Politecnico di Torino

Politecnico di Torino

Cybersecurity

A.a. 2024/2025

Graduation Session 07 2025

Securing the computing continuum with fine-grained automatic network policies

Supervisors:
Fulvio Giovanni Ottavio Risso
Stefano Galatino

Candidate:
Giulio Brazzo

Acknowledgements

É stato un lungo processo e ci sono stati molti contributi che hanno reso possibile questo lavoro. Vorrei esprimere la mia sincera gratitudine a tutti coloro che mi hanno supportato lungo il cammino.

In primo luogo, desidero ringraziare il mio relatore, il Prof. Fulvio Risso, per avermi seguito con pazienza e dedizione durante tutto il percorso di ricerca. Nonostante non fossi un esperto di cloud in nessuna sua forma, mi ha concesso la possibilità di lavorare su questo progetto, dimostrando fiducia nelle mie capacità. Inoltre una menzione anche ad Attilio e ai ragazzi del Lab9 che durante questi mesi di tesi mi hanno ascoltato e aiutato fornendomi spunti, consigli e supporto morale.

A proposito della mia inesperienza, un ringraziamento va al team di ArubaKube, per avermi accolto e guidato durante il mio percorso di tesi, in particolare a Francesco Torta che non si è mai stancato di ripetermi mille volte le stesse cose e per avermi spiegato anche le cose piú basilari.

C'è poi da considerare l'aspetto di vita quotidiana, un particolare merito e ringraziamento va ai miei amici storici: Flavio e Filippo. Mi avete supportato dalla scuola dell'infanzia fino ad oggi, regalandomi svago e giornate di spensieratezza anche nei momenti o periodi piú difficili. Per le nostre serate studio-smash, per non avermi mai giudicato per come sono, per avermi spinto a superare me stesso e a migliorare, sin dalla prima e unica "discussione" che abbiamo avuto, non avete mai smesso di starmi vicino. Sicuramente senza la vostra spinta e il vostro modo di spronarmi non sarei riuscito a portare a termine alcun percorso.

Un ringraziamento speciale ovviamente anche a tutti gli altri amici, la cui lista è troppo lunga per essere menzionata qui, e che quindi riassumerò con i nomi dei vari gruppi: Bottarga e PHG. Dal primo all'ultimo mi siete stati vicini, nelle giornate di studio e in quelle di divertimento.

Impossibile non ringraziare i miei genitori, Marco e Iolanda. Mi avete permesso, anche se per piú tempo del previsto, di perseguire nel mio percorso universitario senza mai mettere in dubbio le mie capacità e fornendomi tutto il supporto di cui avevo bisogno. Non ho mai sentito la mancanza di qualcosa, possibilità o qualsivoglia forma di desiderio. Vi ringrazio per i sogni che mi avete permesso di seguire, senza mai chiedere nulla in cambio e senza mai pretendere un successo nelle

varie prove che ho dovuto affrontare. Grazie per avermi sopportato in casa anche nei periodi piú grigi, pre esami o semplicemente in periodi in cui ero particolarmente scontroso. Spero un giorno di riuscire a ripagarvi per tutta la fiducia che avete riposto in me.

Ringrazio mia sorella, Chiara, per avermi sempre spronato a fare esperienze che altrimenti non avrei neanche pensato, per essermi stata vicino insieme a mamma e papà e per l'esempio di tenacia che serve per affrontare la vita. Nonostante il percorso accademico altalenante, mi hai dimostrato come inseguire i propri sogni sia la base della felicità.

Infine ringrazio Agnese. Ti ringrazio per le ore infinite di chiacchiere fino alle 6 del mattino nonostante entrambi avessimo impegni, per le lunghe videochiamate e per i preziosi consigli. Per avermi fatto scoprire lati di me che neanche pensavo potessero uscire, per avermi fatto cambiare il modo di vedere le cose, per non essermi sentito giudicato qualsiasi confessione ti abbia detto. Anche se é difficile riassumere in poche righe l'intensità di quelle che é stato ed di quelle che é, dopo anni spenti e anonimi, sei arrivata travolgendomi e portandomi con te, rompendo quella monotonia e apatia con cui da tempo ormai convivevo.

"La mia vita con te è di nuovo a colori"

Abstract

The increasing adoption of the computing continuum, where applications span across cloud, edge, and on-premise infrastructures, has introduced new challenges in securing network communications. In such heterogeneous and dynamic environments, Kubernetes has emerged as the standard platform for orchestrating containerized workloads. However, its native networking model and built-in NetworkPolicies are often insufficient to guarantee fine-grained and adaptive traffic control, especially in multi-cluster scenarios.

This thesis investigates how to achieve precise and automated network isolation within Kubernetes-based multi-cluster topologies, with a focus on deployments extended through Liqo, an open-source framework for transparent multi-cluster resource sharing. The proposed solution introduces multiple Kubernetes controllers capable of observing shared resources between different clusters, and dynamically generate security policies mapped to low-level nftables firewall rules or through Kubernetes Network Policies.

Specifically the aim is to define and enforce clear security boundaries around a Kubernetes cluster that is part of a multi-cluster topology. To achieve this, the proposed system introduces a mechanism for selectively blocking network traffic within a peered cluster by dynamically applying fine-grained filtering rules. This is accomplished through the combined use of low-level nftables rules for precise traffic control, and Kubernetes-native controllers. The controllers continuously monitor the environment and adapt network isolation strategies based on workload placement, origin, and namespace context. In doing so, this thesis delivers a flexible and extensible framework that automates network policy enforcement and reduces manual configuration overhead, while ensuring robust workload isolation across cluster boundaries.

Table of Contents

Chapter 1

Introduction

In recent years, the evolution of distributed systems and the increasing availability of heterogeneous resources have led to the emergence of the *computing continuum* paradigm. This model envisions a seamless integration of cloud, edge, and on-premise infrastructures, allowing applications to dynamically span across multiple execution layers. While this approach enables new levels of flexibility and scalability, it also introduces non-trivial challenges in terms of security, especially regarding communication across infrastructure boundaries.

Kubernetes has become the de facto standard for orchestrating containerized workloads, providing primitives for scalability, scheduling, and service abstraction. However, when Kubernetes clusters are interconnected, particularly in multi-cluster and federated setups, enforcing consistent and granular network policies becomes a critical concern. In such environments, traditional Kubernetes *Network Policies* may not provide the level of expressiveness or control required to securely isolate workloads, especially when those workloads are dynamically moved or replicated across clusters.

This thesis focuses on addressing this problem by designing and implementing a system that dynamically generates fine-grained network policies, capable of adapting to the evolving nature of the computing continuum. The proposed solution integrates a Kubernetes-native controller, enabling a more expressive and low-level filtering mechanism by combining *nftables rules* and Kubernetes Network Policies, depending on the context. The implementation is evaluated in the context of **Liqo**, an open-source project for Kubernetes multi-cluster federation, where workloads can be transparently offloaded from one cluster to another.

1.1 Goal of the thesis

The thesis aims to define and enforce clear security boundaries around a Kubernetes cluster that is part of a multi-cluster topology. To achieve this, the proposed system introduces a mechanism for selectively blocking network traffic within a peered cluster by dynamically applying fine-grained filtering rules. This is accomplished through the combined use of low-level `nftables rules` for precise traffic control, and Kubernetes-native controllers. The controllers continuously monitor the environment and adapt network isolation strategies based on workload placement, origin, and namespace context. In doing so, this thesis delivers a flexible and extensible framework that automates network policy enforcement and reduces manual configuration overhead, while ensuring robust workload isolation across cluster boundaries.

Chapter 2

Background

To fully understand the design choices, implementation details, and the overall contribution of this thesis, it is essential to first introduce the core technologies on which the work is based. In particular, the system designed and implemented relies heavily on Kubernetes, as well as on Liqo, a framework that extends Kubernetes to support multi-cluster topologies, and on nftables, a framework for packet filtering and firewalling. A solid understanding of Kubernetes and its internal components is especially important, as the proposed system integrates tightly with the Kubernetes control plane and resource model. Concepts such as Pods, Namespaces, Controllers and Custom Resource Definitions (CRDs) are fundamental to appreciate how security policies are generated and enforced dynamically in response to changes in the cluster topology and workload placement. In addition to Kubernetes, a thorough understanding of Liqo is crucial to fully understand the scope and challenges addressed by this work. Liqo extends Kubernetes by enabling seamless federation of independent clusters, allowing resources such as Pods and Namespaces to be offloaded across administrative boundaries. This capability introduces new security challenges, as traditional intra-cluster isolation mechanisms are no longer sufficient. The system proposed in this thesis builds upon Liqo's offloading mechanisms to implement network isolation and access control policies that span multiple clusters, ensuring that cross-cluster interactions respect strict security boundaries.

2.1 Kubernetes

Kubernetes is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. Originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes has become the de facto standard for managing distributed workloads in cloud-native environments. This section introduces the key components of Kubernetes that are essential for understanding the architecture and implementation of the system proposed in this thesis. In particular, it focuses on the concepts of clusters, workloads, resource isolation, extensibility mechanisms, and network policy enforcement.

2.1.1 Cluster Architecture

A Kubernetes cluster is composed of a control plane and a set of worker machines, known as nodes, which are responsible for running containerized applications. At least one worker node is required in every cluster to host and execute Pods. The control plane is responsible for managing the worker nodes and the Pods running within the cluster. In production-grade deployments, it is typically distributed across multiple machines to ensure fault tolerance and high availability. Similarly, the cluster itself is composed of several nodes to support scalability and resilience.

Control plane components are responsible for maintaining the desired state of the cluster, scheduling workloads, and managing resources. The key components of the control plane include:

- **kube-api-server**: The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.
- **etcd**: A distributed key-value store that stores all cluster data, including configuration, state, and metadata.
- **kube-scheduler**: The scheduler is responsible for assigning Pods to nodes based on resource requirements and constraints.
- **kube-controller-manager**: The controller manager runs controllers that monitor the state of the cluster and make decisions to ensure that the desired state matches the actual state.
- **cloud-controller-manager**: This component interacts with the underlying cloud provider to manage resources such as load balancers, storage, and networking.

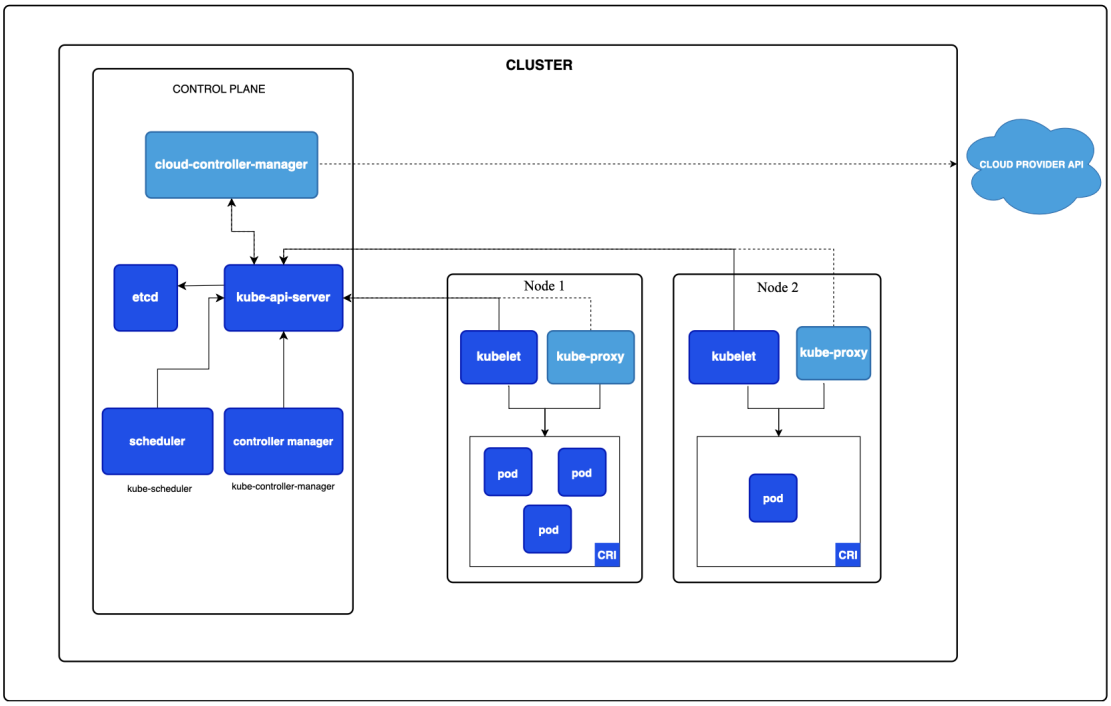

Figure 2.1: Kubernetes Cluster Architecture

Node components are responsible for running the actual workloads and managing the lifecycle of Pods. Each node in a Kubernetes cluster runs several key components:

- **kubelet**: An agent that runs on each node and ensures that containers are running as expected. It communicates with the control plane to report the status of Pods and to receive instructions.
- **container runtime**: The software responsible for running containers. Kubernetes supports various container runtimes, such as Docker, containerd, and CRI-O.
- **kube-proxy (optional)**: A network proxy that maintains network rules on nodes, allowing Pods to communicate with each other and with external services.

