technically precise than those required by approaches based on predefined intents or manual clarification, at the cost of additional model usage and a small decrease in success rate for the most generic inputs.

# Chapter 8

# Conclusions and Future Work

This thesis addressed the gap between what an operator wants to enforce on a Kubernetes cluster and the low-level, CNI-specific YAML required to enforce it. The proposed system, Aegis, extends KubeTeus with a multi-LLM pipeline in which distinct models are responsible for interpreting a natural-language intent, generating the corresponding network policy, validating and correcting it, and finally testing it directly against a running cluster. Aegis goes beyond manifest generation by verifying that the enforced policy actually behaves as intended, which is ultimately the property an operator cares about.

The evaluation presented in Chapter 7 supports the feasibility of this approach. Across a benchmark of one hundred intents spanning both Cilium and Calico, the pipeline achieved a success rate of approximately 96%, with most of the observed failures attributable to test infrastructure limitations or transient communication errors rather than to incorrect policy generation. The results also show that performance degrades as intents become more vague: specific intents, which already state the port, protocol, and label involved, were resolved almost perfectly and at the lowest token cost, while generic, colloquial intents required substantially more reasoning and produced the majority of the observed failures. This behaviour is not specific to Aegis. It reflects a broader limitation of the problem: cluster context can help resolve some ambiguity, but it cannot fully compensate for an intent that is genuinely underspecified. The comparison against Kagent, Nimbus, and KubeTeus, discussed in Section 7.5.5, further situates this result: Aegis is the only one of the four tools reporting an unassisted pass rate over an open-ended set of natural-language intents, backed by a sample large enough to make the number meaningful rather than anecdotal.

## 8.1 Limitations

Despite these results, the system has several limitations.

First, the pipeline's reliability is tied to the underlying LLMs' ability to produce structured, schema-compliant output, which is a general-purpose capability and is not specifically tailored to Kubernetes networking. Although the validation and correction stages mitigate many of the resulting errors, they do not eliminate them entirely. A significant portion of the failures to achieve more general objectives can ultimately be attributed to the model's incorrect interpretation of the user's underlying request, rather than to a flaw in the pipeline's architecture.

Second, the system currently supports only two CNI plugins, Cilium and Calico. While these are among the most widely adopted, they do not represent the entire ecosystem, and organizations relying on other plugins, such as Weave or Antrea, cannot currently benefit from this work.

Finally, the evaluation itself, while broader than what is typically reported for comparable tools, still has boundaries: the hundred-intent benchmark tested in the same simulated environment, and the ten-intent comparative batch used with the other three tools is too small to support strong statistical claims on its own. The wide confidence intervals reported in Section 7.5.5 for Kagent and KubeTeus are themselves evidence of this limitation, not just of a property of those tools.

## 8.2 Future Work

Several directions could be explored in future work.

The most direct one concerns the models themselves. The current pipeline relies entirely on general-purpose LLMs accessed through their standard interfaces, with no task-specific adaptation beyond prompt design. KubeTeus's own authors addressed a similar problem by fine-tuning a model on Kubernetes network policy data, and a similar strategy could plausibly be applied here, particularly to the intent translation and policy generation stages, where a model fine-tuned on a corpus of intents paired with their correct, cluster-verified policies might achieve higher accuracy on generic, underspecified requests while consuming fewer tokens than the current few-shot approach. Whether this trade-off is worth the engineering cost of curating such a dataset and maintaining a fine-tuned model, however, is itself a question that only a dedicated study could answer.

A second, more incremental direction is extending CNI coverage. The pipeline's architecture already separates the policy-agnostic reasoning stages from the CNI-specific generation and enforcement logic, which suggests that adding support for Weave or Antrea would not require redesigning the system from the ground up, but

rather extending the existing generation and validation agents with additional CNI-specific knowledge, along the same lines followed when Calico support was added on top of the original Cilium-only implementation. This would also allow for a more complete comparison against tools such as KubeTeus, whose documentation claims broader CNI support than what is actually supported by its publicly available implementation. A third direction looks beyond network policies altogether. Nimbus, discussed in Chapter 3, takes a fundamentally different approach by mapping a fixed set of intents onto a range of security-relevant Kubernetes resources through hand-written adapters, rather than restricting itself to NetworkPolicy objects. A natural extension of the present work would be to generalize the reasoning pipeline beyond network policies, allowing it to target other enforcement surfaces, such as RBAC policies, Pod Security Standards, or admission control rules, following an adapter-based structure similar to Nimbus's, but retaining the natural-language flexibility and runtime verification that distinguish the approach proposed in this thesis. Such an extension would move the system closer to a general-purpose, intent-driven security layer for Kubernetes, rather than a tool scoped specifically to network connectivity.