2.1.2 Workload Abstractions

Kubernetes provides several high-level abstractions to define and manage workloads:

ReplicaSet: Ensures that a specified number of identical Pods are running at all times.

Deployment: A higher-level abstraction that manages ReplicaSets and provides declarative updates to Pods.

DaemonSet: Ensures that a copy of a Pod is running on each node in the cluster.

StatefulSet: Manages stateful applications by providing stable network identities and persistent storage.

2.1.3 Namespaces and Resource Isolation

Namespaces provide a mechanism for isolating groups of resources within the same cluster. They are commonly used to divide environments (e.g., development, staging, production) or to separate different tenants in multi-tenant architectures. Namespaces are a key concept in this thesis because the system implements fine-grained network policies based on the namespace origin of workloads. For example, offloaded Pods from remote clusters are automatically placed in dedicated namespaces, which can be used as a identifier for policy boundary.

2.1.4 Controllers and Operators

Controllers are control loop processes that watch the state of the cluster and reconcile it with a desired state. Kubernetes provides built-in controllers for core resources (e.g., Pods, Deployments), but the platform is also extensible through the concept of Operators. An Operator is a specialized controller that encodes domain-specific knowledge to manage custom resources and complex applications. In this thesis, custom controllers are implemented to observe and manage network isolation policies dynamically, based on the presence of offloaded Pods and contextual information such as namespaces and peer clusters.

2.1.5 Custom Resource Definitions (CRDs)

Custom Resource Definitions are a powerful feature of Kubernetes that allows users to extend the Kubernetes API with their own resource types. CRDs enable the creation of custom resources that behave like native Kubernetes resources, allowing users to define and manage their own application-specific objects and workflows. A CRD is defined by creating a YAML manifest that specifies the name, schema, and behavior of the new resource type. Once applied to the cluster, users can create, read, update, and delete instances of the custom resource using standard

Kubernetes tools such as `kubectl`. The Kubernetes API server stores these custom resources in `etcd` alongside native resources, and controllers can watch and react to changes in their state.

CRDs are a powerful Kubernetes extension mechanism that allows users to define new types of resources. They are the foundation for building Operators and other Kubernetes-native extensions. In the proposed system, CRDs are used to represent high-level network isolation intents, which are then translated by the controller into low-level firewall rules and Kubernetes Network Policies.

2.1.6 Networking and Network Policies

Kubernetes provides a flexible networking model where each Pod gets its own IP address, and Pods can generally communicate with each other across Nodes. However, this model is too permissive for many use cases that require stricter security boundaries. To address this, Kubernetes supports Network Policies, which allow administrators to define rules that control traffic flow between Pods based on labels and namespaces. Network Policies are enforced by the Container Network Interface (CNI) plugin used by the cluster (e.g., Calico, Cilium). In this thesis, Network Policies are used in conjunction with nftables to implement a layered filtering mechanism. While Network Policies provide a Kubernetes-native way to express security rules, nftables enables low-level enforcement that is independent of the container runtime or network plugin.

2.1.7 Admission Webhooks

Webhooks are a powerful mechanism in Kubernetes that allows users to extend the API server's functionality by intercepting requests to create, update, or delete resources. They can be used for validation, mutation, or admission control of resources before they are persisted in the cluster. Webhooks main use case is to validate and mutate resources before they are stored in the cluster. For example, a validation webhook can check if a new resource meets certain criteria (e.g., required fields, valid values) before allowing it to be created. A mutation webhook can modify the resource to add default values or apply transformations. Official kubernetes documentation defines Admission Webhooks as:

> Admission webhooks are HTTP callbacks that receive admission requests and do something with them. [...] Mutating admission webhooks are invoked first, and can modify objects sent to the API server to enforce custom defaults. After all object modifications are complete, and after the incoming object is validated by the API server, validating admission webhooks are invoked and can reject requests to enforce custom policies. [1]

There are two types of webhooks in Kubernetes:

- **Validating Webhooks**: These webhooks are called before a resource is persisted in the cluster. They can reject requests that do not meet validation criteria, preventing invalid resources from being created or updated.
- **Mutating Webhooks**: These webhooks are called after a resource is validated but before it is stored. They can modify the resource, adding default values or applying transformations.

In this thesis, webhooks are used to validate and mutate custom resources related to the extended Custom Resource Definition for Network Security Engine part. They ensure that only well-formed and secure configurations are applied to the cluster, preventing misconfigurations that could lead to security vulnerabilities or network disruptions.

2.2 Liqo

Liqo is an open-source framework designed to extend Kubernetes with multi-cluster capabilities, enabling seamless workload sharing and resource management across multiple Kubernetes clusters. It provides a set of APIs and controllers that facilitate the discovery, communication, and synchronization of resources between clusters, allowing users to create a unified environment for deploying and managing applications across different Kubernetes instances. Liqo's architecture is based on a peer-to-peer model, where clusters can dynamically discover and connect to each other, forming a federated network of Kubernetes clusters. This approach allows for efficient resource sharing, workload offloading, and cross-cluster communication, making it suitable for scenarios such as hybrid cloud deployments, multi-cloud strategies, and edge computing. Liqo's key features include:

- **Workload Offloading**: Liqo allows users to offload workloads from one cluster to another, enabling efficient resource utilization and load balancing across clusters.
- **Cross-Cluster Communication**: Liqo provides mechanisms for secure and efficient communication between clusters, allowing applications to access resources and services across different Kubernetes instances.
- **Resource Sharing**: Liqo enables the sharing of resources such as Pods, Services, and ConfigMaps between clusters, facilitating collaboration and resource optimization.
- **Dynamic Discovery**: Clusters can dynamically discover each other and establish connections, allowing for flexible and scalable multi-cluster architectures.

2.2.1 Architecture

In this section we will introduce the architecture of Liqo, focusing on its key components and how they interact to enable multi-cluster capabilities in Kubernetes. Liqo defines the concept of **Peering** and **Offloading** as the two main operations that allow clusters to interact and share resources.

Peering

Liqo defines **Peering**

> as a unidirectional resource and service consumption relationship between two Kubernetes clusters. [2]

Peering is initiated by one cluster and accepted by the other cluster. To establish a successful peering, both clusters must have the Liqo components installed and configured correctly. The peering process involves exchanging information, where two kind of traffic are required:

- `Authentication and Offloading traffic`: Liqo uses a token-based authentication mechanism to securely establish trust between clusters, but provider cluster API server must be reachable by the consumer.
- `Network traffic`: this is required for pod-to-pod and pod-to-service communication. The network fabric is established by exposing a UDP endpoint (through a Service). See Controller Manager.

Offloading

Liqo defines **Offloading**

> as a solution to enable the transparent extension of the local cluster. Is enabled by a virtual node, which is spawned in the local (i.e., consumer) cluster at the end of the peering process, and represents (and aggregates) the subset of resources shared by the remote cluster. [3]

Offloading is a key feature of Liqo that enables clusters to share workloads dynamically based on resource availability and workload requirements. When a cluster offloads a workload, it creates a representation of the workload in the target cluster, allowing it to run seamlessly as if it were native to that cluster.

2.2.2 Components

Liqo consists of several components that work together to enable multi-cluster capabilities. The main components include:

Controller Manager

> The controller-manager contains the control plane of the Liqo network fabric. It runs as a pod (liqo-controller-manager) and is responsible for setting up the network CRDs during the connection process to a remote cluster. [4]

In particular, the **network fabric** constitutes the Liqo subsystem responsible for transparently extending the Kubernetes networking model across multiple independent clusters. This enables offloaded pods to communicate with one another as if they were running within the same local cluster.

The network fabric ensures full connectivity between pods of a local cluster and those of remote peered clusters, supporting both direct communication and communication via Network Address Translation (NAT). Given the heterogeneity of the clusters involved, including differences in configuration parameters and CNI plugins, it is not always possible to avoid overlapping PodCIDR ranges. In such cases, address translation mechanisms are employed, although NAT-less communication is preferred when address spaces are disjoint.

Liqo Gateway

> The Liqo Gateway is a component of the network fabric that runs as a pod on each cluster. It is responsible for managing the VPN tunnels between peered clusters and ensuring secure communication. The gateway also updates routing tables and configures NAT rules to handle address conflicts (i.e. overlapped CIDR).

Virtual Kubelet

> A virtual kubelet replaces a traditional kubelet when the controlled entity is not a physical node. In the context of Liqo, it interacts with both the local and the remote clusters. [3]

Virtual Node

> A virtual node summarizes and abstracts the amount of resources (e.g., CPU, memory, ...) shared by a given remote cluster. Specifically, the

virtual kubelet automatically propagates the negotiated configuration into the capacity and allocatable entries of the node status. [3]

2.2.3 Intra-Cluster Communication

Intra-cluster communication in Liqo is managed through the Kubernetes networking model, which allows Pods to communicate with each other using their IP addresses. Each Pod is assigned a unique IP address, and Kubernetes provides a flat network space where Pods can reach each other without Network Address Translation (NAT). However, a distinct mechanism is employed for external communication, specifically for traffic flowing between the gateway and the individual pods. In this particular scenario, the data packets are not routed directly. Instead, the traffic between the gateway and the pods is orchestrated to first traverse through the node where the target pod resides. From that node, the communication is then securely directed to the pod utilizing Geneve tunnels.

2.2.4 Inter-Cluster Communication

The interconnection between peered clusters is implemented via secure VPN tunnels using WireGuard, which are dynamically established at the end of the peering process based on the negotiated parameters. These tunnels are managed by **Liqo Gateways** components of the network fabric running as pods. Each remote cluster has a dedicated Liqo Gateway pod on both ends of the tunnel. The gateways also update the routing tables with the relevant Liqo custom resources and configure the necessary NAT rules to handle address conflicts, leveraging nftables. The overlay network is used to route all traffic originating from local pods or nodes that is destined for a remote cluster through the gateway, where it enters the VPN tunnel. On the receiving side, traffic exiting the VPN tunnel enters the overlay network to reach the node hosting the target pod. As said before Liqo employs a Geneve-based setup, managed by a network fabric component that runs on every physical node in the cluster as a DaemonSet. This component creates tunnels connecting nodes to gateways. The endpoints of these tunnels correspond to node and pod IP addresses, allowing Liqo to leverage the CNI for connections between nodes and gateways and to benefit from CNI features such as encryption. Additionally, it manages the routing entries on each node to ensure proper traffic forwarding.

2.3 NFTable

Nftables is the new framework for packet filtering and manipulation that replaces the legacy iptables framework and offers a more flexible and efficient way to manage network traffic rules. Nftables uses a single, unified syntax for defining rules, making it easier to manage complex firewall configurations. It supports both IPv4 and IPv6, as well as various protocols and connection tracking features. It operates at the kernel level, allowing it to filter and manipulate network packets before they reach user-space applications. This makes it suitable for implementing high-performance firewalls and network security policies.

2.3.1 Key Features of nftables

- **Unified Rule Syntax**: uses a single syntax for defining rules, making it easier to manage and understand complex configurations.
- **Stateful Filtering**: It supports stateful packet filtering, allowing rules to match packets based on their connection state (e.g., established, related).
- **Sets and Maps**: nftables allows the use of sets and maps to group multiple addresses or ports, simplifying rule management and improving performance.

Advanced Capabilities and Advantages

In addition to its basic features, `nftables` introduces several advanced capabilities that significantly enhance its flexibility and scalability compared to legacy tools like `iptables`:

- **Reduced Rule Duplication**: Through the use of sets and maps, administrators can define large groups of IP addresses, ports, or interfaces, applying rules collectively rather than duplicating similar rules for each element.
- **Improved Performance**: Operating within the kernel's Netfilter framework, `nftables` processes packets efficiently with minimal overhead, which is critical for high-throughput environments or systems handling large volumes of network traffic.
- **Atomic Rule Updates**: Rules can be modified or replaced atomically, reducing the risk of inconsistent states during configuration changes and ensuring uninterrupted traffic filtering during updates.
- **Rich Matching Capabilities**: Beyond traditional matching on IP addresses and ports, `nftables` can inspect packet headers, protocols, interfaces, connection states, and other attributes, enabling fine-grained control over traffic.

- **Scripting and Automation Friendly**: Rules can be defined using declarative configuration files or managed dynamically via the `nft` command-line tool, which simplifies integration with automation tools and custom security solutions.

Comparison with iptables

The transition from `iptables` to `nftables` represents a significant evolution in Linux's packet filtering capabilities, driven by the limitations of `iptables` in modern, complex networking environments. While `iptables` has been a robust solution for many years, its design, which essentially duplicated the filtering logic for IPv4, IPv6, ARP, and bridging, became increasingly cumbersome and inefficient.

`nftables` addresses these challenges by offering a unified syntax and framework. Unlike `iptables`, which required separate utilities and rule sets for different protocols (e.g., `iptables` for IPv4, `ip6tables` for IPv6, `arptables` for ARP, and `ebtables` for Ethernet bridging), `nftables` provides a single command-line tool, `nft`, to manage all aspects of packet filtering. This simplifies rule creation, management, and debugging. For instance, a single `inet` table can handle both IPv4 and IPv6 traffic with one consistent set of rules.

From a performance perspective, `iptables` processes rules linearly, which can degrade performance significantly with large rule sets. In contrast, `nftables` leverages optimized kernel data structures such as sets and maps, which allow for more efficient rule lookups, especially when managing large numbers of IP addresses, ports, or other elements. The rules are compiled into bytecode for a virtual machine running in the kernel, resulting in faster execution.

One of the major advantages of `nftables` is its support for atomic rule updates. With `iptables`, updating rules often required flushing the entire rule set and reloading it, leading to brief moments of inconsistency or insecurity. `nftables` allows users to define a new rule set or make incremental changes, and then apply them as a single atomic operation, eliminating race conditions and ensuring consistency, a critical feature in dynamic environments like Kubernetes.

`nftables` also introduces greater flexibility in rule management. While `iptables` enforces a rigid structure with predefined tables and base chains, `nftables` allows administrators to explicitly define tables, chains, and rules from scratch. It supports key concatenation (e.g., matching both IP address and port in a single condition) and allows a single rule to perform multiple actions (e.g., logging and dropping a packet), simplifying complex configurations.

Advanced features are also better supported. Unlike `iptables`, which relied on external tools like `ipset` for managing large groups of IP addresses, `nftables` includes native support for sets and maps. It also provides enhanced logging options and built-in tracing features that aid in debugging.

Additionally, `nftables` reduces code duplication within the Linux kernel. The protocol-specific design of `iptables` resulted in significant duplication across different filtering utilities, while `nftables` simplifies and unifies the packet classification framework, making the firewalling codebase more maintainable and adaptable to new protocols.

In summary, `nftables` overcomes the limitations of `iptables` by offering a more modern, efficient, and flexible approach to packet filtering. Its unified syntax, improved performance, atomic updates, and support for advanced features make it a better fit for dynamic and large-scale environments, such as cloud-native deployments and federated Kubernetes clusters. Although `iptables` remains in use—often with `nftables` acting as a backend through compatibility layers—`nftables` clearly represents the future of Linux firewalling.

Limitations and Considerations

Despite its many advantages, `nftables` also presents certain limitations. One of the main challenges is the learning curve for system administrators familiar with `iptables`, as the new syntax and configuration model may require adaptation. Furthermore, backward compatibility can be an issue; existing `iptables`-based scripts and tools might need modifications or complete rewrites. Finally, while `nftables` is well-supported in most modern Linux distributions, some legacy systems or third-party tools still rely on `iptables`, potentially requiring hybrid configurations.

Nonetheless, in the context of this thesis, `nftables` has proven to be a powerful tool for enforcing inter-cluster security, providing the flexibility and performance needed to implement dynamic and fine-grained filtering policies in a federated Kubernetes environment.

2.3.2 Types of Chains in `nftables`

A **chain** is an ordered set of rules that is triggered at specific points during packet processing. Chains can be classified into:

1. Base Chains (Hook Chains) These chains are directly linked to kernel hooks, meaning they intercept packets at well-defined points in the networking stack. When creating a base chain, you must specify:

- **Hook**: the interception point (e.g., `prerouting`, `input`, `forward`, `output`, `postrouting`).
- **Priority**: the execution order among multiple chains hooked at the same point.
- **Policy** (optional): the default behavior (e.g., `accept`, `drop`) if no rule matches.

Common hook points:

Hook	Description
`prerouting`	Before routing decision
`input`	Packets destined to local system
`forward`	Packets being routed through the system
`output`	Packets generated locally
`postrouting`	Just before leaving the system

2. Regular Chains (Non-Hook Chains) These chains are internal chains that can be called from other chains using actions like `jump` or `goto`. They are used to modularize rulesets for better readability and reuse.

Difference between `jump` and `goto`:

- `jump`: After executing the called chain, return to the next rule in the calling chain.
- `goto`: After executing the called chain, continue processing from the next rule in the target chain.

Types of Rules in `nftables`

Rules consist of conditions and actions applied to packets. Each rule contains:

- **Match conditions**: Filters based on packet attributes, such as:
  - Source/destination IP addresses

  - TCP/UDP ports
  - Protocol types (TCP, UDP, ICMP, etc.)
  - Network interface
  - Connection tracking state (`conntrack`)
  - TCP flags (e.g., SYN, ACK)
- **Actions**: What to do if the condition matches, for example:
  - `accept`: allow the packet
  - `drop`: silently discard the packet
  - `reject`: discard the packet with an error
  - `log`: record the packet information in the system log
  - `counter`: increment a counter for matched packets
  - `jump`/`goto`: transfer processing to another chain

2.3.3 Nftables in Multi-Cluster Kubernetes Security

The system presented in this thesis leverages `nftables` as a fundamental component to enforce network-level isolation and security policies within federated Kubernetes environments. Specifically, `nftables` is used to:

- Implement strict filtering rules on gateway pods responsible for inter-cluster traffic, ensuring that only authorized connections between clusters are permitted.
- Dynamically adapt filtering policies based on changes in the multi-cluster topology, such as the offloading of workloads or the establishment of new peering relationships via Liqo.
- Allowing for disjoint address spaces through NAT in gateway pods, enabling communication between Pods in different clusters without IP address conflicts.

Thanks to its kernel-level efficiency, extensibility, and dynamic management capabilities, `nftables` is particularly well-suited to the requirements of secure, distributed Kubernetes deployments, where maintaining control over traffic boundaries across clusters is essential.

2.3.4 Nftables and Liqo

By default Liqo uses nftables to enforce NAT rules for cross-cluster communication. This allows Pods in different clusters to communicate with each other while maintaining network isolation. Currently, Liqo can operate with `nftables` in **NAT-less** or **NAT** mode, meaning, in the first case, that it does not perform any address translation for offloaded Pods. This is possible because the offloaded Pods are placed in dedicated namespaces, which allows for disjoint address spaces across clusters. However, this mode requires careful management of network policies to ensure that traffic is correctly routed and filtered without overlapping IP addresses.

Liqo introduces also the support for NAT-based offloading, in case where the offloaded Pods share the same address space as the local cluster (overlapped CIDR). In this case, nftables is used to implement NAT rules that allow Pods in different clusters to communicate without IP address conflicts. This is particularly useful in scenarios where clusters have overlapping PodCIDR ranges or when offloaded workloads need to interact with local services.

In order to enforce NAT rules, Liqo uses NFTables Chains of type pre-routing, post-routing, and output. These chains are responsible for handling the traffic that enters and exits the cluster, ensuring that packets are correctly translated and routed to their intended destinations. The rules in these chains are dynamically generated based on the peering relationships and offloaded workloads, allowing for flexible and adaptive network policies. The implementation chapter will explore the implementation of chain type `forward` and how nftables is used in conjunction with Liqo to implement fine-grained network isolation policies that adapt dynamically to changes in the cluster topology and workload placement by applying network filtering rules.

Chapter 3

The Problem of Interest

Modern Kubernetes deployments increasingly rely on multi-cluster architectures to improve scalability, resilience, and resource utilization. In this context, Liqo enables seamless workload offloading by allowing a "consumer" cluster to run its Pods on a remote "provider" cluster without modifying application logic or networking configurations. While this abstraction offers flexibility and resource sharing, it introduces critical security and isolation challenges: Offloaded Pods are executed within the provider cluster but originate from and belong to the consumer cluster; Liqo automatically establishes network peering between the clusters, exposing data plane connectivity across administrative boundaries. By default, all Pods in the consumer cluster, not just those offloaded, may gain network-level access to remote resources within the provider cluster.
This default behavior can lead to unauthorized cross-cluster access, especially problematic in multi-tenant scenarios or collaborative environments where clusters are owned by different organizations or departments.

Without proper isolation, a consumer cluster could interact with offloaded Pods or even sensitive components belonging to other consumers within the same provider cluster. This highlights the necessity of fine-grained network policies and explicit traffic control mechanisms to ensure that: the consumer cluster only accesses the Pods and namespaces it has offloaded. Offloaded Pods are reachable from their source cluster but isolated from other consumers.

3.1 Enforcing Isolation at the Peering Gateway Level

To effectively control cross-cluster communication in a Liqo-enabled environment, it is not sufficient to apply Kubernetes-native constructs like NetworkPolicies alone. These policies control traffic at the IP and port level (OSI layer 3 and 4), but they have limitations in enforcing strict isolation across clusters, especially for inter-cluster traffic. In particular, NetworkPolicies are not designed to handle the complexities of multi-cluster networking, where Pods from different clusters may need to communicate while still maintaining strict boundaries.

3.1.1 Why Not (only) NetworkPolicies?

NetworkPolicies in Kubernetes are primarily designed to manage traffic within a single cluster, focusing on Pod-to-Pod communication based on labels and selectors. However, in a multi-cluster setup like Liqo, where Pods from different clusters can be offloaded and executed in a provider cluster, NetworkPolicies alone are insufficient for several reasons:

- **CNI Dependecy**: NetworkPolicies rely on the underlying Container Network Interface (CNI) plugin to enforce rules. Not all CNI plugins support advanced features required for cross-cluster isolation, leading to inconsistent behavior.
- **Single-Cluster Scope**: NetworkPolicies are scoped to a single cluster and do not inherently understand the context of offloaded Pods from other clusters. This makes it difficult to enforce policies that span multiple clusters.
- **Node-Level Traffic**: Pods can always communicate with each other at the node level, bypassing NetworkPolicies. This is particularly problematic in a multi-cluster setup where Pods from different clusters may share the same nodes in the provider cluster. [5]

Additionally, Liqo encapsulates cross-cluster communication through its peering gateways, which handle the routing of traffic between clusters. This abstraction layer makes invisible to some CNIs, and thus to NetworkPolicies, the actual source and destination of the traffic, complicating the enforcement of isolation rules. From tests done, Cilium CNI is able to enforce NetworkPolicies for offloaded Pods, but this is not the case for all CNIs. For example, Calico does not support this feature, leading to potential security risks in multi-cluster deployments. The solution proposed includes using nftables-based packet filtering directly on the Liqo gateways in the provider cluster, ensures precise traffic filtering, enforcing security before packets reach the node's network stack.

3.2 Case Studies

This work focuses on analyzing the security challenges related to **Provider Protection**, where a consumer cluster offloads Pods to a provider cluster, and strict traffic control is required to avoid unintended interactions with the provider environment. In particular, offloaded Pods may need to access specific services on the provider side (e.g., networking or storage components exposed by the provider), but they must be prevented from interacting with other Pods or namespaces unrelated to the consumer cluster. To better frame this problem, consider the following scenarios:

- **Provider Protection (Focus of this thesis)**: A consumer cluster offloads Pods to a provider cluster. The offloaded Pods are allowed to access only selected resources within the provider environment. Communication towards other Pods or namespaces that do not belong to the same consumer cluster must be strictly prohibited to prevent lateral movement or unintended access to shared infrastructure.
- **Consumer Isolation in Multi-Tenant Environments**: Multiple consumer clusters offload Pods to the same provider cluster. In this scenario, additional safeguards are needed to ensure that Pods belonging to different consumers remain completely isolated from each other.
- **Consumer Protection (Out of Scope)**: A consumer cluster offloads Pods to a provider cluster, but requires mechanisms to prevent the provider cluster from initiating unsolicited communication towards the consumer's resources.
- – **Consumer Protection Leaf-to-Leaf Traffic (Out of Scope)**: A consumer cluster offloads Pods to a provider cluster, but requires mechanisms to prevent Pods offloaded to different provider clusters from communicating with each other. This is particularly relevant in scenarios like federated learning, where strict isolation between offloaded Pods is required.
- **Internet Protection (Out of Scope)**: A consumer cluster offloads Pods to a provider cluster, but requires mechanisms to redirect traffic from the provider cluster to the internet, ensuring that offloaded Pods can only access external resources through controlled gateways, or come back to the consumer cluster.