Overall, these directions aim to further reduce the gap between the operator's intent and the configuration required to enforce it, while maintaining correctness and control.

# Acronyms

**A2A** Agent-to-agent

**API** Application Programming Interface

**BERT** Bidirectional Encoder Representations from Transformers

**BGP** Border Gateway Protocol

**CIDR** Classless Inter-Domain Routing

**CNCF** Cloud Native Computing Foundation

**CNI** Container Network Interface

**CRD** Custom Resource Definition

**eBPF** Extended Berkeley Packet Filter

**GPT** Generative Pre-trained Transformer

**HTTP** Hypertext Transfer Protocol

**IBN** Intent-Based Networking

**IETF** Internet Engineering Task Force

**IP** Internet Protocol

**JSON** JavaScript Object Notation

**K8s** Kubernetes

**LLM** Large Language Model

**MCP** Model Context Protocol

**MoE** Mixture of Experts

**NLP** Natural Language Processing

**RBAC** Role-Based Access Control

**SCTP** Stream Control Transmission Protocol

**TCP** Transmission Control Protocol

**TF-IDF** Term Frequency-Inverse Document Frequency

**UDP** User Datagram Protocol

**YAML** YAML Ain't Markup Language

# Bibliography


[1] Rakesh Keshava, Arun Kumar Elengovan, Isan Sahoo, Pavan Nutalapati, Bindu Mohan Harve, Deepak Kole, and Nandagopal Seshagiri. «Security at Scale: Ethical Container Exploitation in Orchestrated Environments with Kubernetes». In: *2025 5th International Conference on Intelligent Cybernetics Technology and Applications (ICICyTA)*. 2025, pp. 1–6. DOI: 10.1109/ICICyTA68677.2025.11362372 (cit. on pp. 1, 16).

[2] Docker. *Docker*. URL: https://www.docker.com/ (visited on 08/09/2026) (cit. on p. 4).

[3] Dirk Merkel. «Docker: Lightweight Linux Containers for Consistent Development and Deployment». In: *Linux Journal* 239 (2014). URL: https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment (cit. on p. 5).

[4] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. «An Introduction to Docker and Analysis of Its Performance». In: *International Journal of Computer Science and Network Security* 17.3 (2017), p. 228 (cit. on p. 5).

[5] Denis Zuev, Artemii Kropachev, and Aleksey Usov. *Learn OpenShift*. Packt Publishing, July 2018. Chap. Docker Architecture. ISBN: 9781788992329. URL: https://www.oreilly.com/library/view/learn-openshift/9781788992329/33d025bf-27fa-49b9-99b3-673a20ae6d1e.xhtml (cit. on p. 5).

[6] Sumit Sachdeva. «Kubernetes and Docker: An Introduction to Container Orchestration and Management». In: *International Journal of Computer Trends and Technology* 71.8 (2023), pp. 57–62 (cit. on p. 6).

[7] Kubernetes Authors. *Kubernetes*. URL: https://kubernetes.io/ (visited on 08/09/2026) (cit. on p. 6).

[8] Kubernetes Authors. *Kubernetes*. 2019. URL: https://kubernetes.io/ (visited on 05/24/2019) (cit. on p. 7).

[9] Kubernetes Authors. *Kubernetes Components*. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 08/09/2026) (cit. on p. 7).

[10] Bom Kim, Jinwoo Kim, and Seungsoo Lee. «Exploring Security Enhancements in Kubernetes CNI: A Deep Dive Into Network Policies». In: *IEEE Access* 13 (2025), pp. 35322–35338. DOI: 10.1109/ACCESS.2025.3543841 (cit. on p. 7).

[11] Kubernetes Authors. *Network Policies*. URL: https://kubernetes.io/docs/concepts/services-networking/network-policies/ (visited on 08/09/2026) (cit. on pp. 8, 9).

[12] Calico Authors. *Calico*. URL: https://docs.tigera.io/calico/latest/about (visited on 08/09/2026) (cit. on pp. 10–12).

[13] Cilium Authors. *Cilium*. URL: https://docs.cilium.io/en/stable/overview/intro/ (visited on 08/09/2026) (cit. on pp. 12, 14, 15).