This thesis provides a detailed design and implementation of **Provider Protection**, proposing a solution to enforce strict network isolation for offloaded workloads on the provider side. The other scenarios are acknowledged for completeness but are left as future work.

3.2.1 Provider Protection

Consumer cluster can only contact their offloaded pods in the provider cluster

For the sake of simplicity in this scenario, we will consider a single consumer cluster that offloads Pods (and the relative namespaces) to the provider cluster. The key requirement is to ensure that consumer cluster can only access its own offloaded resources, without any possibility of interacting with non-offloaded Pods or namespaces within the provider cluster.

The proposed solution enforces traffic filtering directly at the gateway level, ensuring that:

- The consumer cluster can only reach the Pods and namespaces offloaded by itself.
- Traffic from the consumer cluster to the provider cluster is restricted to offloaded Pods and namespaces, preventing access to any other resources in the provider cluster.
- Offloaded Pods can communicate with each other, but they are isolated from non-offloaded Pods in the provider cluster.

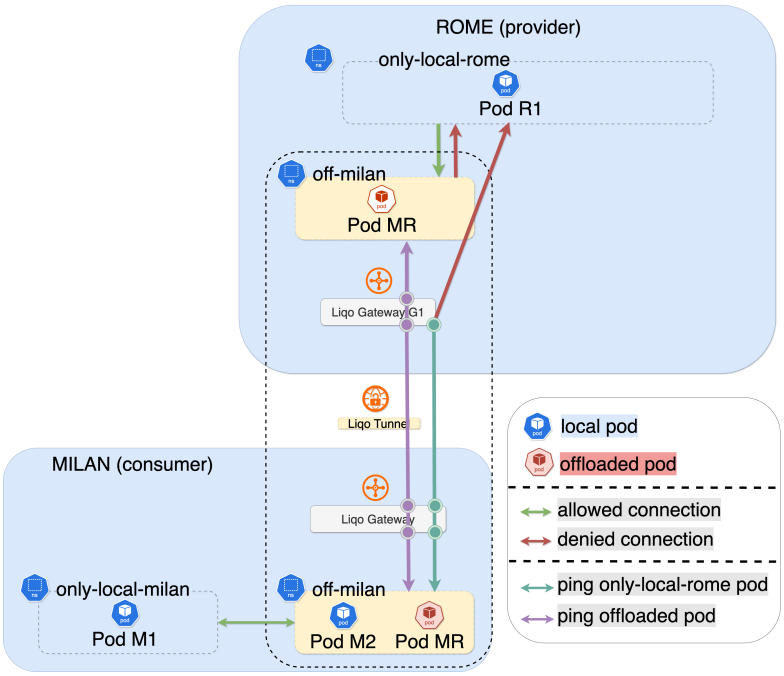

Figure 3.1: Provider Protection Scenario

Green arrows indicate allowed traffic within the same consumer's offloaded resources. Red arrows indicate blocked traffic between different consumers. Dotted arrows represent logical traffic paths.

This level of protection guarantees strict tenant isolation in shared provider environments, preventing unauthorized cross-consumer access while maintaining full functionality for legitimate offloaded workloads.

Note that the `Pod M1` in the namespace `only-local-milan` will not be able to communicate with any other pod in the provider cluster, as it is not offloaded to the provider cluster. This is a key aspect of the isolation mechanism, ensuring that only Pods explicitly offloaded by the consumer cluster can interact with the provider's resources. So, specifically, each traffic that traverse the 'liqo tunnel' will be filtered by the gateway pod.
Another different case is when the consumer performs an 'exec' command on an offloaded Pod, which is allowed by default. In this case, the consumer cluster can interact with the offloaded Pod, and the traffic from the offloaded Pod travels directly in the provider cluster's network stack, bypassing the gateway pod. This is the case when the provider cluster needs to enforce Network Policies on the offloaded namespace (see implementation).

Provider Protection Feature: Consumer Isolation in Multi-Tenant Environments

In this scenario, multiple consumer clusters offload Pods to the same provider cluster. The key requirement is to ensure that each consumer cluster can only access its own offloaded resources, without any possibility of interacting with Pods or namespaces belonging to other consumer clusters within the same provider cluster. This setup represents a common multi-tenant environment, where different organizations or departments share the same provider cluster for workload offloading but require strict isolation of their respective workloads for security and compliance reasons.

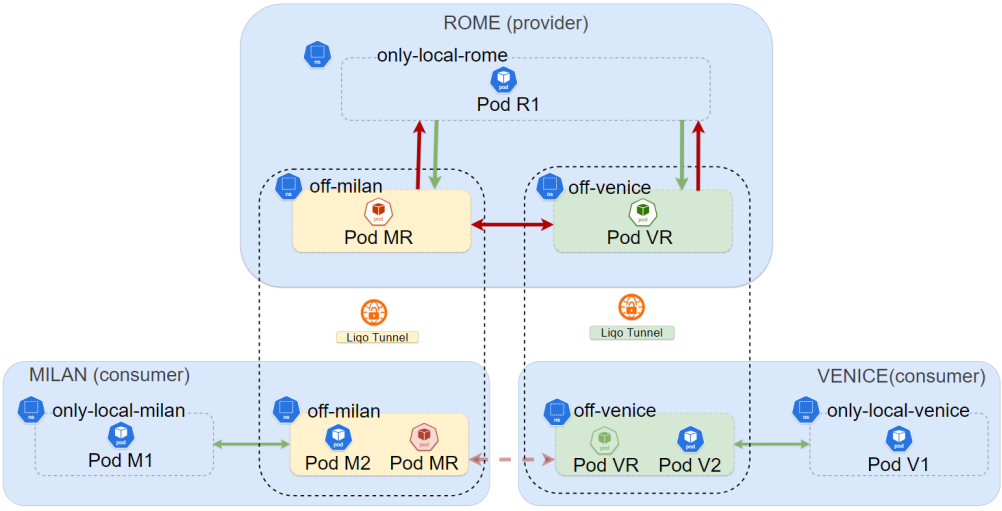

Figure 3.2: Provider Protection (multi-tentant) Scenario

The proposed solution enforces traffic filtering directly at the gateway level, (not shown in the figure) but the results are the same as in the previous scenario. In this case, arrows represent the allowed traffic between offloaded Pods and namespaces, while red arrows indicate blocked traffic between different consumers. The dotted arrows represent logical traffic paths, showing how offloaded Pods can communicate with each other within the same consumer cluster but are isolated from Pods belonging to other consumers. Note that the traffic is blocked in the gateway pod, before it reaches the provider cluster's network stack, ensuring that no unauthorized traffic can traverse the provider cluster's infrastructure.

3.2.2 Consumer Protection

Provider pods cannot contact consumer pods in non-offloaded namespaces

With this feature enabled, Pods running in the provider cluster outside of offloaded namespaces are prevented from initiating connections to consumer cluster Pods. Only offloaded Pods are allowed to communicate with consumer Pods, and even in that case, communication is restricted to Pods belonging to the same offloaded namespace, regardless of whether they are offloaded to different provider clusters (i.e., leaf-to-leaf traffic is allowed). A practical example of this requirement is applications like database replication, where each replica (potentially offloaded to different provider clusters) must be able to communicate with other replicas in the same namespace (for example, in a full-mesh topology), while strict isolation from other namespaces is preserved. This mechanism is conceptually similar to the provider protection feature but is applied on the consumer cluster, ensuring that its Pods are shielded from unwanted or unauthorized traffic originating from provider clusters. The technical implementation is expected to mirror the approach used for provider protection, likely leveraging the same components and logic.

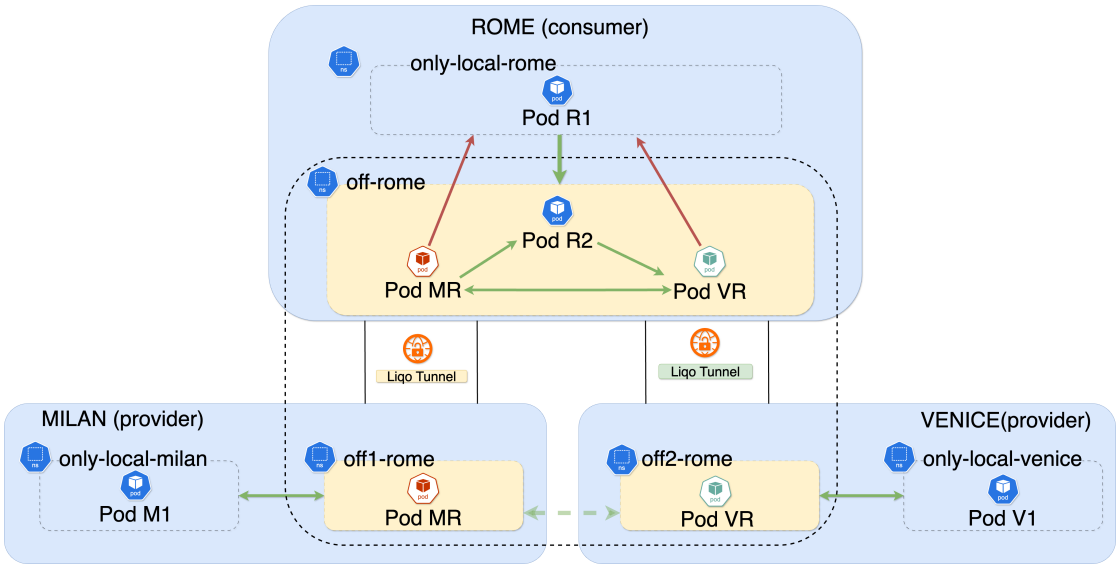

Figure 3.3: Consumer Protection Scenario

3.2.3 Consumer Protection (2)

Disable Leaf-to-Leaf Traffic

With this feature enabled, the consumer cluster blocks all traffic between pods offloaded to different provider clusters (commonly referred to as "leaf-to-leaf" traffic). This extends the first consumer protection mechanism, where leaf-to-leaf communication was still permitted. Such an approach is suitable for use cases where pods running on different providers should remain fully isolated from each other. A typical example is federated learning, where client pods are offloaded to different providers, and strict policies require that these clients cannot communicate directly with one another — they should only exchange data with a central server pod running on the consumer cluster.

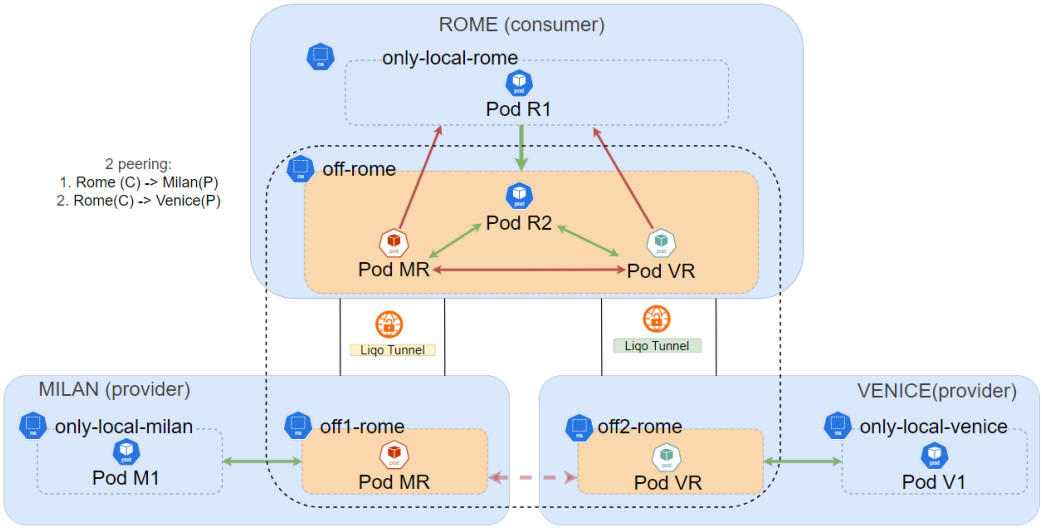

Figure 3.4: Provider Protection Scenario

Chapter 4

Implementation Overview

The implementation of the proposed solution involves several key components and steps to ensure effective isolation and security in Liqo-enabled multi-cluster environments. The main focus is on leveraging nftables for packet filtering at the peering gateway level, in particular the actual implementation will be based on first case study, where the provider cluster is protected from unauthorized access by consumer clusters.

As said before, the consumer cluster offloads pods to the provider cluster, which needs to access specific resources in the provider cluster, but it should not interact with other Pods or namespaces that do not belong to the consumer cluster. The aim is to avoid completely the traffic between the two clusters that is not related to the offloaded Pods, while allowing the traffic between offloaded namespaces by the same consumer cluster.

4.1 Components

Before delving into the implementation details, it is essential to understand the components involved in the proposed solution. The architecture consists of two main components: the Network Security Engine and the Liqo Security Agent. These components work together to enforce network security policies in a Liqo-enabled multi-cluster environment.

Network Security Engine |NSE|

A controller that applies generic nftables rules to the pods it runs on, or to the underlying node in case the pod is using hostNetwork. These rules are defined through a `Custom Resource Definition` called `FirewallConfiguration`. The controller monitors these CRs, translates the specified rules into nftables format, and enforces them accordingly on the pod or node. The FirewallConfiguration CRD is already available within the Liqo project; however, its API does not yet support defining filtering rules. To achieve the desired functionality, the API needs to be extended to cover the main nftables filtering capabilities. In this document, we use the term Network Security Engine to refer to the combined system consisting of the extended FirewallConfiguration API and the controller responsible for enforcing the rules. This component is designed to be integrated into the open-source Liqo project as part of this thesis work.

Liqo Security Agent |LSA|

It is a component responsible for enforcing various security features by generating and applying all the required `FirewallConfiguration` and `NetworkPolicy` custom resources to the cluster. Once these resources are created on the cluster's `API server`, the `Network Security Engine` reconciles the FirewallConfiguration CRs,translates them into `nftables` rules, and applies them to the pod or node where it is running. Enforced by the `CNI plugin` installed on the cluster.

4.1.1 High-Level design

To provide a comprehensive understanding of the current architecture, we can briefly outline the general workflow, which is visually summarized in the following image:

The proposed solution is structured around a clear separation of concerns, involving multiple components that operate at different stages of the process to ensure the correct application of security policies across the distributed environment, specifically leveraging `nftables` in a Kubernetes and Liqo context.

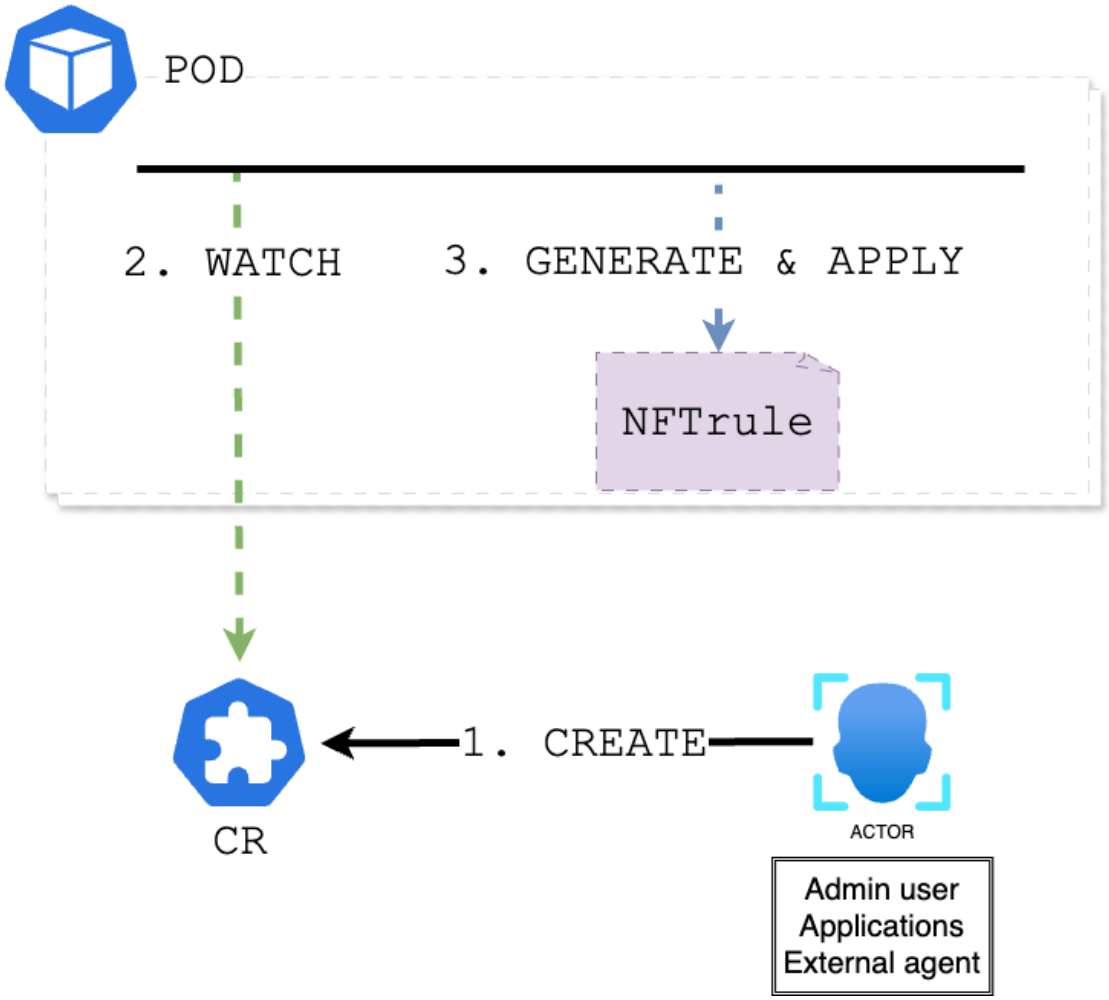

Figure 4.1: High-Level Design Architecture

In the **first phase (1. CREATE)**, an **Actor**, which can be an admin user, an application, or an external agent (i.e. LiqoSecurityAgent), is responsible for defining the desired network policy. This high-level policy is then translated into a **Custom Resource (CR)** and applied to the Kubernetes cluster. This **CR** acts as a declarative representation of the security requirements, abstracting away the low-level networking details.

Once the **CRs** have been generated and applied to the Kubernetes cluster, a dedicated component, represented by the **POD** in the diagram, continuously monitors these resources. This component, acting as a controller, performs a **second phase (2. WATCH)** operation on the **CRs**. As part of its reconciliation loop, it compares the desired state, as declared in the **CRs**, with the current system state. Upon detecting discrepancies, new configurations, or updates to

existing policies, it proceeds to **third phase(3. GENERATE & APPLY)** the corresponding `NFTrules`. These `NFTrules` represent the specific `nftables` commands required to implement the desired network policy.

These `NFTrules` are then programmatically applied (i.e. $|\underline{\text{NSE}}|$) to the relevant network nodes within the cluster, enforcing fine-grained control over inter-cluster and intra-cluster traffic. This includes, for example, selectively allowing or denying communication between offloaded workloads, namespaces, or entire clusters, based on the user-defined policies encapsulated within the **CRs** and translated into `NFTrules`.

The separation between the high-level **CR** creation by the **Actor** and the low-level controller within the **POD** that generates and applies `nftables` rules (`NFTrules`) ensures both modularity and scalability. It allows for the gradual evolution of the system, where additional logic can be incorporated into the **CR** generation process without requiring modifications to the enforcement layer. This approach also aligns with Kubernetes-native design principles, leveraging Custom Resources and controllers to manage complex system behavior declaratively, while providing a robust mechanism for dynamically adapting to changes in a multi-cluster environment, particularly relevant for distributed systems managed by Liqo.

4.1.2 Network Security Engine

The goal is to extend the CRD (networking.liqo.io/v1beta1/FirewallConfiguration) to define filtering network policies, allowing rules based on IP, IP ranges, ports, CIDR, and protocols. As we know, the firewall configuration controller reconciles FirewallConfiguration CRs, which describe (along with others) the RulesSet that has to be applied (natRule, routeRule, filterRule). More specifically, in the context of filterRule, three different filterAction should be implemented: drop, accept, reject (in addition to the existing one (ctmark)).

```
chains:           # List of firewall chains
  - type: filter  # Chain type
    rules:        # List of rules
      - filterRules:
          - name: drop_traffic          # Rule name
            action: drop                # Action to apply (drop,
    accept, reject)
            match:                      # Defines match conditions
              ip.value:                 # IP to match
                - "10.10.1.52"          # Single IP
                - "10.10.1.52-10.10.1.53"  # Range
                - "10.10.0.0/24"        # CIDR
            counter: true               # (Optional, default true)
    Tracks rule triggers
```

Listing 4.1: Example of FirewallConfiguration CR with filterRule

api_fields_extended

- **action**: drop, accept, reject (in addition to the existing one (ctmark))
- **match_ip_value**: range support
- **counter — optional (default true)**: tracks the number of times the rule is triggered

Example

Implementation Example: Blocking Inter-Cluster Traffic

The following image illustrates the concept in a real scenario, where `nftables` is used to filter traffic between two clusters, Cluster1 (Consumer) and Cluster2 (Provider), with a focus on the peering gateway.

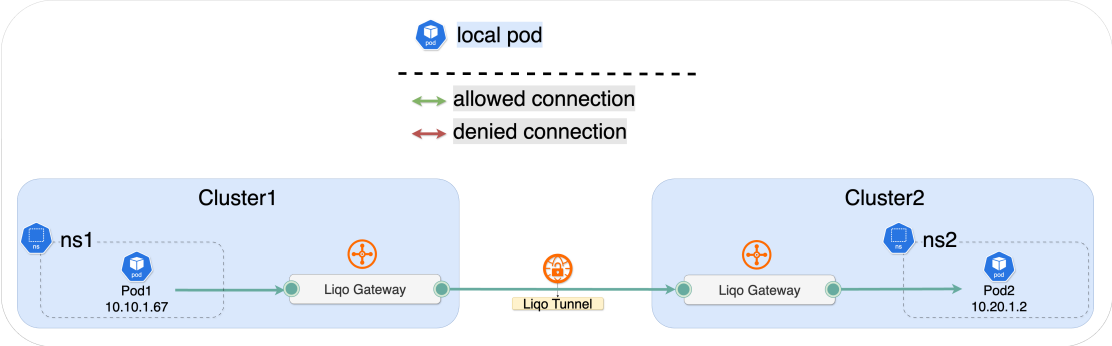

Figure 4.2: High-Level Design Architecture of the Security Agent

For the sake of simplicity, the component that will run the firewall configuration controller will be the Liqo Gateway Pod on Cluster2. The provided image illustrates Liqo's default behavior. The topology shows two clusters, Cluster1 and Cluster2, which have been peered using Liqo. Green arrows indicate permitted connections between pods; consequently, in the default behavior, all pods belonging to different clusters can communicate with each other without any restrictions. For example, `Pod1` (IP address `10.10.1.67`) on `Cluster1` can directly ping `Pod2` (IP address `10.20.1.2`) on `Cluster2`.

Our objective is to block traffic between `Pod1` (`Cluster1`) and `Pod2` (`Cluster2`) by applying a `FirewallConfiguration` and specifying the related IP addresses. The applied `FirewallConfiguration` is as follows:

```
1
2  table:
3      name: "table_name"
4      family: "IPv4"
5      chains:
6        - hook: "forward"
7          name: "filter_chain_name"
8          policy: "accept"
9          priority: 0
10         type: "filter "
11         rules:
12           filterRules:
13             - name: "drop_traffic"
14               counter: true
```

```
15                action: "drop"
16                match:
17                  - ip:
18                      value: "10.10.1.67"
19                      position: "src"
20                    op: "eq"
21                  - ip:
22                      value: "10.20.1.2"
23                      position: "dst"
24                    op: "eq"
```

Listing 4.2: Example of FirewallConfiguration CR with filterRule

It's important to note that reverse traffic (from `Pod2` to `Pod1`) could be blocked by adding another `match` rule with the same IP values but with the positions (`position`) inverted (i.e., `src` for `10.20.1.2` and `dst` for `10.10.1.67`).