[14] Cyber Security Incident Database (CSIDB). *Tesla (2018-01-20)*. Accessed: 2026-09-10. 2018. URL: https://www.csidb.net/csidb/incidents/4cacc7ee-f23f-4ec9-9633-6fcf58b5283d/ (visited on 09/10/2026) (cit. on p. 17).

[15] The Chief IO. *Cryptojacking Attacks in Kubernetes: A Serious Threat That Deserves Attention*. Accessed: 2026-09-10. 2021. URL: https://thechief.io/c/editorial/cryptojacking-attacks-kubernetes-serious-threat-deserves-attention/ (visited on 09/10/2026) (cit. on p. 17).

[16] Red Hat. *The State of Kubernetes Security Report, 2024 Edition*. Accessed: 2026-09-10. 2024. URL: https://www.redhat.com/tracks/_pfcdn/assets/10330/contents/646381/001b54d3-7c21-417b-995c-ebe7be8328a1.pdf (visited on 09/10/2026) (cit. on p. 17).

[17] Internet Engineering Task Force. *IETF*. URL: https://www.ietf.org/ (visited on 08/09/2026) (cit. on p. 18).

[18] Nathan Sousa, Nazrul Islam, Danny Perez, and Christian Esteve Rothenberg. «Policy-Driven Network Traffic Rerouting Through Intent-Based Control Loops». In: July 2019 (cit. on p. 18).

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. «Attention Is All You Need». In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017. URL: https://arxiv.org/abs/1706.03762 (cit. on p. 19).

[20] Ian L. Alberts, Thomas Pyka, Axel Rominger, Ali Afshar-Oromieh, Kuangyu Shi, and George Prenosil. «Large language models (LLM) and ChatGPT: what will the impact on nuclear medicine be?» In: *European Journal of Nuclear Medicine and Molecular Imaging* 50 (2023). DOI: 10.1007/s00259-023-06172-w (cit. on p. 19).

[21] Ryan M. Nefdt. *What It's Like to Be an LLM*. Manuscript. 2026. URL: https://philpapers.org/rec/NEFWIL (visited on 08/09/2026) (cit. on p. 19).

[22] Wenpeng Yin, Qinyuan Ye, Pengfei Liu, Xiang Ren, and Hinrich Schütze. «LLM-driven Instruction Following: Progresses and Concerns». In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*. Ed. by Qi Zhang and Hassan Sajjad. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 19–25. DOI: 10.18653/v1/2023.emnlp-tutorial.4. URL: https://aclanthology.org/2023.emnlp-tutorial.4/ (cit. on p. 20).

[23] Ariel Rosenfeld and Teddy Lazebnik. «Whose LLM Is It Anyway? Linguistic Comparison and LLM Attribution for GPT-3.5, GPT-4 and Bard». In: *Mathematics* 14.10 (2026), p. 1683. DOI: 10.3390/math14101683. URL: https://doi.org/10.3390/math14101683 (cit. on p. 20).

[24] SE-RAN Team. *Nimbus: Intent-Driven Security Automation*. 2022. URL: https://www.5gsec.com/post/technical-nimbus-intent-driven-security-automation (visited on 08/09/2026) (cit. on pp. 21, 22).

[25] Bom Kim, Hyeonjun Park, and Seungsoo Lee. «KUBETEUS: An Intelligent Network Policy Generation Framework for Containers». In: *IEEE INFOCOM 2025 - IEEE Conference on Computer Communications*. 2025, pp. 1–10. DOI: 10.1109/INFOCOM55648.2025.11044592 (cit. on pp. 21, 24, 25).

[26] Anukul Parajuli, Krishna M. Sivalingam, and Madhan Raj Kanagarathinam. «A Framework for Intent-Driven Kubernetes Configuration Generation Using Large Language Models». In: *GenAI and LLMs for Beyond 5G Networks*. Cham: Springer Nature Switzerland, 2026, pp. 273–293. ISBN: 978-3-032-06418-9. DOI: 10.1007/978-3-032-06418-9_13. URL: https://doi.org/10.1007/978-3-032-06418-9_13 (cit. on pp. 21, 22).

[27] SE-RAN Team. *Nimbus GitHub Code*. URL: https://github.com/5GSEC/nimbus/ (visited on 08/09/2026) (cit. on p. 22).

[28] kagent contributors. *What Is kagent*. 2026. URL: https://kagent.dev/docs/kagent/introduction/what-is-kagent (visited on 07/22/2026) (cit. on p. 23).