As a result of this implementation, we block traffic originating from `Pod1` in `Cluster1` directed towards `Pod2` in `Cluster2`. The effect of these enforced security policies is illustrated by the red arrow, indicating a denied connection. For example, `Pod1` (`10.10.1.67`) on `Cluster1` will no longer be able to ping `Pod2` (`10.20.1.2`) on `Cluster2`, as the security policy prevents such communication.

Result of the implementation after the FirewallConfiguration CR is applied

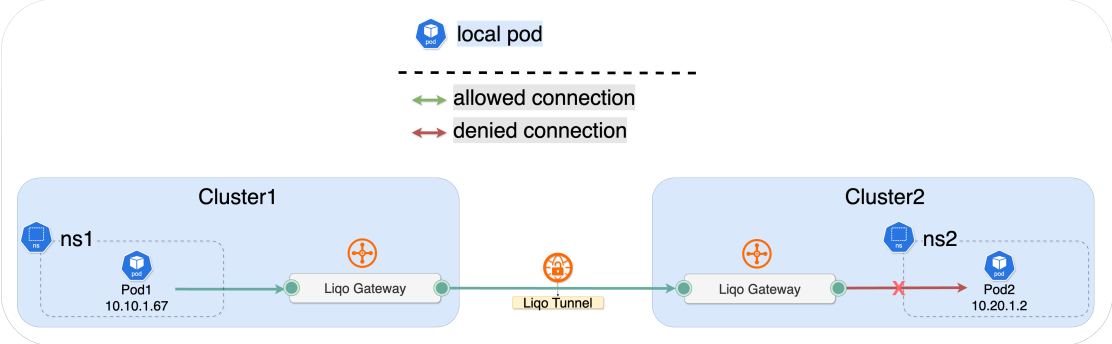

Figure 4.3: High-Level Design Architecture of the Security Agent

4.1.3 NSE Sub controller

The Network Security Engine sub-controller is responsible for applying the initial configuration to the gateway pod. This configuration is crucial for establishing a default-deny policy, which blocks all traffic from the consumer cluster to the provider cluster. The sub-controller will create a `FirewallConfiguration` CR that explicitly denies all incoming traffic from the consumer cluster, ensuring that no unauthorized access is allowed until further rules are applied by the Liqo Security Agent. From the moment the `FirewallConfiguration` CR is applied, the gateway pod will enforce this default-deny policy, effectively preventing any traffic from the consumer cluster from reaching the provider cluster's network stack. After this initial setup, the Liqo Security Agent will take over to apply more specific rules that allow only the necessary traffic types, as defined in the provider protection feature, adding the IP addresses of offloaded pods to the `FirewallConfiguration` CRs. This ensures that the provider cluster remains secure while still allowing legitimate traffic from offloaded pods.

```
1
2  table:
3      name: "table_name"
4      family: "IPv4"
5      chains:
6        - hook: "forward"
7          name: "filter_chain_name"
8          policy: "deny"
9          priority: 0
10         type: "filter"
```

Listing 4.3: Default deny FirewallConfiguration CR

4.1.4 Liqo Security Agent

The Liqo Security Agent is a pivotal component responsible for implementing Liqo's security features. It achieves this by forging and applying the necessary `FirewallConfiguration` and `NetworkPolicy` Custom Resources (CRs) to enforce security policies on the cluster.
A fundamental assumption underpinning the agent's design is that it exclusively safeguards its own cluster. It is not tasked with protecting other clusters, nor can it make assumptions about the security features enabled on remote clusters, given the inherent trust boundaries in this distributed environment.

Providing a specific example of such policy application would be redundant or misleading. Given the abstract and declarative nature of Custom Resources (CRs) like Network Policies, the enforcement mechanism is agnostic to the underlying application context. The agent focuses on generating and applying the necessary CRs, which, being based on Kubernetes standards (or extensions defined via CRDs), are universally applicable in any Kubernetes environment capable of interpreting and enforcing such definitions. This ensures inherent flexibility and broad compatibility, rendering a contextualized example for a particular scenario superfluous.

Specifically for the traffic that is not passing through the Liqo gateway, the agent will generate `NetworkPolicy` CRs to enforce security policies on offloaded pods. These policies will be applied to the namespaces offloaded by the consumer cluster, ensuring that only traffic originating from the same consumer cluster is allowed to reach its offloaded pods. While the traffic that passes through the Liqo gateway is managed by NFTables rules, directly on the gateway pods; the agent will focus on generating the necessary `FirewallConfiguration` CRs to enforce the provider protection feature.

For the scope of this thesis, the Liqo Security Agent will focus solely on implementing the **Provider Protection feature**. However, its architectural design facilitates future extensions to incorporate other security features (e.g., consumer protection).

4.1.5 Webhook

The Webhook is a crucial component that intercepts requests to the Kubernetes API server, allowing for dynamic validation and mutation of resources before they are persisted in the cluster. In the context of the Liqo Security Agent, the Webhook plays a vital role in ensuring that only valid and secure configurations are applied to the cluster.

When a new `FirewallConfiguration` or `NetworkPolicy` CR is created or updated, the Webhook intercepts these requests and performs validation checks. It ensures that the specified rules conform to the expected format and semantics, preventing misconfigurations that could lead to security vulnerabilities or network disruptions. In order to implement the Network Security Engine, the Webhook in Liqo must be extended to support the new `FirewallConfiguration` CRD. This extension will involve defining the schema for the CRD, including the necessary fields and validation rules. The Webhook will then be responsible for validating incoming requests against this schema, ensuring that only well-formed and secure configurations are accepted. In particular more specific validation rules about the previously mentioned `FirewallConfiguration` CRD should be implemented, such as:

- **Action Validation**: Validate that the specified actions (e.g., `drop`, `accept`, `reject`) are supported and correctly applied. This prevents the application of unsupported or potentially harmful actions.
- **IP Address Validation**: Ensure that the IP addresses specified in the `FirewallConfiguration` CR are valid and correctly formatted. This includes checking for valid IPv4, ranges, and CIDR notations.
- **Counter Validation(Optional)**: Validate the presence and correctness of the `counter` field, ensuring it is set to a boolean value. This field is crucial for tracking rule triggers and should be correctly configured.

4.2 Provider Protection Feature

The core concept behind the Provider Protection feature is to enable provider clusters to shield their pods from undesirable or unauthorized traffic originating from consumer clusters. Specifically, the provider cluster will explicitly permit **ONLY** the following categories of traffic:

1. Traffic originating from the consumer cluster (which passes through the Liqo gateway) and destined for pods residing in the namespaces offloaded by that consumer cluster.
2. Traffic between offloaded pods within the provider cluster that are owned by the same consumer cluster (i.e., pods that have been offloaded from the same consumer cluster).
3. Traffic originating from local (non-offloaded) pods on the provider cluster.

Let's delve into the implementation details for the first two aforementioned traffic types:

1) Traffic Originating from the Consumer Cluster (passing through the Liqo Gateway)

The provider cluster strictly permits traffic originating from a consumer cluster (and traversing the Liqo gateway) only when it is specifically destined for pods located within the namespaces that were offloaded by that particular consumer cluster. Any other traffic from the consumer cluster is unequivocally blocked.

This blocking mechanism is enacted directly at an early stage within the gateway through custom `FirewallConfiguration` rules. These rules are dynamically forged by the Liqo Security Agent and enforced by the Network Security Engine. Relying on Kubernetes `NetworkPolicies` for this specific type of traffic presents challenges, as some CNI plugins (e.g.,Cilium/Calico) may not be able to adequately inspect or enforce policies on traffic encapsulated within the Liqo tunnel (which utilizes Geneve tunnels). While Cilium CNI appears to offer some capabilities in this domain, the precise enforcement point and overall reliability remain unclear. By explicitly blocking this traffic with `nftables` rules, we achieve two significant advantages:

- **CNI Agnosticism**: We eliminate dependencies on the specific CNI plugin deployed and its particular implementation of `NetworkPolicies`.
- **Early Traffic Termination and Reduced Attack Surface**: Traffic is halted at an early stage within the gateway, preventing packets from unnecessarily traversing the cluster within Geneve tunnels. This strategy significantly

reduces the overall attack surface and mitigates potential Denial of Service (DoS) attacks.

2) Traffic Between Offloaded Pods in the Provider Cluster

Offloaded pods are designed to be able to communicate with each other, provided they originate from the same consumer cluster. Since an offloaded namespace is exclusively owned by a single consumer cluster, Kubernetes `NetworkPolicies` can be effectively leveraged for this purpose. The Liqo Security Agent will generate a `NetworkPolicy` CR for each offloaded namespace within the provider cluster. These policies will specifically permit egress traffic only to pods residing in namespaces that are also owned by the same consumer (i.e., where the tenant namespace label matches the consumer cluster ID).

Visual Representation of the Provider Protection Feature (1)

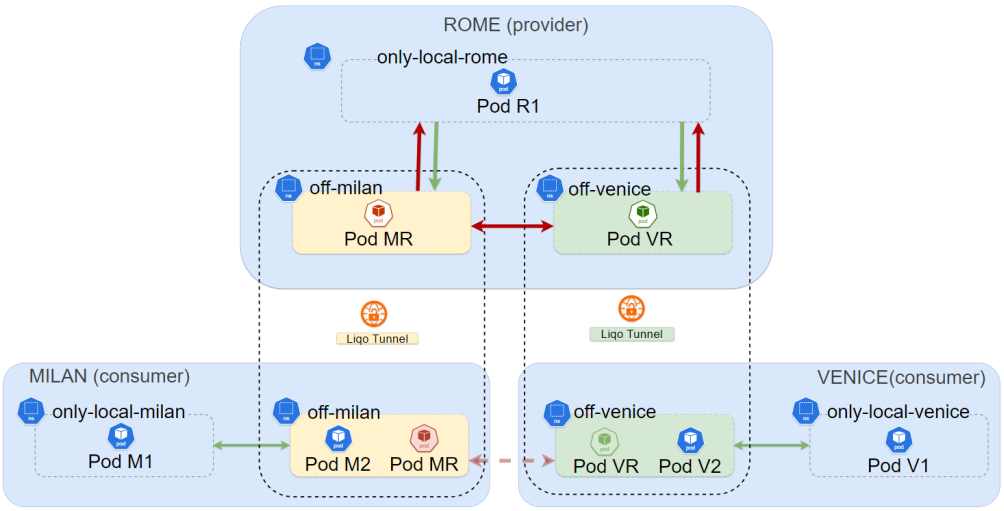

Figure 4.4: Multiple consumer clusters offloading single namespace to a Provider Cluster

The image illustrates the Provider Protection feature, where multiple consumer clusters offload a single namespace to a provider cluster. The provider cluster enforces strict traffic control, allowing only specific traffic types as described above.

Visual Representation of the Provider Protection Feature (2)

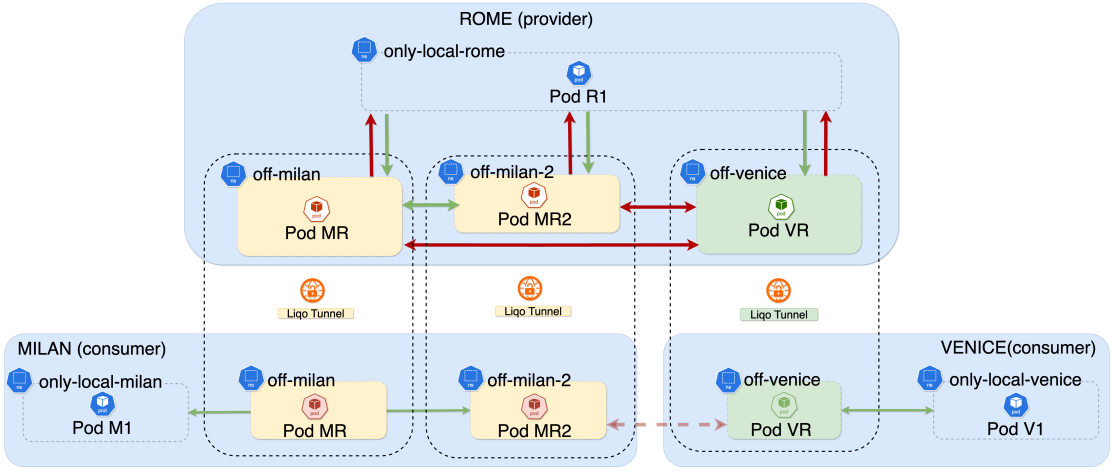

Figure 4.5: Consumer clusters offloading multiple namespace to a Provider Cluster

The image illustrates the Provider Protection feature, where one cluster offloads multiple namespaces to a provider cluster, while the other offloads just one namespace. The provider cluster enforces strict traffic control, allowing only specific traffic types as described above.

In this particular scenario, the provider cluster is designed to allow traffic from the consumer cluster only to the offloaded pods in the namespaces that belong to that specific consumer cluster. This means that even if multiple namespaces are offloaded by different consumer clusters, each namespace is isolated from others, while traffic between offloaded pods by the same consumer cluster is allowed.

4.2.1 Implemented Controllers

To bring this feature to fruition, three dedicated controllers have been implemented:

Gateway Controller

This controller performs reconciliation on gateway pods.
Its primary function is to apply a `FirewallConfiguration` CR that initially blocks *all* traffic arriving from the WireGuard gateway. This establishes a strict default-deny policy, ensuring that no traffic from the consumer cluster is permitted into the provider cluster unless explicitly whitelisted by the subsequent controller. As shown in the previous section, Figure 3.1, in the Liqo Gateway Pod the NFTable forged will be:

```
1 table inet liqo_gateway {
2     chain input {
3         type filter hook input priority 0; policy drop;
4     }
5 }
```

Listing 4.4: Initial NFTable configuration applied by the Gateway Controller

This controller in necessary to ensure that even if the Liqo Security Agent is running and a new Liqo peering is established, the provider cluster will not allow any traffic from the consumer cluster until the Liqo Security Agent applies the necessary `FirewallConfiguration` CRs to allow specific traffic types. This is crucial for maintaining security and preventing unauthorized access to the provider cluster's resources. A visual representation of the Gateway Controller's role in the provider protection feature is shown in the following image, where the `liqo_gateway` table has been forged in `Liqo Gateway G1`:

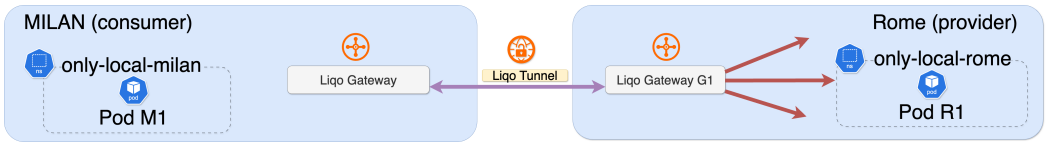

Figure 4.6: Gateway Controller's Role in Provider Protection Feature

Gateway Controller Code

Following is reported the code of the reconcile function of the Gateway Controller, which is responsible for applying the initial `FirewallConfiguration` CR to the Liqo Gateway Pod. This code is written in Go and uses the controller-runtime library to interact with the Kubernetes API server.

```
1   // Reconcile is the main logic triggered for gateway pods.
2  func (r *FirewallStartupReconciler) Reconcile(ctx context.Context,
       req ctrl.Request) (ctrl.Result, error) {
3    var pod corev1.Pod
4    var err error
5
6    if getErr := r.Get(ctx, req.NamespacedName, &pod); getErr != nil
       {
7      klog.Errorf("Can't find pod %s: %v", req.NamespacedName,
      getErr)
8      return ctrl.Result{}, fmt.Errorf("failed to get pod: %w",
      client.IgnoreNotFound(getErr))
9    }
10
11   clusterID := pod.Labels[consts.K8sAppNameKey]
12   if clusterID == "" {
13     klog.Warningf("Pod without label %s: missing label key %s",
      pod.Name, consts.K8sAppNameKey)
14     return ctrl.Result{}, nil
15   }
16
17   fwcfg := &firewall.FirewallConfiguration{
18     ObjectMeta: metav1.ObjectMeta{
19       Name:      "block-traffic-rules-" + clusterID,
20       Namespace: "liqo",
21     },
22   }
23
24   mutateFn := r.mutateFn(clusterID, fwcfg)
25
26   result, err := controllerutil.CreateOrUpdate(ctx, r.Client,
      fwcfg, mutateFn)
27   if err != nil {
28     klog.Errorf("Error creating or updating firewall configuration
       for cluster %s: %v", clusterID, err)
29     return ctrl.Result{}, fmt.Errorf("%w: %w", errors.
      ErrFailedCreateFirewallConfig, err)
30   }
31
32   switch result {
33   case controllerutil.OperationResultCreated:
```

```
34      klog.Infof("Firewall configuration created for cluster: %s",
        clusterID)
35    case controllerutil.OperationResultUpdated:
36      klog.Infof("Firewall configuration updated for cluster: %s",
        clusterID)
37    case controllerutil.OperationResultNone:
38      klog.Infof("No changes made to the firewall configuration for
        cluster: %s", clusterID)
39    case controllerutil.OperationResultUpdatedStatus:
40      klog.Infof("Firewall configuration status updated for cluster:
        %s", clusterID)
41    case controllerutil.OperationResultUpdatedStatusOnly:
42      klog.Infof("Only the status of the firewall configuration was
        updated for cluster: %s", clusterID)
43    }
44
45    klog.Infof("FirewallConfiguration for cluster: %s updated",
        clusterID)
46    return ctrl.Result{}, nil
47  }
```

Listing 4.5: Gateway Controller Code

ShadowPod Controller

This controller reconciles on shadow pods. Its role is to ensure that the IP address of each offloaded pod is added to a `FirewallConfiguration` CR, thereby allowing it to be contacted by its corresponding consumer cluster's gateway. This specific `FirewallConfiguration` CR is applied to the respective gateway pod and encompasses all the IP addresses of the offloaded pods for that particular consumer cluster. The global result is that we maintain N `FirewallConfiguration` CRs, one for each consumer cluster, each applied to its dedicated gateway pod. A critical challenge in this implementation is managing the cleanup of IP addresses from these CRs when shadow pods are deleted, and diligently handling controller restarts (necessitating a cleanup function executed at the controller's startup to remove old IPs). Furthermore, it must correctly manage scenarios where IP addresses are reassigned to other pods that do not belong to the same consumer cluster, ensuring such IPs are promptly removed from the relevant CRs. Future improvements could be to reconcile on more stable resources such as ShadowPods, GatewayServers, GatewayClients, and NamespaceOffloadings, rather than reconciling directly on pods, which are more ephemeral and numerous. This shift would reduce the overall number of reconciliation events, improving the controller's scalability and performance. Additionally, leveraging the controller-runtime cache more effectively—by minimizing unnecessary direct API server calls and relying instead on indexed lookups or cached informer data—could lead to better responsiveness and reduced load on the Kubernetes API server.

ShadowPod Controller Code

Following is reported the code of the reconcile function of the ShadowPod Controller, which is responsible for adding the IP address of each offloaded pod to a `FirewallConfiguration` CR. This code is written in Go and uses the controller-runtime library to interact with the Kubernetes API server.

```
1  func (r *FirewallReconciler) Reconcile(ctx context.Context, req
       ctrl.Request) (ctrl.Result, error) {
2    var pod v1.Pod
3    err := r.Get(ctx, req.NamespacedName, &pod)
4    if apierrors.IsNotFound(err) {
5      klog.Infof("Pod %s not found, possibly deleted", req.
       NamespacedName)
6      return ctrl.Result{}, nil
7    }
8    if err != nil {
9      klog.Errorf("Error retrieving pod %s: %v", req.NamespacedName,
       err)
10     return ctrl.Result{}, fmt.Errorf("failed to retrieve pod: %w",
       client.IgnoreNotFound(err))
11   }
12   if pod.Status.PodIP == "" {
13     return ctrl.Result{}, nil
14   }
15
16   // list of offloaded pods
17   offloadedPodList, err := r.GetOffloadedPods(ctx)
18   if err != nil {
19     klog.Errorf("Error retrieving offloaded pods: %v", err)
20   }
21
22   firewallCluster := utils.FirewallCluster{}
23   utils.AddPodToFirewallCluster(&offloadedPodList, &
       firewallCluster)
24
25   if err != nil {
26     klog.Errorf("Error setting cluster list for offloaded pods: %v
       ", err)
27     return ctrl.Result{}, fmt.Errorf("%w: %w", errors.
       ErrFailedToGetShadowPodList, err)
28   }
29
30   tableName := "firewall-table"
31   chainName := "firewall-chain"
32
33   originCluster := pod.Labels[consts.RemoteClusterID]
34
35   if len(firewallCluster.ClusterPodList) == 0 {
```

```
36      if err := r.createEmptyNFTtable(ctx, originCluster, chainName,
        tableName); err != nil {
37          klog.Errorf("Error creating empty NFT table: %v", err)
38          return ctrl.Result{}, err
39      }
40    } else {
41      if err := r.updateClusterFirewallConfiguration(ctx,
       firewallCluster.ClusterPodList, chainName, tableName); err !=
       nil {
42          klog.Errorf("Error updating cluster firewall configuration:
       %v", err)
43          return ctrl.Result{}, err
44       }
45    }
46
47    return ctrl.Result{}, nil
48  }
```

Listing 4.6: ShadowPod Controller Code

Namespace Controller (Egress Policy)

This controller reconciles on namespaces. Its primary function is to generate and apply `NetworkPolicy` CRs for each offloaded namespace in the provider cluster. These policies are designed to allow egress traffic only to pods within namespaces that are owned by the same consumer cluster, ensuring that offloaded pods can communicate with each other while preventing unauthorized access from other clusters. With this approach, the provider cluster can effectively isolate traffic between different consumer clusters, allowing only the necessary communication between offloaded pods that belong to the same consumer cluster. There is no need to apply `NetworkPolicy` CRs for local namespaces of the provider cluster, as they are not offloaded and do not require the same level of isolation as offloaded namespaces. The controller will also ensure that the `NetworkPolicy` CRs are applied to the correct namespaces and that they are updated whenever new offloaded pods (and the relative namespace) are added or removed.

Namespace Controller Alternative (Ingress Policy)

The actual implementation is to reconcile on namespace applying a default-deny egress approach , which blocks all traffic from the consumer cluster to the provider cluster. This is achieved by applying a `NetworkPolicy` CR that denies all traffic from the consumer cluster to the provider cluster, ensuring that no unauthorized access is allowed until further rules are applied by the Liqo Security Agent. In this case, all connection outgoing from the offloaded namespaces by the consumer cluster will be blocked, and the only traffic allowed will be the one that is explicitly whitelisted by the Liqo Security Agent. An alternative approach could be to apply an ingress-deny policy, but the problem is that would require ‘N‘ `NetworkPolicy` CRs, one for each offloaded namespace in addiction to one for each local namespace of the provider cluster, which would be too much overhead. Instead, as we said, we can apply NetworkPolicy only for the offloaded namespaces.

Namespace Controller Code

Following is reported the code of the reconcile function of the Namespace Controller, which is responsible for generating and applying `NetworkPolicy` CRs for each offloaded namespace in the provider cluster. This code is written in Go and uses the controller-runtime library to interact with the Kubernetes API server.

```
1  func (r *NetworkPolicyReconciler) Reconcile(ctx context.Context,
       req ctrl.Request) (ctrl.Result, error) {
2    var ns corev1.Namespace
3    if err := r.Get(ctx, req.NamespacedName, &ns); err != nil {
4      klog.Errorf("Namespace %q not found", req.Name)
5      return ctrl.Result{}, fmt.Errorf("failed to get namespace %q:
       %w", req.Name, err)
6    }
7
8    klog.Infof("Reconciling NetworkPolicy for namespace %s", req.
       Name)
9
10   originCluster := ns.Labels[consts.RemoteClusterID]
11
12   nw, err := liqogetters.GetNetworksByLabel(ctx, r.Client,
13     labels.SelectorFromSet(map[string]string{
14       consts.RemoteClusterID:              originCluster,
15       liqoControllerConsts.LabelCIDRType: "pod",
16     }), "liqo-tenant-"+originCluster)
17   if err != nil {
18     klog.Errorf("failed to get Network %s: %v", originCluster, err
       )
19     return ctrl.Result{}, fmt.Errorf("failed to get Network %s: %w
       ", originCluster, err)
20   }
21
22   if len(nw) == 0 {
23     klog.Infof("%s is not a tenant namespace from %s", req.Name,
       originCluster)
24     return ctrl.Result{}, nil
25   }
26
27   // Applica la network policy di default
28   netPol := &networkingv1.NetworkPolicy{
29     ObjectMeta: metav1.ObjectMeta{
30       Name:      "allow-from-cluster-" + ns.Labels[consts.
       RemoteClusterID],
31       Namespace: req.Name,
32     },
33   }
34   mutateFn := r.mutateFnNetworkPolicies(
35     &nw[0],
```

```
36      ns.Labels[consts.RemoteClusterID],
37      req.Name,
38      netPol,
39    )
40
41    res, err := controllerutil.CreateOrUpdate(ctx, r.Client, netPol,
        mutateFn)
42    if err != nil {
43      klog.Errorf("Failed to create or update NetworkPolicy %s in
      namespace %s: %v", netPol.Name, req.Name, err)
44      return ctrl.Result{}, fmt.Errorf("error during CreateOrUpdate
      for NetworkPolicy %s in namespace %s: %w", netPol.Name, req.
      Name, err)
45    }
46    if res != controllerutil.OperationResultNone {
47      klog.Infof("NetworkPolicy %s in namespace %s was %s", netPol.
      Name, req.Name, res)
48    }
49
50    return ctrl.Result{}, nil
51  }
```

Listing 4.7: Namespace Controller Code

4.2.2 Tests

To ensure the correctness and reliability of the implemented controllers, a comprehensive suite of tests has been developed. Various bash scripts have been created to validate the functionality of each controller, ensuring that they behave as expected in different scenarios. These tests cover a wide range of cases, facilitating the identification of potential issues and ensuring that the controllers can handle various edge cases effectively. The tests are designed to be run in a controlled environment, allowing for easy verification of the controllers' behavior and ensuring that they meet the desired specifications. A tipical example of a test is to create a new namespace, offload it to the provider cluster, and verify that the corresponding `NetworkPolicy` CR is created and applied correctly. Another example is to create a new pod in an offloaded namespace and verify that its IP address is added to the appropriate `FirewallConfiguration` CR.

```
1  pingPodToPod() {
2      local CLUSTER_SOURCE_ID="$1"
3      local CLUSTER_SOURCE_NAMESPACE="$2"
4      local CLUSTER_DEST_ID="$3"
5      local CLUSTER_DEST_NAMESPACE="$4"
6      local KUBECONF_SOURCE="${CLUSTERS[$CLUSTER_SOURCE_ID]}"
7      export KUBECONF="$KUBECONF_SOURCE"
8
9      local pod_name_source
10     pod_name_source=$(kubectl --kubeconfig="$KUBECONF_SOURCE" get
       pod -n "$CLUSTER_SOURCE_NAMESPACE" -l app=netshoot -o jsonpath
       ='{.items[0].metadata.name}' 2>/dev/null)
11     if [[ -z "$pod_name_source" ]]; then
12         echo "[WARNING] No netshoot pod found in namespace
       $CLUSTER_SOURCE_NAMESPACE (Cluster $CLUSTER_SOURCE_ID)" >&2
13         return 1
14     fi
15
16     local DEST_MAP_NAME="POD_IP_MAP_CLUSTER${CLUSTER_DEST_ID}"
17     local pod_ip_dest
18     eval "pod_ip_dest=\${$DEST_MAP_NAME[\"$CLUSTER_DEST_NAMESPACE
       \"]}"
19     if [[ -z "$pod_ip_dest" ]]; then
20         echo "[WARNING] No netshoot pod found in namespace
       $CLUSTER_DEST_NAMESPACE (Cluster $CLUSTER_DEST_ID) trovato."
       >&2
21         return 1
22     fi
23     local SOURCE_MAP_NAME="POD_IP_MAP_CLUSTER${CLUSTER_SOURCE_ID}"
24     local source_ip
25     eval "source_ip=\${$SOURCE_MAP_NAME[\"
       $CLUSTER_SOURCE_NAMESPACE\"]}"
26
```

```
27      if [[ $source_ip == $pod_ip_dest ]]; then
28          return 0
29      fi
30
31      ping_output=$(kubectl --kubeconfig="$KUBECONF_SOURCE" exec -n
        "$CLUSTER_SOURCE_NAMESPACE" "$pod_name_source" -- ping -c 1 -W
        1 "$pod_ip_dest")
32
33
34
35      if [[ "$ping_output" =~ "1 received" ]]; then
36          result="OK"
37      else
38          result="FAIL"
39      fi
40
41      MATRIX_ROWS+=("$result | [Cluster $CLUSTER_SOURCE_ID -
        $CLUSTER_SOURCE_NAMESPACE] -> [Cluster $CLUSTER_DEST_ID -
        $CLUSTER_DEST_NAMESPACE] || $source_ip -> $pod_ip_dest")
42
43  }
```

Listing 4.8: Example of ping test between offloaded pods

where params are defines as follows:

- `CLUSTER_SOURCE_ID = $1`: The ID of the source cluster, passed as the first argument to the function.
- `CLUSTER_SOURCE_NAMESPACE = $2`: The namespace within the source cluster, passed as the second argument.
- `CLUSTER_DEST_ID = $3`: The ID of the destination cluster, passed as the third argument.
- `CLUSTER_DEST_NAMESPACE = $4`: The namespace within the destination cluster, passed as the fourth argument.
- `KUBECONF_SOURCE = ${CLUSTERS[$CLUSTER_SOURCE_ID]}`: The path to the kubeconfig file associated with the source cluster, retrieved from the `CLUSTERS` associative array using the source cluster ID as key.

The `pingPodToPod` function attempts to ping a pod in the destination cluster from each pods that belong to the consumer cluster (local and offloaded), verifying connectivity between offloaded pods across clusters. The results of the tests are stored in the `MATRIX_ROWS` array, which can be printed or processed further to analyze the test outcomes. It is used in the following loop:

```
1  pingBetweenClusters() {
2      local SOURCE_CLUSTER_ID="$1"
3      local DEST_CLUSTER_ID="$2"
4      local -n source_ips="POD_IP_MAP_CLUSTER${SOURCE_CLUSTER_ID}"
5      local -n dest_ips="POD_IP_MAP_CLUSTER${DEST_CLUSTER_ID}"
6
7      echo "--- Cluster $SOURCE_CLUSTER_ID -> Cluster
    $DEST_CLUSTER_ID ---"
8      for source_ns in "${!source_ips[@]}"; do
9          for dest_ns in "${!dest_ips[@]}"; do
10             pingPodToPod "$SOURCE_CLUSTER_ID" "$source_ns" "
    $DEST_CLUSTER_ID" "$dest_ns"
11         done
12     done
13 }
```

Listing 4.9: Example of ping test between all pods in two clusters

where params are defines as follows:

- `SOURCE_CLUSTER_ID = $1`: The source cluster ID, passed as the first argument to the function.
- `DEST_CLUSTER_ID = $2`: The destination cluster ID, passed as the second argument.
- `source_ips`: A nameref (using `local -n`) to the associative array holding the pod IP addresses for the source cluster, stored in the variable `POD_IP_MAP_CLUSTER{SOURCE_CLUSTER_ID}`.
- `dest_ips`: A nameref (using `local -n`) to the associative array holding the pod IP addresses for the destination cluster, stored in the variable `POD_IP_MAP_CLUSTER{DEST_CLUSTER_ID}`.

Chapter 5

Conclusions and Future Work

The growing adoption of federated Kubernetes platforms like Liqo introduces significant security challenges, particularly in controlling traffic across clusters and enforcing strict tenant isolation. This thesis addressed these issues by designing and implementing a hybrid security enforcement mechanism within the Liqo ecosystem. The proposed architecture combines low-level packet filtering via `nftables` with high-level traffic control through Kubernetes NetworkPolicies. At the network perimeter, nftables rules are applied on gateway Pods in the provider cluster to restrict ingress and egress traffic based on the consumer cluster identity and namespace ownership. These rules are dynamically managed by a custom Gateway Controller that ensures only authorized communication is permitted while optimizing for performance. Within the provider cluster, offloaded workloads are placed in dedicated namespaces protected by Kubernetes NetworkPolicies. These policies enforce intra-cluster isolation, allowing communication only between Pods that belong to the same offloading tenant. The enforcement is automated through two additional controllers: the ShadowPod Controller, which reconciles rules in response to offloaded Pod lifecycle events, and the Namespace Controller, which dynamically manages NetworkPolicies for offloaded namespaces. This hybrid and automated architecture provides robust protection against unauthorized access and lateral movement, ensuring that consumer clusters can only interact with their own offloaded resources. It delivers a scalable, reactive, and fault-tolerant framework that adapts to topology changes without manual reconfiguration. The implemented solution demonstrates that integrating nftables with Kubernetes-native constructs significantly strengthens the security posture of multi-tenant federated environments.

5.1 Future Work

The implementation of the security enforcement mechanism in Liqo has successfully addressed the challenges of inter-cluster traffic control and tenant isolation. However, there are several areas that could benefit from further improvement and enhancement. One direction could be the integration with additional container network interfaces (CNIs), beyond Cilium and Calico, to complement the existing network-level. Another important aspect is performance optimization, which involves conducting benchmarks and applying improvements to ensure that nftables rules and NetworkPolicies can scale effectively in large deployments while maintaining minimal latency and overhead. Expanding the testing and validation framework, particularly to cover a wider range of scenarios such as consumer protection, cross-cluster failure recovery, and resilience against invalid configurations. Scalability improvements could be achieved by refactoring the controllers to observe higher-level and less frequent custom resources, such as ShadowPods, GatewayServers, GatewayClients, and NamespaceOffloadings, instead of low-level, high-frequency resources like Pods. This would reduce the load on the controllers and enhance reconciliation efficiency. Additionally, optimizing the use of the controller-runtime cache would help limit unnecessary API server calls and improve overall performance, especially in environments with a high number of namespaces and pods. Finally, simplifying the deployment process by offering a Helm chart or static manifests for the Liqo Security Agent, with minimal required RBAC permissions, would make adoption easier and lower the risk surface in security-sensitive scenarios.

Bibliography

[1] Kubernetes. *Webhooks - K8s Documentation*. 2025. URL: https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/ (cit. on p. 7).

[2] Liqo Project. *Peering - Liqo Documentation*. 2025. URL: https://docs.liqo.io/en/stable/features/peering.html (cit. on p. 10).

[3] Liqo Project. *Offloading - Liqo Documentation*. 2025. URL: https://docs.liqo.io/en/stable/features/offloading.html (cit. on pp. 10–12).

[4] Liqo Project. *Controller Manager - Liqo Documentation*. 2025. URL: https://docs.liqo.io/en/stable/features/network-fabric.html (cit. on p. 11).

[5] Kubernetes. *Node Level Traffic - K8s Documentation*. 2025. URL: https://kubernetes.io/docs/concepts/services-networking/network-policies/ (cit. on p. 21).

[6] Liqo Project. *Network Fabric - Liqo Documentation*. 2025. URL: https://docs.liqo.io/en/stable/features/network-fabric.html.

[7] Liqo Project. *Gateway Custom Resource Definition - Liqo Documentation*. 2025. URL: https://docs.liqo.io/en/v1.0.0/advanced/peering/inter-cluster-network.html#gateway-crds.

[8] Netfilter Project. *nftables - Netfilter Documentation*. 2025. URL: https://netfilter.org/projects/nftables/.

[9] WikiNftables. *wiki nftables*. 2025. URL: https://wiki.nftables.org/wiki-nftables/index.php/Quick_reference-nftables_in_10_minutes.

[10] Liqo Project. *Configure a Liqo gateway server behind a NAT*. 2025. URL: https://docs.liqo.io/en/latest/advanced/nat.html.

[11] Dan Winship. *NFTables mode for kube-proxy*. 2025. URL: https://kubernetes.io/blog/2025/02/28/nftables-kube-proxy/.

[12] Kubernetes SIG-Network. *KEP-3866: Add an nftables-based kube-proxy backend*. 2025. URL: https://github.com/kubernetes/enhancements/blob/master/keps/sig-network/3866-nftables-proxy/README.md.

[13] Project Calico. *Data plane guide: nftables – Calico Documentation*. 2025. URL: https://docs.tigera.io/calico/latest/getting-started/kubernetes/nftables.

[14] Mageshwaran Sekar. *nftables in the Kubernetes World*. 2025. URL: https://dev.to/mageshwaransekar/nftables-in-the-kubernetes-world-21k9.

[15] Liqo Project. *Requirements — Liqo Documentation*. 2025. URL: https://docs.liqo.io/en/v0.5.1/installation/requirements.html.

[16] Liqo Project. *Service Continuity - Liqo Documentation*. 2025. URL: https://docs.liqo.io/en/v1.0.0/usage/service-continuity.html.

[17] Politecnico di Torino. *Fine-grained security for connectivity in Liqo (Master's thesis)*. 2023. URL: https://webthesis.biblio.polito.it/28635/1/tesi.pdf.

[18] Google. *google/nftables - Go bindings for nftables*. 2025. URL: https://github.com/google/nftables.